User Mode Linux®

BRUCE PERENS' OPEN SOURCE SERIES

www.prenhallprofessional.com/perens

Bruce Perens' Open Source Series is a definitive series of books on Linux and open source technologies, written by many of the world's leading open source professionals. It is also a voice for up-and-coming open source authors. Each book in the series is published under the Open Publication License (www.opencontent.org), an open source compatible book license, which means that electronic versions will be made available at no cost after the books have been in print for six months.

- *Java™ Application Development on Linux®*
  Carl Albing and Michael Schwarz
- *C++ GUI Programming with Qt 3*
  Jasmin Blanchette and Mark Summerfield
- *Managing Linux Systems with Webmin: System Administration and Module Development*
  Jamie Cameron
- *Understanding the Linux Virtual Memory Manager*
  Mel Gorman
- *PHP 5 Power Programming*
  Andi Gutmans, Stig Bakken, and Derick Rethans
- *Linux® Quick Fix Notebook*
  Peter Harrison
- *Implementing CIFS: The Common Internet File System*
  Christopher Hertel
- *Open Source Security Tools: A Practical Guide to Security Applications*
  Tony Howlett
- *Apache Jakarta Commons: Reusable Java™ Components*
  Will Iverson
- *Linux® Patch Management: Keeping Linux® Systems Up To Date*
  Michael Jang
- *Embedded Software Development with eCos*
  Anthony Massa
- *Rapid Application Development with Mozilla*
  Nigel McFarlane
- *Subversion Version Control: Using the Subversion Version Control System in Development Projects*
  William Nagel
- *Intrusion Detection with SNORT: Advanced IDS Techniques Using SNORT, Apache, MySQL, PHP, and ACID*
  Rafeeq Ur Rehman
- *Cross-Platform GUI Programming with wxWidgets*
  Julian Smart and Kevin Hock with Stefan Csomor
- *Samba-3 by Example, Second Edition: Practical Exercises to Successful Deployment*
  John H. Terpstra
- *The Official Samba-3 HOWTO and Reference Guide, Second Edition*
  John H. Terpstra and Jelmer R. Vernooij, Editors
- *Self-Service Linux®: Mastering the Art of Problem Determination*
  Mark Wilding and Dan Behman

User Mode Linux®

Jeff Dike

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Dike, Jeff.
User Mode Linux / Jeff Dike.
p. cm.
Includes bibliographical references and index.
ISBN 0-13-186505-6 (pbk. : alk. paper)
1. Linux. 2. Operating systems (Computers) 3. Application software
porting. I. Title.
QA76.76.O63D545 2006
005.4'32--dc22

2006004225

ISBN 0-13-186505-6
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, April 2006

Contents

Preface

When I started the User Mode Linux (UML) project in 1999, I had no idea how large a project it would become or how much of my time it would end up consuming. As time went on, the UML user base grew, and people found new ways to use it. As a result of their requests, UML contains a number of features that would never have occurred to me.

This book concentrates on the use of UML rather than its internals or plans for the future. I've tried to make it as easy as possible to get started with UML and put to good use all of the features my users induced me to add. Of course, I couldn't resist going into how UML works and what I have planned for its future. That would be too much to ask of any developer of any project. I hope this content adds to the book and the readers' understanding and appreciation of UML.

Acknowledgments

It is not much of an exaggeration to say that the User Mode Linux (UML) project would not exist without its users. They provide testing, bug reports, and suggestions for improvements. Therefore, I would first like to thank everyone who has used UML for taking part, especially those who have tested bleeding-edge versions and who have provided feedback, good or bad. I would especially like to thank Bill Stearns, who has supported UML in innumerable ways since its early days.

A number of people have made contributions to the UML code base by fixing bugs or by contributing new features. Some important features were contributed by users who saw a need and wrote the code. My thanks go out to them. Most significant are the contributions of Paolo Giarrusso, who has become my right-hand man during the last year or so. His contributions include bug fixes and features, documentation and support, and improvements to the hosts in order to allow them to better support UML.

The UML project has been supported financially by a number of organizations, some of whom contracted for specific improvements; others provided more general support. Among these, I would especially like to thank the Dartmouth Institute for Security Technology Studies, which saw in UML the potential for a new and powerful security tool.

Thanks also go to Intel Corporation for hiring me to work on UML full time, and especially for tolerating a significant amount of that time going toward writing this book.

About the Author

Jeff Dike grew up in rural northwest Connecticut. He graduated from MIT and went to work at Digital Equipment Corporation in New Hampshire. There he met several people who became prominent in the Linux world, including Jon Hall and a large contingent that now works at Red Hat. Jeff left Digital in 1993 during the implosion of the minicomputer market. He spent the next decade as an independent contractor and became a Linux kernel developer in 1999 after conceiving of and implementing UML. Since then, UML has been his job, becoming a full-time paid one in mid-2004 when Intel hired him.

CHAPTER 1

Introduction

WHAT IS UML?

User Mode Linux (UML) is a virtual Linux machine that runs on Linux. Technically, UML is a port of Linux to Linux. Linux has been ported to many different processors, including the ubiquitous x86, Sun's SPARC, IBM and Motorola's PowerPC, DEC's (then Compaq's and HP's) Alpha, and a variety of others. UML is a port of Linux in exactly the same sense as these. The difference is that it is a port to the software interface defined by Linux rather than the hardware interface defined by the processor and the rest of the physical computer.

UML has manifold uses for system administrators, users, and developers. UML virtual machines are useful for test environments that can be set up quickly and thrown away when no longer needed, production environments that efficiently use the available hardware, development setups that can make it much more convenient to test software, plus a surprising number of other things.

COMPARISON WITH OTHER VIRTUALIZATION TECHNOLOGIES

UML differs from other virtualization technologies in being more of a virtual operating system (OS) rather than a virtual machine. In spite of this, I will stick to the common terminology and call UML a virtual machine technology rather than a virtual OS, which would be somewhat more accurate.

Technologies such as VMWare really are virtual machines. They emulate a physical platform, from the CPU to the peripherals, well enough that any OS that runs on the physical platform also runs on the emulated platform provided by VMWare. This has the advantage that it is fairly OS-agnostic—in principle, any OS that runs on the platform can boot under VMWare. In contrast, UML can be only a Linux guest. On the other hand, being a virtual OS rather than a virtual machine allows UML to interact more fully with the host OS, which has advantages we will see later.

Other virtualization technologies such as Xen, BSD jail, Solaris zones, and `chroot` are integrated into the host OS, as opposed to UML, which runs in a process. This gives UML the advantage of being independent from the host OS version, at the cost of some performance. However, a lot (maybe all) of this performance can be regained without losing the flexibility and manageability that UML gains from being in userspace.

As we will see later, the benefits of virtualization accrue largely from the degree of isolation between users and processes inside the virtual machine or jail and those outside it. Most of these technologies (excluding Xen and VMWare) provide only partial virtualization and, thus, partial isolation.

The least complete virtualization is provided by `chroot`, which only jails processes into a directory. In all other respects, the processes are unconfined. Even then, on Linux, `chroot` can't confine a process with root privileges, since its design allows superuser processes to escape.

BSD jail and vserver (a Linux-based project with roughly the same properties) provide stronger confinement. They confine processes to a subset of the filesystem and don't allow them to see processes outside the jail. A jail is also restricted to using a single IP address, and it can't manipulate its firewall rules. Jailed processes are not restricted in their use of CPU time or I/O. The jails on a system are implemented within the system's kernel and therefore share the kernel, along with

the bugs and security holes it contains. The inability to change firewall rules is a consequence of incomplete virtualization, as is the requirement to share the kernel with the host.

Solaris zones are much closer to full-blown virtual machines and complete isolation. Processes within a zone can't see outside files or processes, as is the case with a jail. Zones have their own logical devices, with some restrictions on their access to the network. For example, raw access to packets isn't allowed. A zone can be assigned a certain number of shares within the global fair share scheduler, limiting the share of CPU that the processes within a zone can consume. We will see this concept later in the form of virtual processors in a multiprocessor virtual machine. Zones, like the other technologies described so far, are implemented within the kernel and share the kernel version and configuration with each other and the host.

Finally, technologies such as VMWare, Xen, and UML implement full virtualization and isolation. They all have fully virtualized devices with no restrictions on how they may be used. They also confine their processes with respect to CPU consumption by virtue of having a certain number of virtual processors they may use. They also all run separate instances of the OS, which may be different versions (and even a completely different OS in the case of VMWare) than the host.

WHY VIRTUAL MACHINES?

A UML instance is a full-fledged Linux machine running on the host Linux. It runs all the software and services that any other Linux machine does. The difference is that UML instances can be conjured up on demand and then thrown away when not needed. This advantage lies behind the large range of applications that I and other people have found for UML.

In addition to the flexibility of being able to create and destroy virtual machines within seconds, the instances themselves can be dynamically reconfigured. Virtual peripherals, processors, and memory can be added and removed arbitrarily to and from a running UML instance.

There are also much looser limits on hardware configurations for UML instances than for physical machines. In particular, they are not limited to the hardware they are running on. A UML instance may have more memory, more processors, and more network interfaces, disks, and other devices than its host, or even any possible host. This

makes it possible to test software for hardware you don't own, but have to support, or to configure software for a network before the network is available.

In this book, I will describe the many uses of UML and provide step-by-step instructions for using it. In doing so, I will provide you, the reader, with the information and techniques needed to make full use of UML. As the original author and current maintainer of UML, I have seen UML mature from its decidedly cheesy beginnings to its current state where it can do basically everything that any other Linux machine can do (see Table 1.1).

A Bit of History

I started working on UML in earnest in February 1999 after having the idea that porting Linux to itself might be practical. I tossed the idea around in the back of my head for a few months in late 1998 and early 1999. I was thinking about what facilities it would need from the host and whether the system call interface provided by Linux was rich enough to provide those facilities. Ultimately, I decided it probably was, and in the cases where I wasn't sure, I could think of workarounds.

So, around February, I pulled a copy of the 2.0.32 kernel tree off of a Linux CD (probably a Red Hat source CD) because it was too painful to try to download it through my dialup. Within the resulting kernel tree, I created the directories my new port was going to need without putting any files in them. This is the absolute minimum amount of infrastructure you need for a new port. With the directories present, the kernel build process can descend into them and try to build what's there.

Table 1.1 UML Development Timeline

Date	Event
Late 1998 to early 1999	I think about whether UML is possible.
Feb. 1999	I start working on UML.
June 3, 1999	UML is announced to the Linux kernel mailing list.
Sept. 12, 2002	UML is merged into 2.5.34.
June 21, 2004	I join Intel.

Needless to say, with nothing in those directories, the build didn't even start to work. I needed to add the necessary build infrastructure, such as Makefiles. So, I added the minimal set of things needed to get the kernel build to continue and looked at what failed next. Missing were a number of header files used by the generic (hardware-independent) portions of the kernel that the port needs to provide. I created them as empty files, so that the `#include` preprocessor directives would at least succeed, and proceeded onward.

At this point, the kernel build started complaining about missing macros, variables, and functions—the things that should have been present in my empty header files and nonexistent C source files. This told me what I needed to think about implementing. I did so in the same way as before: For the most part, I implemented the functions as stubs that didn't do anything except print an error message. I also started adding real headers, mostly by copying the x86 headers into my `include` directory and removing the things that had no chance of compiling.

After defining many of these useless procedures, I got the UML build to "succeed." It succeeded in the sense that it produced a program I could run. However, running it caused immediate failures due to the large number of procedures I defined that didn't do what they were supposed to—they did nothing at all except print errors. The utility of these errors is that they told me in what order I had to implement these things for real.

So, for the most part, I plodded along, implementing whatever function printed its name first, making small increments of progress through the boot process with each addition. In some cases, I needed to implement a subsystem, resulting in a related set of functions.

Implementation continued in this vein for a few months, interrupted by about a month of real, paying work. In early June, I got UML to boot a small filesystem up to a login prompt, at which point I could log in and run commands. This may sound impressive, but UML was still bug-ridden and full of design mistakes. These would be rooted out later, but at the time, UML was not much more than a proof of concept.

Because of design decisions made earlier, such fundamental things as shared libraries and the ability to log in on the main console didn't work. I worked around the first problem by compiling a minimal set of tools statically, so they didn't need shared libraries. This minimal set of tools was what I populated my first UML filesystem with. At the time of my announcement, I made this filesystem available for download since it was the only way anyone else was going to get UML to boot.

Because of another design decision, UML, in effect, put itself in the background, making it impossible for it to accept input from the terminal. This became a problem when you tried to log in. I worked around this by writing what amounted to a serial line driver, allowing me to attach to a virtual serial line on which I could log in.

These are two of the most glaring examples of what didn't work at that point. The full list was much longer and included other things such as signal delivery and process preemption. They didn't prevent UML from working convincingly, even though they were fairly fundamental problems, and they would get fixed later.

At the time, Linus was just starting the 2.3 development kernel series. My first "UML-ized" kernel was 2.0.32, which, even at the time, was fairly old. So, I bit the bullet and downloaded a "modern" kernel, which was 2.3.5 or so. This started the process, which continues to this day, of keeping in close touch with the current development kernels (and as of 2.4.0, the stable ones as well).

Development continued, with bugs being fixed, design mistakes rectified (and large pieces of code rewritten from scratch), and drivers and filesystems added. UML spent a longer than usual amount of time being developed out of pool, that is, not integrated into the mainline Linus' kernel tree. In part, this was due to laziness. I was comfortable with the development methodology I had fallen into and didn't see much point in changing it.

However, pressure mounted from various sources to get UML into the main kernel tree. Many people wanted to be able to build UML from the kernel tree they downloaded from http://www.kernel.org or got with their distribution. Others, wanting the best for the UML project, saw inclusion in Linus' kernel as being a way of getting some public recognition or as a stamp of approval from Linus, thus attracting more users to UML. More pragmatically, some people, who were largely developers, noted that inclusion in the official kernel would cause updates and bug fixes to happen in UML "automatically." This would happen as someone made a pass over the kernel sources, for example, to change an interface or fix a family of bugs, and would cover UML as part of that pass. This would save me the effort of looking through the patch representing a new kernel release, finding those changes, figuring out the equivalent changes needed in UML, and making them. This had become my habit over the roughly four years of UML development before it was merged by Linus. It had become a routine part of UML development, so I didn't begrudge the time it took,

but there is no denying that it did take time that would have been better spent on other things.

So, roughly in the spring of 2002, I started sending updated UML patches to Linus, requesting that they be merged. These were ignored for some months, and I was starting to feel a bit discouraged, when out of the blue, he merged my 2.5.34 patch on September 12, 2002. I had sent the patch earlier to Linus as well as the kernel mailing list and one of my own UML lists, as usual, and had not thought about it further. That day, I was idling on an Internet Relay Chat (IRC) channel where a good number of the kernel developers hang around and talk. Suddenly, Arnaldo Carvalho de Melo (a kernel contributor from Brazil and the CTO of Conectiva, the largest Linux distribution in South America) noticed that Linus had merged my patch into his tree.

The response to this from the other kernel hackers, and a little later, from the UML community and wider Linux community, was gratifying positive. A surprisingly (to me) large number of people were genuinely happy that UML had been merged, and, in doing so, got the recognition they thought it deserved.

At this writing, it is three years later, and UML is still under very active development. There have been ups and downs. Some months after UML was merged, I started finding it hard to get Linus to accept updated patches. After a number of ignored patches, I started maintaining UML out of tree again, with the effect that the in-tree version of UML started to bit-rot. It stopped compiling because no one was keeping it up to date with changes to internal kernel interfaces, and of course bugs stopped being fixed because my fixes weren't being merged by Linus.

Late in 2004, an energetic young Italian hacker named Paolo Giarrusso got Andrew Morton, Linus' second-in-command, to include UML in his tree. The so-called "-mm" tree is a sort of purgatory for kernel patches. Andrew merges patches that may or may not be suitable for Linus' kernel in order to give them some wider exposure and see if they are suitable. Andrew took patches representing the current UML at the time from Paolo, and I followed that up with some more patches. Presently, Andrew forwarded those patches, along with many others, to Linus, who included them in his tree. All of a sudden, UML was up to date in the official kernel tree, and I had a reliable conduit for UML updates.

I fed a steady stream of patches through this conduit, and by the time of the 2.6.9 release, you could build a working UML from the official tree, and it was reasonably up to date.

Throughout this period, I had been working on UML on a volunteer basis. I took enough contracting work to keep the bills paid and the cats fed. Primarily, this was spending a day a week at the Institute for Security Technology Studies at Dartmouth College, in northern New Hampshire, about an hour from my house. This changed around May and June of 2004, when, nearly simultaneously, I got job offers from Red Hat and Intel. Both were very generous, offering to have me spend my time on UML, with no requirements to move. I ultimately accepted Intel's offer and have been an Intel employee in the Linux OS group since.

Coincidentally, the job offers came on the fifth anniversary of UML's first public announcement. So, in five years, UML went from nothing to a fully supported part of the official Linux kernel.

WHAT IS UML USED FOR?

During the five years since UML began, I have seen steady growth in the UML user base and in the number and variety of applications and uses for UML. My users have been nothing if not inventive, and I have seen uses for UML that I would never have thought of.

Server Consolidation

Naturally, the most common applications of UML are the obvious ones. Virtualization has become a hot area of the computer industry, and UML is being used for the same things as other virtualization technologies. Server consolidation is a major one, both internally within organizations and externally between them. Internal consolidation usually takes the form of moving several physical servers into the same number of virtual machines running on a single physical host. External consolidation is usually an ISP or hosting company offering to rent UML instances to the public just as they rent physical servers. Here, multiple organizations end up sharing physical hardware with each other.

The main attraction is cost savings. Computer hardware has become so powerful and so cheap that the old model of one service, or maybe two, per machine now results in hardware that is almost totally idle. There is no technical reason that many services, and their data and configurations, couldn't be copied onto a single server. However, it is easier in many cases to copy each entire server into a virtual machine

and run them all unchanged on a single host. It is less risky since the configuration of each is the same as on the physical server, so moving it poses no chance of upsetting an already-debugged environment.

In other cases, virtual servers may offer organizational or political benefits. Different services may be run by different organizations, and putting them on a single physical server would require giving the root password to each organization. The owner of the hardware would naturally tend to feel queasy about this, as would any given organization with respect to the others. A virtual server neatly solves this by giving each service its own virtual machine with its own root password. Having root privileges in a virtual machine in no way requires root privileges on the host. Thus, the services are isolated from the physical host, as well as from each other. If one of them gets messed up, it won't affect the host or the other services.

Moving from production to development, UML virtual machines are commonly used to set up and test environments before they go live in production. Any type of environment from a single service running on a single machine to a network running many services can be tested on a single physical host. In the latter case, you would set up a virtual network of UMLs on the host, run the appropriate services on the virtual hosts, and test the network to see that it behaves properly.

In a complex situation like this, UML shines because of the ease of setting up and shutting down a virtual network. This is simply a matter of running a set of commands, which can be scripted. Doing this without using virtual machines would require setting up a network of physical machines, which is vastly more expensive in terms of time, effort, space, and hardware. You would have to find the hardware, from systems to network cables, find some space to put it in, hook it all together, install and configure software, and test it all. In addition to the extra time and other resources this takes, compared to a virtual test environment, none of this can be automated.

In contrast, with a UML testbed, this can be completely automated. It is possible, and fairly easy, to full automate the configuration and booting of a virtual network and the testing of services running on that network. With some work, this can be reduced to a single script that can be run with one command. In addition, you can make changes to the network configuration by changing the scripts that set it up, rather than rewiring and rearranging hardware. Different people can also work independently on different areas of the environment by booting virtual networks on their own workstations. Doing this in a physical

environment would require separate physical testbeds for each person working on the project.

Implementing this sort of testbed using UML systems instead of physical ones results in the near-elimination of hardware requirements, much greater parallelism of development and testing, and greatly reduced turnaround time on configuration changes. This can reduce the time needed for testing and improve the quality of the subsequent deployment by increasing the amount and variety of testing that's possible in a virtual environment.

A number of open source projects, and certainly a much larger number of private projects, use UML in this way. Here are a couple that I am aware of.

- Openswan (http://www.openswan.org), the open source IPSec project, uses a UML network for nightly regression testing and its kernel development.
- BusyBox (http://www.busybox.net), a small-footprint set of Linux utilities, uses UML for its testing.

Education

Consider moving the sort of UML setup I just described from a corporate environment to an educational one. Instead of having a temporary virtual staging environment, you would have a permanent virtual environment in which students will wreak havoc and, in doing so, hopefully learn something.

Now, the point of setting up a complicated network with interrelated services running on it is simply to get it working in the virtual environment, rather than to replicate it onto a physical network once it's debugged. Students will be assigned to make things work, and once they do (or don't), the whole thing will be torn down and replaced with the next assignment.

The educational uses of UML are legion, including courses that involve any sort of system administration and many that involve programming. System administration requires the students to have root privileges on the machines they are learning on. Doing this with physical machines on a physical network is problematic, to say the least.

As root, a student can completely destroy the system software (and possibly damage the hardware). With the system on a physical network, a student with privileges can make the network unusable by,

wittingly or unwittingly, spoofing IP addresses, setting up rogue DNS or DHCP servers, or poisoning ARP (Address Resolution Protocol)[1] caches on other machines on the network.

These problems all have solutions in a physical environment. Machines can be completely reimaged between boots to undo whatever damage was done to the system software. The physical network can be isolated from any other networks on which people are trying to do real work. However, all this takes planning, setup, time, and resources that just aren't needed when using a UML environment.

The boot disk of a UML instance is simply a file in the host's filesystem. Instead of reimaging the disk of a physical machine between boots, the old UML root filesystem file can be deleted and replaced with a copy of the original. As will be described in later chapters, UML has a technology called COW (Copy-On-Write) files, which allow changes to a filesystem to be stored in a host file separate from the filesystem itself. Using this, undoing changes to a filesystem is simply a matter of deleting the file that contains the changes. Thus, reimaging a UML system takes a fraction of a second, rather than the minutes that reimaging a disk can take.

Looking at the network, a virtual network of UMLs is by default isolated from everything else. It takes effort, and privileges on the host, to allow a virtual network to communicate with a physical one. In addition, an isolated physical network is likely to have a group of students on it, so that one sufficiently malign or incompetent student could prevent any of the others from getting anything done. With a UML instance, it is feasible (and the simplest option) to give each student a private network. Then, an incompetent student can't mess up anyone else's network.

1. ARP is used on an Ethernet network to convert IP addresses to Ethernet addresses. Each machine on an Ethernet network advertises what IP addresses it owns, and this information is stored by the other machines on the network in their ARP caches. A malicious system could advertise that it owns an IP address that really belongs to a different machine, in effect, hijacking the address. For example, hijacking the address of the local name server would result in name server requests being sent to the hijacking machine rather than the legitimate name server. Nearly all Internet operations begin with a name lookup, so hijacking the address of the name server gives an enormous amount of control of the local network to the attacker.

UML is also commonly used for learning kernel-level programming. For novice to intermediate kernel programming students, UML is a perfect environment in which to learn. It provides an authentic kernel to modify, with the development and debugging tools that should already be familiar. In addition, the hardware underneath this kernel is virtualized and thus better behaved than physical hardware. Failures will be caused by buggy software, not by misbehaving devices. So, students can concentrate on debugging the code rather than diagnosing broken or flaky hardware.

Obviously, dealing with broken, flaky, slightly out-of-spec, not-quite-standards-compliant devices are an essential part of an expert kernel developer's repertoire. To reach that exalted status, it is necessary to do development on physical machines. But learning within a UML environment can take you most of the way there.

Over the years, I have heard of education institutions teaching many sort of Linux administration courses using UML. Some commercial companies even offer system administration courses over the Internet using UML. Each student is assigned a personal UML, which is accessible over the Internet, and uses it to complete the coursework.

Development

Moving from system administration to development, I've seen a number of programming courses that use UML instances. Kernel-level programming is the most obvious place for UMLs. A system-level programming course is similar to a system administration course in that each student should have a dedicated machine. Anyone learning kernel programming is probably going to crash the machine, so you can't really teach such a course on a shared machine.

UML instances have all the advantages already described, plus a couple of bonuses. The biggest extra is that, as a normal process running on the host, a UML instance can be debugged with all the tools that someone learning system development is presumably already familiar with. It can be run under the control of `gdb`, where the student can set breakpoints, step through code, examine data, and do everything else you can do with `gdb`. The rest of the Linux development environment works as well with UML as with anything else. This includes `gprof` and `gcov` for profiling and test coverage and `strace` and `ltrace` for system call and library tracing.

Another bonus is that, for tracking down tricky timing bugs, the debugging tool of last resort, the print statement, can be used to dump data out to the host without affecting the timing of events within the UML kernel. With a physical machine, this ranges from extremely hard to impossible. Anything you do to store information for later retrieval can, and probably will, change the timing enough to obscure the bug you are chasing. With a UML instance, time is virtual, and it stops whenever the virtual machine isn't in the host's userspace, as it is when it enters the host kernel to log data to a file.

A popular use for UML is development for hardware that does not yet exist. Usually, this is for a piece of embedded hardware—an appliance of some sort that runs Linux but doesn't expose it. Developing the software inside UML allows the software and hardware development to run in parallel. Until the actual devices are available, the software can be developed in a UML instance that is emulating the hardware.

Examples of this are hard to come by because embedded developers are notoriously close-lipped, but I know of a major networking equipment manufacturer that is doing development with UML. The device will consist of several systems hooked together with an internal network. This is being simulated by a script that runs a set of UML instances (one per system in the device) with a virtual network running between them and a virtual network to the outside. The software is controlling the instances in exactly the same that it will control the systems within the final device.

Going outside the embedded device market, UML is used to simulate large systems. A UML instance can have a very large amount of memory, lots of processors, and lots of devices. It can have more of all these things than the host can, making it an ideal way to simulate a larger system than you can buy. In addition to simulating large systems, UML can also simulate clusters. A couple of open source clustering systems and a larger number of cluster components, such as filesystems and heartbeats, have been developed using UML and are distributed in a form that will run within a set of UMLs.

Disaster Recovery Practice

A fourth area of UML use, which is sort of a combination of the previous two, is disaster recovery practice. It's a combination in the sense that this would normally be done in a corporate environment, but the UML virtual machines are used for training.

The idea is that you make a virtual copy of a service or set of services, mess it up somehow, and figure out how to fix it. There will likely be requirements beyond simply fixing what is broken. You may require that the still-working parts of the service not be shut down or that the recovery be done in the least amount of time or with the smallest number of operations.

The benefits of this are similar to those mentioned earlier. Virtual environments are far more convenient to set up, so these sorts of exercises become far easier when virtual machines are available. In many cases, they simply become possible since hardware can't be dedicated to disaster recovery practice. The system administration staff can practice separately at their desks, and, given a well-chosen set of exercises, they can be well prepared when disaster strikes.

THE FUTURE

This chapter provided a summary of the present state of UML and its user community. This book will also describe what I have planned for the future of UML and what those plans mean for its users.

Among the plans is a project to port UML into the host kernel so that it runs inside the kernel rather than in a process. With some restructuring of UML, breaking it up into independent subsystems that directly use the resources provided by the host kernel, this in-kernel UML can be used for a variety of resource limitation applications such as resource control and jailing.

This will provide highly customizable jailing, where a jail is constructed by combining the appropriate subsystems into a single package. Processes in such a jail will be confined with respect to the resources controlled by the jail, and otherwise unconfined. This structure of layering subsystems on top of each other has some other advantages as well. It allows them to be nested, so that a user confined within a jail could construct a subjail and put processes inside it. It also allows the nested subsystems to use different algorithms than the host subsystems. So, a workload with unusual scheduling or memory needs could be run inside a jail with algorithms suitable for it.

However, the project I'm most excited about is using UML as a library, allowing other applications to link against it and thereby gain a captive virtual machine. This would have a great number of uses:

- ☞ Managing an application or service from the inside, by logging in to the embedded UML

- ☞ Running scripts inside the embedded UML to control, monitor, and extend the application
- ☞ Using clustering technology to link multiple embedded UMLs into a cluster and use scripts running on this cluster to integrate the applications in ways that are currently not possible

CHAPTER 2

A Quick Look at UML

This chapter will take a quick look at the inside of a UML. I will concentrate on the relationship between the UML and the host. For many people, encountering a virtual machine for the first time can be confusing because it may not be clear where the host ends and the virtual machine starts.

For example, the virtual machine obviously is part of the host since it can't exist without the host. However, it is totally separate from the host in other ways. You can be root inside the UML and have no privileges[1] whatsoever on the host. When UML is run, it is provided some host resources to use as its own. The root user within UML has absolute control over those, but no control, not even access, to anything else on the host. It's this extremely sharp distinction between what the UML has access to and what it doesn't that makes UML useful for a large number of applications.

1. In order to run a process, you obviously need some level of privilege on the system. However, a UML host can be set up such that the user that owns the UML processes on the host can do nothing but run the UML process.

A second common source of confusion is the duality of UML. It is both a Linux kernel and a Linux process. It is useful, and instructive, to look at UML from both perspectives. However, to many people, a kernel and a process are two completely different things, and there can be no overlap between them. So, we will look at a UML from both inside and outside, on the host, in order to compare the two views to each other. We will see different views of the same things. They will look different but will both be correct in their own ways. Hopefully, by the end of the chapter, it will be clear how something can be both a Linux kernel and a Linux process.

Figure 2.1 shows the relationship among a UML instance, the host kernel, and UML processes. To the host kernel, the UML instance is a normal process. To the UML processes, the UML instance is a kernel. Processes interact with the kernel by making system calls, which are like procedure calls except that they request the kernel do something on their behalf.

Like all other processes on the host, UML makes system calls to the host kernel in order to do its work. Unlike the other host processes, UML has its own system call interface for its processes to use. This is the source of the duality of UML. It makes system calls to the host, which makes it a process, and it implements system calls for its own processes, making it a kernel.

Let's take a look at the UML binary, which is normally called `linux`:

```
host% ls -l linux
-rwxrw-rw-  2 jdike jdike  23346513 Jan 27 12:16 linux
```

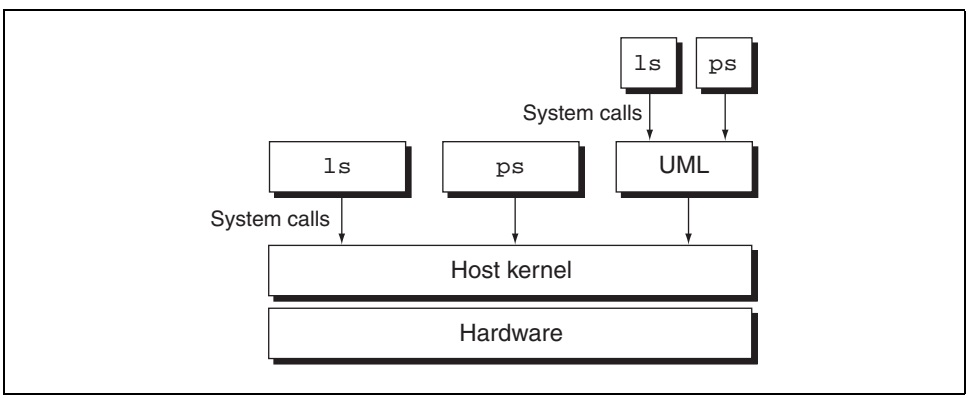


Figure 2.1 UML as both a process and a kernel

This is a normal Linux ELF binary, as you can see by running `file` on it:

```
host% file linux
linux: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), \
for GNU/Linux 2.2.5, statically linked, not stripped
```

It is also a Linux kernel, so it may be instructive to compare it to the kernel running on this machine:

```
host% ls -l /boot/vmlinuz-2.4.26
-rw-r--r--  1 root root 945800 Sep 18 17:12 /boot/vmlinuz-2.4.26
```

The UML binary is quite a bit larger than the kernel on the host, but it has a full symbol table, as you can see from the output of `file` above. So, let's strip it and see what that does:

```
host% ls -l linux
-rwxrw-rw-  2 jdike jdike 2236936 Jan 27 15:01 linux
```

It's a bit more than twice as large as the host kernel, possibly because the configurations are different. I tend to build options into UML, which on the host are modules. Checking this by adding up the sizes of the modules loaded on the host yields this:

```
host% lsmod
Module                  Size  Used by
usblp                  17473  0
parport_pc             31749  1
lp                     16713  0
parport                39561  2 parport_pc,lp
autofs4                23493  2
sunrpc                145541  1
...
host% lsmod | awk '{n += $2} END {print n}'
1147092
```

Adding that to the file size of `vmlinuz-2.4.26` gives us something close to the size of the UML binary after the symbol table has been stripped off.

What is the point of this comparison? It is to introduce the fact that UML is both a Linux kernel and a Linux process. As a Linux process, it can be run just like any other executable on the system, such as `bash` or `ls`.

BOOTING UML FOR THE FIRST TIME

Let's boot UML now:

```
host% ./linux
```

```
Checking for /proc/mm...not found
Checking for the skas3 patch in the host...not found
Checking PROT_EXEC mmap in /tmp...OK
Linux version 2.6.11-rc1-mm1 (jdike@tp.user-mode-linux.org) (gcc version 3.3.2
     20031022 (Red Hat Linux 3.3.2-1)) #83 Thu Jan 27 12:16:00 EST 2005
Built 1 zonelists
Kernel command line: root=98:0
PID hash table entries: 256 (order: 8, 4096 bytes)
Dentry cache hash table entries: 8192 (order: 3, 32768 bytes)
Inode-cache hash table entries: 4096 (order: 2, 16384 bytes)
Memory: 29368k available
Mount-cache hash table entries: 512 (order: 0, 4096 bytes)
Checking for host processor cmov support...Yes
Checking for host processor xmm support...No
Checking that ptrace can change system call numbers...OK
Checking syscall emulation patch for ptrace...missing
Checking that host ptys support output SIGIO...Yes
Checking that host ptys support SIGIO on close...No, enabling workaround
Checking for /dev/anon on the host...Not available (open failed with errno 2)
NET: Registered protocol family 16
mconsole (version 2) initialized on /home/jdike/.uml/3m3vDd/mconsole
VFS: Disk quotas dquot_6.5.1
Dquot-cache hash table entries: 1024 (order 0, 4096 bytes)
io scheduler noop registered
io scheduler anticipatory registered
io scheduler deadline registered
io scheduler cfq registered
NET: Registered protocol family 2
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP established hash table entries: 2048 (order: 2, 16384 bytes)
TCP bind hash table entries: 2048 (order: 1, 8192 bytes)
TCP: Hash tables configured (established 2048 bind 2048)
NET: Registered protocol family 1
NET: Registered protocol family 17
Initialized stdio console driver
Console initialized on /dev/tty0
Initializing software serial port version 1
VFS: Waiting 19sec for root device...
VFS: Waiting 18sec for root device...
VFS: Waiting 17sec for root device...
VFS: Waiting 16sec for root device...
```

Figure 2.2 Output from the first boot of UML

```
VFS: Waiting 15sec for root device...
VFS: Waiting 14sec for root device...
VFS: Waiting 13sec for root device...
VFS: Waiting 12sec for root device...
VFS: Waiting 11sec for root device...
VFS: Waiting 10sec for root device...
VFS: Waiting 9sec for root device...
VFS: Waiting 8sec for root device...
VFS: Waiting 7sec for root device...
VFS: Waiting 6sec for root device...
VFS: Waiting 5sec for root device...
VFS: Waiting 4sec for root device...
VFS: Waiting 3sec for root device...
VFS: Waiting 2sec for root device...
VFS: Waiting 1sec for root device...
VFS: Cannot open root device "98:0" or unknown-block(98,0)
Please append a correct "root=" boot option
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(98,0)

EIP: 0023:[<a015a751>] CPU: 0 Not tainted ESP: 002b:40001fa0 EFLAGS: 00000206
    Not tainted
EAX: 00000000 EBX: 00002146 ECX: 00000013 EDX: 00002146
ESI: 00002145 EDI: 00000000 EBP: 40001fbc DS: 002b ES: 002b
Call Trace:
a0863af0:  [<a0030446>] printk+0x12/0x14
a0863b00:  [<a003ff32>] notifier_call_chain+0x22/0x40
a0863b30:  [<a002f9f2>] panic+0x56/0x108
a0863b40:  [<a003c0f6>] msleep+0x42/0x4c
a0863b50:  [<a0002d96>] mount_block_root+0xd6/0x188
a0863bb0:  [<a0002e9c>] mount_root+0x54/0x5c
a0863bc0:  [<a0002f07>] prepare_namespace+0x63/0xa8
a0863bd0:  [<a0002ebb>] prepare_namespace+0x17/0xa8
a0863bd4:  [<a000e190>] init+0x0/0x108
a0863be4:  [<a000e190>] init+0x0/0x108
a0863bf0:  [<a000e291>] init+0x101/0x108
a0863c00:  [<a0027131>] run_kernel_thread+0x39/0x40
a0863c18:  [<a000e190>] init+0x0/0x108
a0863c28:  [<a0027117>] run_kernel_thread+0x1f/0x40
a0863c50:  [<a0013211>] unblock_signals+0xd/0x10
a0863c70:  [<a002c51c>] finish_task_switch+0x24/0xa4
a0863c84:  [<a000e190>] init+0x0/0x108
a0863c90:  [<a002c5ad>] schedule_tail+0x11/0x124
a0863cc4:  [<a000e190>] init+0x0/0x108
a0863cd0:  [<a001ad58>] new_thread_handler+0xb0/0x104
a0863cd4:  [<a000e190>] init+0x0/0x108
a0863d20:  [<a015a508>] __restore+0x0/0x8
a0863d60:  [<a015a751>] kill+0x11/0x20
```

Figure 2.2 Output from the first boot of UML (*continued*)

Notice two obvious things about the results, shown in Figure 2.2.

1. The output resembles the boot output of a normal Linux machine.
2. The boot was not very successful, as you can see from the panic and stack dump at the end.

It's worth comparing this to the boot output of a Linux system, which is normally available by running `dmesg`. You'll see a lot of similarities—many of the messages, such as the ones from the filesystem and network subsystems, are identical. Much of the rest are totally different, although they should seem similar in purpose. This is largely due to hardware drivers initializing. UML doesn't have the same hardware or drivers as the host, so their bootup messages will be different.

If you have access to Linux on several different architectures, such as x86 and x86_64 or ppc, you'll see the same sorts of differences between their boot output. In fact, this is a very apt comparison because UML is a different architecture from the Linux kernel running on the host.

Let's look at the output in more detail.

```
Checking for /proc/mm...not found
Checking for the skas3 patch in the host...not found
Checking PROT_EXEC mmap in /tmp...OK
```

These are checking the environment on the host to see if it can run at all (the executable `/tmp` check) and whether the host kernel has capabilities that allow UML to run more efficiently. You'll see more of this below, but these particular checks need to be done very early.

```
Checking for host processor cmov support...Yes
Checking for host processor xmm support...No
Checking that ptrace can change system call numbers...OK
```

These are checking some more capabilities of the host. The first two are checking processor capabilities, and the last is checking whether the host has a feature that's absolutely needed for UML to run (which all modern hosts do).

```
mconsole (version 2) initialized on /home/jdike/.uml/3m3vDd/mconsole
...
Initialized stdio console driver
...
Initializing software serial port version 1
```

Here, UML is initializing its drivers. A UML boot has much less output of this sort compared with a boot of a physical Linux system.

This is because UML uses resources on the host to support its virtual hardware, and there are many fewer types of these resources than there are different types of devices on a physical system. For example, every possible sort of block device within UML can be accessed as a host file, so block devices require a single UML driver. In contrast, the host has multitudes of block drivers, for IDE disks, SCSI disks, SATA disks, and so on. Because of the uniform interface provided by the host, UML requires many fewer drivers in order to access these devices and the data on them.

The first driver is the `mconsole`[2] driver, which allows a UML to be controlled and managed from the host. This has no hardware equivalent on most Linux systems. The last two are the console and serial line drivers, which obviously do have hardware equivalents, except that the UML drivers will communicate using virtual devices such as pseudo-terminals rather than physical devices such as a graphics card or serial line.

```
VFS: Waiting 1sec for root device...
VFS: Cannot open root device "98:0" or unknown-block(98,0)
Please append a correct "root=" boot option
Kernel panic - not syncing: VFS: Unable to mount root fs on \
      unknown-block(98,0)
```

Here is the panic that killed off this attempted run of UML. The problem is that we didn't provide UML with a root device, so it couldn't mount its root filesystem. This is fatal and causes the panic and the stack trace. You can make a physical Linux machine do exactly the same thing by putting a bogus `root=` option on the kernel command line using `LILO` or `GRUB`.[3]

Finally, an important point is that we just panicked a UML kernel, and the only result was that we were dropped back to the shell prompt. The host system itself, and everything else on the system, was totally unaffected by the crash. This demonstrates the basis of many of the advantages of UML over a physical system—it can be used in ways that may cause system crashes or other software malfunctions, but the

2. MConsole stands for "Management Console" and is a mechanism for controlling and monitoring a UML instance from the host.

3. UML needs no bootloader like the host needs `LILO` or `GRUB`. As it is run from the command line, you can think of the host as being the UML bootloader.

damage is limited to the virtual machine. As we will see later, even this damage can be undone quite easily.

That may have been interesting, but not very useful. Now, we will boot UML successfully and see how it looks inside.

BOOTING UML SUCCESSFULLY

The problem was that we didn't tell UML what its root device was. This is an important special case of a more general property of UML—its hardware is configured on the fly. In contrast to a physical system, whose hardware is fixed, a virtual system can be different every time it is booted. So, it expects to be told, either on the command line or later via the `mconsole` interface, what hardware it possesses.

Here, we will configure UML on the command line. The first order of business is to give it a proper root device so that it has something it can boot. As I mentioned earlier, UML devices are virtual and constructed from host resources. Specifically, UML's disks are generally (but not always, as we will see later) files in the host's filesystem.

For example, here is the filesystem we will boot:

```
host% ls -l ~/roots/debian_22
-rw-rw-r--  1 jdike jdike 1074790400 Jan 27 18:31 \
      /home/jdike/roots/debian_22
```

One obvious thing here is that the filesystem image is very large. `file` will tell us a bit more about it:

```
host% file ~/roots/debian_22
/home/jdike/roots/debian_22: Linux rev 1.0 ext2 filesystem data
```

This tells us that the data in this file is an `ext2` filesystem image. In other words, we can loopback-mount it and see that it contains a full filesystem:

```
host# mount ~/roots/debian_22 ~/mnt -o loop
host% ls ~/mnt
bfs  boot   dev  floppy  initrd  lib         mnt   root  tmp  var
bin  cdrom  etc  home    kernel  lost+found  proc  sbin  usr
```

In fact, when mounting this as its root filesystem, UML will do something very similar to a loopback mount. The UML block driver operates by calling read and write on this file on the host, analogous to a block driver on the host doing reads and writes on a physical disk.

The loopback driver on the host is doing exactly the same thing, except from within the host kernel, rather than from a process, where the UML block driver is.

So, in order to provide this file to UML as its root device, we need to tell the UML block driver (the `ubd` or UML Block Device driver) to attach itself to it. This is done with this option:

```
ubda=~/roots/debian_22
```

This is the easiest way to initialize a UML block device, and it simply says that the first UML block device is to be attached to the file `~/roots/debian_22`. Internally, UML tells the kernel initialization code to use the `ubda` device as its default root device (this can be overridden by specifying a different device with the `root=` switch, as the panic message suggested).

I'm going to add one more option to the command line to make the virtual machine's configuration more explicit:

```
mem=128M
```

This makes UML believe it has 128MB of physical memory but does not actually allocate 128MB on the host. Rather, this creates a 128MB sparse file on the host. Being sparse, this file will occupy very little space until data starts being written to it. As the UML instance uses its memory, it will start putting data in the memory backed by this file. As that happens, the host will start allocating memory to hold that data. Since the file is fixed in size, the UML instance is limited to that amount of memory. Its memory consumption will approach this limit asymptotically as it reads file data from its own disks and caches it in its memory.

Since the host will be allocating memory for the UML instance dynamically, as needed, the actual consumption will be less than the maximum for a time. This conserves memory, making it possible to run a greater number of not-too-active UML instances than would be possible otherwise.

The host memory consumption will, in this case, be at most 128MB. Even if the UML instance is fully using its memory, the host memory consumption may be less, as it may have swapped out some of the UML memory. The UML instance, like any other process that has been swapped out, will be unaware of this and will use its memory as though it is present in the host's memory. The host kernel is responsible for swapping data back in as needed in order to maintain this illusion.

The UML instance will also swap if its workload exceeds its physical memory. This is entirely independent from the host swapping the UML instance's memory. Each system will swap when it needs more memory, so if the host is short of memory and the UML instance has plenty, the host will swap and the UML instance won't. Conversely, if the UML instance is short of memory and the host isn't, the UML instance will swap and the host won't. The case where both are swapping at the same time is interesting and can lead to pathological performance problems.[4]

So, the UML command ends up looking like this:

```
~/linux mem=128M ubda=/home/jdike/roots/debian_22
```

Figure 2.3 shows the results.

This is much more interesting than the last attempt. We get to see the filesystem booting. Note that it's almost exactly the same as it would be if the same filesystem were booted on the host. The underlying virtual machine shows through in only a couple of places. One is when the root filesystem is checked[5]:

```
/dev/ubd0: clean, 9591/131328 files, 64611/262144 blocks
```

where we see the UML device name, `/dev/ubd0`, rather than `hda1` or `sda1` as on a physical machine.

4. Consider the case where both the host and the UML instance are swapping at the same time. They may both choose the same page to swap out. If the host swaps it out first, then when the UML instance swaps it, the host will need to read it back from disk so that the UML instance can write it to its own swap device. This will cause the page to be read and written a total of three times, when only once was desirable. This will increase the I/O load on the host at a time when it is already under stress. Solutions for this sort of situation are under investigation and will be described in Chapter 10.

5. The `fsck` message refers to `/dev/ubd0` rather than `/dev/ubda`. Devices can be specified with either numbers or letters. Using letters is generally favored since it is similar to current practice with other drivers, such as naming IDE disks `hda`, `hdb`, and so on. It also makes the use of multiple `ubd` devices within UML less confusing. There's less expectation that `ubdb` on the command line corresponds to minor number 1 inside the UML instance, as the use of `ubd1` does. In fact, `ubdb` has minor number 16 (to allow for partitions on `ubda`). The one case where numbers are needed is when you are plugging a large number of disks into a UML instance. There is no letter equivalent of `ubd512`, so you'd have to use a number to describe this device.

```
~/linux/2.6/2.6.10 22849: ./linux mem=128M ubda=/home/jdike/roots/debian_22
Checking for /proc/mm...not found
Checking for the skas3 patch in the host...not found
Checking PROT_EXEC mmap in /tmp...OK
Linux version 2.6.11-rc1-mm1 (jdike@tp.user-mode-linux.org) (gcc version 3.3.2
     20031022 (Red Hat Linux 3.3.2-1)) #83 Thu Jan 27 12:16:00 EST 2005
Built 1 zonelists
Kernel command line: mem=128M ubda=/home/jdike/roots/debian_22 root=98:0
PID hash table entries: 1024 (order: 10, 16384 bytes)
Dentry cache hash table entries: 32768 (order: 5, 131072 bytes)
Inode-cache hash table entries: 16384 (order: 4, 65536 bytes)
Memory: 126720k available
Mount-cache hash table entries: 512 (order: 0, 4096 bytes)
Checking for host processor cmov support...Yes
Checking for host processor xmm support...No
Checking that ptrace can change system call numbers...OK
Checking syscall emulation patch for ptrace...missing
Checking that host ptys support output SIGIO...Yes
Checking that host ptys support SIGIO on close...No, enabling workaround
Checking for /dev/anon on the host...Not available (open failed with errno 2)
NET: Registered protocol family 16
mconsole (version 2) initialized on /home/jdike/.uml/igpn9r/mconsole
VFS: Disk quotas dquot_6.5.1
Dquot-cache hash table entries: 1024 (order 0, 4096 bytes)
io scheduler noop registered
io scheduler anticipatory registered
io scheduler deadline registered
io scheduler cfq registered
NET: Registered protocol family 2
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP established hash table entries: 8192 (order: 4, 65536 bytes)
TCP bind hash table entries: 8192 (order: 3, 32768 bytes)
TCP: Hash tables configured (established 8192 bind 8192)
NET: Registered protocol family 1
NET: Registered protocol family 17
Initialized stdio console driver
Console initialized on /dev/tty0
Initializing software serial port version 1
 ubda: unknown partition table
VFS: Mounted root (ext2 filesystem) readonly.
line_ioctl: tty0: ioctl KDSIGACCEPT called
INIT: version 2.78 booting
Activating swap...
Checking root file system...
Parallelizing fsck version 1.18 (11-Nov-1999)
/dev/ubd0: clean, 9591/131328 files, 64611/262144 blocks
Calculating module dependencies... depmod: get_kernel_syms: Function not
     implemented
done.
Loading modules: cat: /etc/modules: No such file or directory
```

(*continues*)

Figure 2.3 Output from the first successful boot of UML

```
modprobe: Can't open dependencies file /lib/modules/2.6.11-rc1-mm1/modules.dep
     (No such file or directory)
Checking all file systems...
Parallelizing fsck version 1.18 (11-Nov-1999)
Setting kernel variables.
Mounting local filesystems...
mount: devpts already mounted on /dev/pts
none on /tmp type tmpfs (rw)
Setting up IP spoofing protection: rp_filter.
Configuring network interfaces: done.

Setting the System Clock using the Hardware Clock as reference...
line_ioctl: tty1: unknown ioctl: 0x4b50
hwclock is unable to get I/O port access:  the iopl(3) call failed.
System Clock set. Local time: Thu Jan 27 18:51:28 EST 2005

Cleaning: /tmp /var/lock /var/run.
Initializing random number generator... done.
Recovering nvi editor sessions... done.
INIT: Entering runlevel: 2
Starting system log daemon: syslogd  syslogd: /dev/xconsole: No such file or
     directory
 klogd.
Starting portmap daemon: portmap.
Starting NFS common utilities: statd lockdlockdsvc: Function not implemented
.
Starting internet superserver: inetd.
Starting MySQL database server: mysqld.
Not starting NFS kernel daemon: No exports.
Starting OpenBSD Secure Shell server: sshd.
Starting web server: apache.
/usr/sbin/apachectl start: httpd started

Debian GNU/Linux 2.2 usermode tty0

usermode login:
```

Figure 2.3 Output from the first successful boot of UML (*continued*)

The other is when the boot scripts try to synchronize the internal kernel clock with the system's hardware clock:

```
Setting the System Clock using the Hardware Clock as reference...
line_ioctl: tty1: unknown ioctl: 0x4b50
hwclock is unable to get I/O port access:  the iopl(3) call \
      failed.
```

The UML serial line driver is complaining about an `ioctl` it doesn't implement, and the `hwclock` program inside UML is complain-

ing that it tried to execute the `iopl` instruction and failed. These are both symptoms of `hwclock` trying different methods of accessing the hardware system clock and failing because the device doesn't exist in UML. The UML kernel does have access to a clock, but it is not one that `hwclock` will recognize. Rather, it is simply a call to the host's `gettimeofday`.

After that, you'll notice that a relatively small number of services are started, but they do include such things as NFS, MySQL, and Apache. All of these run just as they would on a physical machine. This boot process took about 5 seconds on my laptop, demonstrating one of the conveniences of UML—the ability to quickly create and destroy virtual machines.

LOOKING AT A UML FROM THE INSIDE AND OUTSIDE

Finally, we'll see a login prompt. Actually, I see three on my screen. One is in the xterm window in which I ran UML. The other two are in xterm windows run by UML in order to hold the second console and the first serial line, which are configured to have gettys running on them. We'll log in as root (using the highly secure default root password of `root` that most of my UML filesystems have) and get a shell:

```
usermode login: root
Password:
Last login: Thu Jan 27 18:51:35 2005 on tty0
Linux usermode 2.6.11-rc1-mm1 #83 Thu Jan 27 12:16:00 EST 2005 \
      i686 unknown
usermode:~#
```

Again, this is identical to what you'd see if you logged in to a physical machine booted on this filesystem.

Now it's time to start poking around inside this UML and see what it looks like. First, we'll look at what processes are running, as shown in Figure 2.4.

There's not much to comment on except the total normality of this output. What's interesting here is to look at the host. Figure 2.5 shows the corresponding processes on the host.

Each of the nameless host processes corresponds to an address space inside this UML instance. Except for application and kernel threads, there's a one-to-one correspondence between UML processes and these host processes.

```
usermode:~# ps uax
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.3   1100   464 ?        S    19:17   0:00 init [2]
root         2  0.0  0.0      0     0 ?        RWN  19:17   0:00 [ksoftirqd/0]
root         3  0.0  0.0      0     0 ?        SW<  19:17   0:00 [events/0]
root         4  0.0  0.0      0     0 ?        SW<  19:17   0:00 [khelper]
root         5  0.0  0.0      0     0 ?        SW<  19:17   0:00 [kthread]
root         6  0.0  0.0      0     0 ?        SW<  19:17   0:00 [kblockd/0]
root         7  0.0  0.0      0     0 ?        SW   19:17   0:00 [pdflush]
root         8  0.0  0.0      0     0 ?        SW   19:17   0:00 [pdflush]
root        10  0.0  0.0      0     0 ?        SW<  19:17   0:00 [aio/0]
root         9  0.0  0.0      0     0 ?        SW   19:17   0:00 [kswapd0]
root        96  0.0  0.4   1420   624 ?        S    19:17   0:00 /sbin/syslogd
root        98  0.0  0.3   1084   408 ?        S    19:17   0:00 /sbin/klogd
daemon     102  0.0  0.3   1200   420 ?        S    19:17   0:00 /sbin/portmap
root       105  0.0  0.4   1128   548 ?        S    19:17   0:00 /sbin/rpc.statd
root       111  0.0  0.4   1376   540 ?        S    19:17   0:00 /usr/sbin/inetd
root       120  0.0  0.6   1820   828 ?        S    19:17   0:00 /bin/sh /usr/bin/
mysql      133  0.1  1.2  19244  1540 ?        S    19:17   0:00 /usr/sbin/mysqld
mysql      135  0.0  1.2  19244  1540 ?        S    19:17   0:00 /usr/sbin/mysqld
mysql      136  0.0  1.2  19244  1540 ?        S    19:17   0:00 /usr/sbin/mysqld
root       144  0.9  0.9   2616  1224 ?        S    19:17   0:00 /usr/sbin/sshd
root       149  0.0  1.0   2588  1288 ?        S    19:17   0:00 /usr/sbin/apache
root       152  0.0  0.9   2084  1220 tty0     S    19:17   0:00 -bash
root       153  0.0  0.3   1084   444 tty1     S    19:17   0:00 /sbin/getty 38400
root       154  0.0  0.3   1084   444 tty2     S    19:17   0:00 /sbin/getty 38400
root       155  0.0  0.3   1084   444 ttyS0    S    19:17   0:00 /sbin/getty 38400
www-data   156  0.0  1.0   2600  1284 ?        S    19:17   0:00 /usr/sbin/apache
www-data   157  0.0  1.0   2600  1284 ?        S    19:17   0:00 /usr/sbin/apache
www-data   158  0.0  1.0   2600  1284 ?        S    19:17   0:00 /usr/sbin/apache
www-data   159  0.0  1.0   2600  1284 ?        S    19:17   0:00 /usr/sbin/apache
www-data   160  0.0  1.0   2600  1284 ?        S    19:17   0:00 /usr/sbin/apache
root       162  2.0  0.5   2384   736 tty0     R    19:17   0:00 ps uax
usermode:~#
```

Figure 2.4 Output from `ps uax` inside UML

Notice that the properties of the UML processes and the corresponding host processes don't have much in common. All of the host processes are owned by me, whereas the UML processes have various owners, including root. The process IDs are totally different, as are the virtual and resident memory sizes.

This is because the host processes are simply containers for UML address spaces. All of the properties visible inside UML are maintained by UML totally separate from the host. For example, the owner of the host processes will be whoever ran UML. However, many UML processes will be owned by root. These processes have root privileges

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
jdike     9938  0.1  3.1 131112 16264 pts/3  R    19:17   0:03 ./linux [ps]
jdike     9942  0.0  3.1 131112 16264 pts/3  S    19:17   0:00 ./linux [ps]
jdike     9943  0.0  3.1 131112 16264 pts/3  S    19:17   0:00 ./linux [ps]
jdike     9944  0.0  0.0    472   132 pts/3    T    19:17   0:00
jdike    10036  0.0  0.5   8640  2960 pts/3    S    19:17   0:00 xterm -T Virtual
jdike    10038  0.0  0.0   1368   232 ?        S    19:17   0:00 /usr/lib/uml/port
jdike    10039  0.0  1.5 131092  8076 pts/6    S    19:17   0:00 ./linux [hwclock]
jdike    10095  0.0  0.1    632   604 pts/3    T    19:17   0:00
jdike    10099  0.0  0.0    416   352 pts/3    T    19:17   0:00
jdike    10107  0.0  0.0    428   332 pts/3    T    19:17   0:00
jdike    10113  0.0  0.1    556   516 pts/3    T    19:17   0:00
jdike    10126  0.0  0.0    548   508 pts/3    T    19:17   0:00
jdike    10143  0.0  0.0    840   160 pts/3    T    19:17   0:00
jdike    10173  0.0  0.2   1548  1140 pts/3    T    19:17   0:00
jdike    10188  0.0  0.1   1232   780 pts/3    T    19:17   0:00
jdike    10197  0.0  0.1   1296   712 pts/3    T    19:17   0:00
jdike    10205  0.0  0.0    452   452 pts/3    T    19:17   0:00
jdike    10207  0.0  0.0    452   452 pts/3    T    19:17   0:00
jdike    10209  0.0  0.0    452   452 pts/3    T    19:17   0:00
jdike    10210  0.0  0.5   8640  2960 pts/3    S    19:17   0:00 xterm -T Virtual
jdike    10212  0.0  0.0   1368   232 ?        S    19:17   0:00 /usr/lib/uml/port
jdike    10213  0.0  2.9 131092 15092 pts/7    S    19:17   0:00 ./linux [/sbin/ge
jdike    10214  0.0  0.1   1292   688 pts/3    T    19:17   0:00
jdike    10215  0.0  0.1   1292   676 pts/3    T    19:17   0:00
jdike    10216  0.0  0.1   1292   676 pts/3    T    19:17   0:00
jdike    10217  0.0  0.1   1292   676 pts/3    T    19:17   0:00
jdike    10218  0.0  0.1   1292   676 pts/3    T    19:17   0:00
jdike    10220  0.0  0.1   1228   552 pts/3    T    19:17   0:00
```

Figure 2.5 Partial output from `ps uax` on the host

inside UML, but they have no special privileges on the host. This important fact means that root can do anything inside UML without being able to do anything on the host. A user logged in to a UML as root has no special abilities on the host and, in fact, may not have any abilities at all on the host.

Now, let's look at the memory usage information in `/proc/meminfo`, shown in Figure 2.6.

The total amount of memory shown, 126796K, is close to the 128MB we specified on the command line. It's not exactly 128MB because some memory allocated during early boot isn't counted in the total. Going back to the host `ps` output in Figure 2.5, notice that the `linux` processes have a virtual size (the VSZ column) of almost exactly 128MB. The difference of 50K is due to a small amount of memory in the UML binary, which isn't counted as part of its physical memory.

```
usermode:~# cat /proc/meminfo
MemTotal:       126796 kB
MemFree:        112952 kB
Buffers:           512 kB
Cached:           7388 kB
SwapCached:          0 kB
Active:           6596 kB
Inactive:         3844 kB
HighTotal:           0 kB
HighFree:            0 kB
LowTotal:       126796 kB
LowFree:        112952 kB
SwapTotal:           0 kB
SwapFree:            0 kB
Dirty:               0 kB
Writeback:           0 kB
Mapped:           5424 kB
Slab:             2660 kB
CommitLimit:     63396 kB
Committed_AS:    23100 kB
PageTables:        248 kB
VmallocTotal:   383984 kB
VmallocUsed:        24 kB
VmallocChunk:   383960 kB
```

Figure 2.6 The UML `/proc/meminfo`

Now, let's go back to the host `ps` output and pick one of the UML processes:

```
jdike     9938  0.1  3.1 131112 16264 pts/3  R    19:17   0:03 \
      ./linux [ps]
```

We can look at its open files by looking at the `/proc/9938/fd` directory, which shows an entry like this:

```
lrwx------  1 jdike jdike 64 Jan 28 12:48 3 -> \
      /tmp/vm_file-AwBs1z (deleted)
```

This is the host file that holds, and is the same size (128MB in our case) as, the UML "physical" memory. It is created in `/tmp` and then deleted. The deletion prevents something else on the host from opening it and corrupting it. However, this has the somewhat undesirable side effect that `/tmp` can become filled with invisible files, which can confuse people who don't know about this aspect of UML's behavior.

To make matters worse, it is recommended for performance reasons to use `tmpfs` on `/tmp`. UML performs noticeably better when its memory

file is on `tmpfs` rather than on a disk-based filesystem such as `ext3`. However, a `tmpfs` mount is smaller than the disk-based filesystem `/tmp` would normally be on and thus more likely to run out of space when running multiple UML instances. This can be handled by making the `tmpfs` mount large enough to hold the maximum physical memories of all the UML instances on the host or by creating a `tmpfs` mount for each UML instance that is large enough to hold its physical memory.

Take a look at the root directory:

```
UML# ls /
bfs  boot   dev  floppy  initrd  lib         mnt   root  tmp  var
bin  cdrom  etc  home    kernel  lost+found  proc  sbin  usr
```

This looks strikingly similar to the listing of the loopback mount earlier and somewhat different from the host. Here UML has done the equivalent of a loopback mount of the `~/roots/debian_22` file on the host.

Note that making the loopback mount on the host required root privileges, while I ran UML as my normal, nonroot self and accomplished the same thing. You might think this demonstrates that either the requirement of root privileges on the host is unnecessary or that UML is some sort of security hole for not requiring root privileges to do the same thing. Actually, neither is true because the two operations, the loopback mount on the host and UML mounting its root filesystem, aren't quite the same thing. The loopback mount added a mount point to the host's filesystem, while the mount of `/` within UML doesn't. The UML mount is completely separate from the host's filesystem, so the ability to do this has no security implications.

However, from a different point of view, some security implications do arise. There is no access from the UML filesystem to the host filesystem. The root user inside the UML can do anything on the UML filesystem, and thus, to the host file that contains it, but can't do anything outside it. So, inside UML, even root is jailed and can't break out.[6]

6. We will talk about this in greater detail in Chapter 10, but UML is secure against a breakout by the superuser only if it is configured properly. Most important, module support and the ability to write to physical memory must be disabled within the UML instance. The UML instance is owned by some user on the host, and the UML kernel has the same privileges as that user. So, the ability for root to modify kernel memory and inject code into it would allow doing anything on the host that the host user can do. Disallowing this ensures that even the superuser inside UML stays jailed.

This is a general property of UML—a UML is a full-blown Linux machine with its own resources. With respect to those resources, the root user within UML can do anything. But it can do nothing at all to anything on the host that's not explicitly provided to the UML. We've just seen this with disk space and files, and it's also true for networking, memory, and every other type of host resource that can be made accessible within UML.

Next, we can see some of UML's hardware support by looking at the mount table:

```
UML# mount
/dev/ubd0 on / type ext2 (rw)
proc on /proc type proc (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
none on /tmp type tmpfs (rw)
```

Here we see the `ubd` device we configured on the command line now mounted as the root filesystem. The other mounts are normal virtual filesystems, `procfs` and `devpts`, and a `tmpfs` mount on `/tmp`.

`df` will show us how much space is available on the virtual disk:

```
UML# df
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/ubd0              1032056    242108    737468  25% /
none                     63396         0     63396   0% /tmp
```

Compare the total size of `/dev/ubd0` (1032056K) to that of the host file:

```
-rw-rw-r--  1 jdike jdike 1074790400 Jan 27 18:31 \
      /home/jdike/roots/debian_22
```

They are nearly the same,[7] with the difference probably being the `ext2` filesystem overhead. The entire UML filesystem exists in and is confined to that host file. This is another way in which users inside the UML are confined or jailed. A UML user has no way to consume more disk space than is in that host file.

However, on the host, it is possible to extend the filesystem file, and the extra space becomes available to UML. In Chapter 6 we will see exactly how this is done, but for now, it's just important to note that this is a good example of how much more flexible virtual hardware is in

7. The difference between the 1074790400 byte host file and 1032056K (1056825344 bytes) is 1.7%.

comparison to physical hardware. Try adding extra space to a physical disk or a physical disk partition. You can repartition the disk in order to extend a partition, but that's a nontrivial, angst-ridden operation that potentially puts all of the data on the disk at risk if you make a mistake. You can also add a new volume to the volume group you wish to increase, but this requires that the volume group be set up beforehand and that you have a spare partition to add to it. In comparison, extending a file using `dd` is a trivial operation that can be done as a normal user, doesn't put any data at risk except that in the file, and doesn't require any prior setup.

We can poke around `/proc` some more to compare and contrast this virtual machine with the physical host it's running on. For some similarities, let's look at `/proc/filesystems`:

```
UML# more /proc/filesystems
nodev   sysfs
nodev   rootfs
nodev   bdev
nodev   proc
nodev   sockfs
nodev   pipefs
nodev   futexfs
nodev   tmpfs
nodev   eventpollfs
nodev   devpts
        reiserfs
        ext3
        ext2
nodev   ramfs
nodev   mqueue
```

There's no sign of any UML oddities here at all. The reason is that the filesystems are not hardware dependent. Anything that doesn't depend on hardware will be exactly the same in UML as on the host. This includes things such as virtual devices (e.g., pseudo-terminals, loop devices, and TUN/TAP[8] network interfaces) and network protocols, as well as the filesystems.

So, in order to see something different from the host, we have to look at hardware-specific stuff. For example, `/proc/interrupts` contains information about all interrupt sources on the system. On the

8. The TUN/TAP driver is a virtual network interface that allows packets to be handled by a process, in order to create a tunnel (the origin of "TUN") or a virtual Ethernet device ("TAP").

host, it contains information about devices such as the timer, keyboard, and disks. In UML, it looks like this:

```
UML# more /proc/interrupts
           CPU0
  0:     211586       SIGVTALRM  timer
  2:         87           SIGIO  console, console, console
  3:          0           SIGIO  console-write, console-write, \
                          console-write
  4:       2061           SIGIO  ubd
  6:          0           SIGIO  ssl
  7:          0           SIGIO  ssl-write
  9:          0           SIGIO  mconsole
 10:          0           SIGIO  winch, winch, winch
 11:         56           SIGIO  write sigio
```

The timer, keyboard, and disks are here (entries 0, 2 and 6, and 4, respectively), as are a bunch of mysterious-looking entries. The `-write` entries stem from a weakness in the host Linux `SIGIO` support. `SIGIO` is a signal generated when input is available, or output is possible, on a file descriptor. A process wishing to do interrupt-driven I/O would set up `SIGIO` support on the file descriptors it's using. An interrupt when input is available on a file descriptor is obviously useful. However, an interrupt when output is possible is also sometimes needed.

If a process is writing to a descriptor, such as one belonging to a pipe or a network socket, faster than the process on the other side is reading it, then the kernel will buffer the extra data. However, only a limited amount of buffering is available. When that limit is reached, further writes will fail, returning `EAGAIN`. It is necessary to know when some of the data has been read by the other side and writes may be attempted again. Here, a `SIGIO` signal would be very handy. The trouble is that support of `SIGIO` when output is possible is not universal. Some IPC mechanisms support `SIGIO` when input is available, but not when output is possible.

In these cases, UML emulates this support with a separate thread that calls `poll` to wait for output to become possible on these descriptors, interrupting the UML kernel when this happens. The interrupt this generates is represented by one of the `-write` interrupts.

The other mysterious entry is the `winch` interrupt. This appears because UML wants to detect when one of its consoles changes size, as when you resize the xterm in which you ran UML. Obviously this is not a concern for the host, but it is for a virtual machine. Because of the interface for registering for `SIGWINCH` on a host device, a separate thread is created to receive `SIGWINCH`, and it interrupts UML itself

whenever one comes in. Thus, SIGWINCH looks like a separate device from the point of view of /proc/interrupts.

/proc/cpuinfo is interesting:

```
UML# more /proc/cpuinfo
processor       : 0
vendor_id       : User Mode Linux
model name      : UML
mode            : skas
host            : Linux tp.user-mode-linux.org 2.4.27 #6 \
      Thu Jan 13 17:06:15 EST 2005 i686
bogomips        : 1592.52
```

Much of the information in the host's /proc/cpuinfo makes no sense in UML. It contains information about the physical CPU, which UML doesn't have. So, I just put in some information about the host, plus some about the UML itself.

CONCLUSION

At this point, we've seen a UML from both the inside and the outside. We've seen how a UML can use host resources for its hardware and how it's confined to whatever has been provided to it.

A UML is both very similar to and very different from a physical machine. It is similar as long as you don't look at its hardware. When you do, it becomes clear that you are looking at a virtual machine with virtual hardware. However, as long as you stay away from the hardware, it is very hard to tell that you are inside a virtual machine.

Both the similarities and the differences have advantages. Obviously, having a UML run applications in exactly the same way as on the host is critical for it to be useful. In this chapter we glimpsed some of the advantages of virtual hardware. Soon we will see that virtualized hardware can be plugged, unplugged, extended, and managed in ways that physical hardware can't. The next chapter begins to show you what this means.

CHAPTER 3

Exploring UML

LOGGING IN AS A NORMAL USER

In this chapter we will explore a UML instance in more detail, looking at how it is similar to and how it differs from a physical Linux machine. While doing a set of fairly simple, standard system administration chores in the instance, we will see some UML twists to them. For example, we will add swap space and mount filesystems. The twist is that we will do these things by plugging the required devices into the UML at runtime, from the host, without rebooting the UML.

First, let's log in to the UML instance, as we did in the previous chapter. When the UML boots, we see a login prompt in the window in which we started it. Some xterm windows pop up on the screen, which we ignore. They also contain login prompts. We could log in as root, but let's log in as a normal user, username `user`, with the very secure password `user`:

```
Debian GNU/Linux 2.2 usermode tty1

usermode login: user
Password:
Last login: Sun Dec 22 21:50:44 2002 from uml on pts/0
```

```
Linux usermode 2.6.11-rc3-mm1 #2 Tue Feb 8 15:41:40 EST 2005 \
i686 unknown
UML%  pwd
/home/user
```

This is basically the same as a physical system. In this window, we are a normal, unprivileged user, in a normal home directory. We can test our lack of privileges by trying to do something nasty:

```
UML% rm -f /bin/ls
rm: cannot unlink `/bin/ls': Permission denied
```

CONSOLES AND SERIAL LINES

In addition to the xterm consoles that made themselves visible, some others have attached themselves less visibly to other host resources. You can attach UML consoles to almost any host device that can be used for that purpose. For example, they can be (and some, by default, are) attached to host pseudo-terminals. They announce themselves in the kernel log, which we can see by running `dmesg`:

```
UML% dmesg | grep "Serial line"
Serial line 0 assigned device '/dev/pts/13'
```

This tells us that one UML serial line has been configured in `/etc/inittab` to have a login prompt on it. The serial line has been configured at the "hardware" level to be attached to a host pseudo-terminal, and it has allocated the host's `/dev/pts/13`.

Now we can run a terminal program, such as `screen` or `minicom`, on the host, attach it to `/dev/pts/13`, and log in to UML on its one serial line. After running

```
host% screen /dev/pts/13
```

we see a blank screen session. Hitting return gives us another UML login prompt, as advertised:

```
Debian GNU/Linux 2.2 usermode ttyS0

usermode login:
```

Notice the `ttyS0` in the banner, in comparison to the `tty0` we saw while logging in as root in the previous chapter and the `tty1` we just saw while logging in as `user`. The `tty0` and `tty1` devices are UML

consoles, while `ttyS0` is the first serial line. On a physical machine, the consoles are devices that are displayed on the screen, and the serial lines are ports coming out of the back of the box. There's a clear difference between them.

In contrast, there is almost no difference between the consoles and serial lines in UML. They plug themselves into the console and serial line infrastructures, respectively, in the UML kernel. This is the cause of the different device names. However, in all other ways, they are identical in UML. They share essentially all their code, they can be configured to attach to exactly the same host devices, and they behave in the same ways.

In fact, the serial line driver in UML owes its existence to a historical quirk. Because of a limitation in the first implementation of UML, it was impossible to log in on a console in the window in which you ran it. To allow logging in to UML at all, I implemented the serial line driver to connect itself to a host device, and you would attach to this using something like `screen`.

As time went on and limitations disappeared, I implemented a real console driver. After a while, it dawned on me that there was no real difference between it and the serial line driver, so I started merging the two drivers, making them share more and more code. Now almost the only differences between them are that they plug themselves into different parts of the kernel.

UML consoles and serial lines can be attached to the same devices on the host, and we've seen a console attached to `stdin` and `stdout` of the `linux` process, consoles appearing in xterms, and a serial line attached to a host pseudo-terminal. They can also be attached to host ports, allowing you to telnet to the specified port on the host and log in to the UML from there. This is a convenient way to make a UML accessible from the network without enabling the network within UML.

Finally, UML consoles and serial lines can be attached to host terminals, which can be host consoles, such as `/dev/tty*`, or the slave side of pseudo-terminals. Attaching a UML console to a host virtual console has the interesting effect of putting the UML login prompt on the host console, making it appear (to someone not paying sufficient attention) to be the host login.

Let's look at some examples. First, let's attach a console to a host port. We need to find an unused console to work with, so let's use the UML management console tool to query the UML configuration:

```
host% uml_mconsole debian config con0
OK fd:0,fd:1
```

```
host% uml_mconsole debian config con1
OK none
host% uml_mconsole debian config con2
OK pts:/dev/pts/10
host% uml_mconsole debian config con3
OK pts
```

We will cover the full capabilities of `uml_mconsole` in Chapter 8, but this gives us an initial look at it. The first argument, `debian`, specifies which UML we wish to talk to. A UML can be named and given a unique machine ID, or `umid`. When I ran this UML, I added `umid=debian` to the command line, giving this instance the name `debian`. `uml_mconsole` knows how to use this name to communicate with the `debian` UML.

If you didn't specify the `umid` on the command line, UML gives itself a random `umid`. There are a couple of ways to tell what it chose. First, look through the boot output or output from `dmesg` for a line that looks like this:

```
mconsole (version 2) initialized on /home/jdike/.uml/3m3vDd/mconsole
```

In this case, the `umid` is `3m3vDd`. You can communicate with this instance by using that `umid` on the `uml_mconsole` command line.

Second, UML puts a directory with the same name as the `umid` in a special parent directory, by default, `~/.uml`. So, you could also look at the subdirectory[1] of your `~/.uml` directory for the `umid` to use.

The rest of the `uml_mconsole` command line is the command to send to the specified UML. In this case, we are asking for the configurations of the first few consoles. Console names start with `con`; serial line names begin with `ssl`.

I will describe as much of the output format as needed here; Figure 3.1 contains a more complete and careful description.

Looking at the output about the UML configuration, we see an `OK` on each response, which means that the command succeeded in communicating with the UML and getting a response. The `con0` response says that console 0 is attached to `stdin` and `stdout`. This bears some explaining, so let's pull apart that response. There are two pieces to it, `fd:0` and `fd:1`, separated by a comma. In a comma-separated configuration like this, the first part refers to input to the console (or serial line), and the second part refers to output from it.

1. At this point, there should be only one.

A UML console or serial line configuration can consist of separate input and output configurations, or a single configuration for both. If both are present, they are separated by a colon. For example, `fd:0,fd:1` specifies that console input comes from UML's file descriptor 0 and that output goes to file descriptor 1. In contrast, `fd:3` specifies that both input and output are attached to file descriptor 3, which should have been set up on the UML command line with something like `3<>filename`.

A single device configuration consists of a device type (`fd` in the examples above) and device-specific information separated by a colon. The possible device types and additional information are as follows.

- `fd`—A host file descriptor belonging to the UML process; specify the file descriptor number after the colon.
- `pty`—A BSD pseudo-terminal; specify the `/dev/ptyxx` name of the pseudo-terminal you wish to attach the console to. To access it, you will attach a terminal program, such as `screen` or `minicom`, to the corresponding `/dev/ttyxx` file.
- `pts`—A `devpts` pseudo-terminal; there is no `pts`-specific data you need to add. In order to connect to it, you will need to find which `pts` device it allocated by reading the UML kernel log through `dmesg` or by using `uml_mconsole` to query the configuration.
- `port`—A host port; specify the port number. You access the port by telnetting to it. If you're on the host, you will telnet to localhost:

  ```
  host% telnet localhost port-number
  ```

  You can also telnet to that port from another machine on the network:

  ```
  host% telnet uml-host port-number
  ```

- `xterm`—No extra information needed. This will display an xterm on your screen with the console in it. UML needs a valid `DISPLAY` environment variable and `xterm` installed on the host, so this won't work on headless servers. This is the default for consoles other than console 0, so for headless servers, you will need to change this.
- `null`—No extra information needed. This makes the console available inside UML, but output is ignored and there is never any input. This would be very similar to attaching the console to the host's `/dev/null`.
- `none`—No extra information needed. This removes the device from UML, so that attempts to access it will fail with "No such device."

When requesting configuration information through `uml_mconsole` for `pts` consoles, it will report the actual device that it allocated after the colon, as follows:

```
host% uml_mconsole debian config con2
OK pts:/dev/pts/10
```

The syntax for specifying console and serial line configurations is the same on the UML and `uml_mconsole` command lines, except that the UML command line allows giving all devices the same configuration. A specific console or serial line is specified as either `con` *n* or `ssl` *n*.

(*continues*)

Figure 3.1 Detailed description of UML console and serial line configuration

On the UML command line, all consoles or serial lines may be given the same configuration with just `con=` *`configuration`* or `ssl=` *`configuration`*.

Any specific device configurations that overlap this will override it. So

```
con=pts con0=fd:0,fd:1
```

attaches all consoles to `pts` devices, except for the first one, which is attached to `stdin` and `stdout`.

Console input and output can be specified separately. They are completely independent—the host device types don't even need to match. For example,

```
ssl2=pts,xterm
```

will attach the second serial line's input to a host `pts` device and the output to an xterm. The effect of this is that when you attach `screen` or another terminal program to the host `pts` device, that's the input to the serial line. No output will appear in `screen`—that will all be directed to the xterm. Most input will also appear in the xterm because that is echoed in the shell.

This can have unexpected effects. Repeating a configuration for both the input and output will, in some cases, attach them to distinct host devices of the same type. For example,

```
con2=xterm,xterm
```

will create two xterms—one will accept console input, and the other will display the console's output. The same is true for `pts`.

Figure 3.1 Detailed description of UML console and serial line configuration (*continued*)

The `fd:0` part also has two pieces, `fd` and `0`, separated by a colon. `fd` says that the console input is to be attached to a file descriptor of the `linux` process, and `0` says that file descriptor will be `stdin`, file descriptor zero. Similarly, the output is specified to be file descriptor one, `stdout`.

When the console input and output go to the same device, as we can see with `con2` being attached to `pts:/dev/pts/10`, input and output are not specified separately. There is only a single colon-separated device description. As you might have guessed, `pts` refers to a `devpts` pseudo-terminal, and `/dev/pts/10` tells you specifically which pseudo-terminal the console is attached to.

The `con1` configuration is one we haven't seen before. It simply says that the console doesn't exist—there is no such device.

The configuration for `con3` is the one we are looking for. `pts` says that this is a `pts` console, and there's no specific `pts` device listed, so it has not yet been activated by having a UML getty running on it. We will reconfigure this one to be attached to a host port:

```
host% uml_mconsole debian config con3=port:9000
OK
```

`port:9000` says that the console should be attached to the host's port 9000, which we will access by telnetting to that port.

We can double-check that the change actually happened:

```
host% uml_mconsole debian config con3
OK port:9000
```

So far, so good. Let's try telnetting there now:

```
host% telnet localhost 9000
Trying 127.0.0.1...
telnet: connect to address 127.0.0.1: Connection refused
```

This failed because UML hasn't run a getty on its console 3. We can fix this by editing its `/etc/inittab`. Looking there on my machine, I see:

```
#3:2345:respawn:/sbin/getty 38400 tty3
```

I had enabled this one in the past but since disabled it. You may not have a `tty3` entry at all. You want to end up with a line that looks like this:

```
3:2345:respawn:/sbin/getty 38400 tty3
```

I'll just uncomment mine; you may have to add the line in its entirety, so fire up your favorite editor on `/etc/inittab` and fix it. Now, tell `init` it needs to reread the `inittab` file:

```
UML# kill -HUP 1
```

Let's go back to the host and try the telnet again:

```
host% telnet localhost 9000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Fedora Core release 1 (Yarrow)
Kernel 2.4.27 on an i686

Debian GNU/Linux 2.2 usermode tty3

usermode login:
```

Here we have the UML's console, as advertised. Notice the discrepancy between the telnet banner and the login banner. Telnet is telling us that we are attaching to a Fedora Core 1 (FC1) system running a 2.4.27 kernel, while login is saying that we are attaching to a Debian

system. This is because the host is the FC1 system, and `telnetd` running on the host and attaching us to the host's port 9000 is telling us about the host. There is some abuse of `telnetd`'s capabilities going on in order to allow the redirection of traffic between the host port and UML, and this is responsible for the confusion.

Now, let's stick a UML console on a host console. First, we need to make sure there's no host getty or login running on the chosen console. Looking at my host's `/etc/inittab`, I see:

```
6:2345:respawn:/sbin/mingetty tty6
```

for the last console, and hitting Ctrl-Alt-F6 to switch to that virtual console confirms that a getty is running on it. I'll comment it out, so it looks like this:

```
#6:2345:respawn:/sbin/mingetty tty6
```

I tell `init` to reread `inittab`:

```
host# kill -HUP 1
```

and switch back to that console to make sure it is not being used by the host any more. I now need to make sure that UML can open it:

```
host% ls -l /dev/tty6
crw-------  1 root root 4, 6 Feb 17 16:26 /dev/tty6
```

This not being the case, I'll change the permissions so that UML has both read and write access to it:

```
host# chmod 666 /dev/tty6
```

After you make any similar changes needed on your own machine, we can tell UML to take over the console. We used the UML `tty3` for the host port console, so let's look at `tty4`:

```
host% uml_mconsole debian config con4
OK pts
```

So, let's assign `con4` to the host's `/dev/tty6` in the usual way:

```
host% uml_mconsole debian config con4=tty:/dev/tty6
OK
```

After enabling `tty4` in the UML `/etc/inittab` and telling `init` to reread the file, we should be able to switch to the host's virtual console

6 and see the UML login prompt. Taken to extremes, this can be somewhat mind bending. Applying this technique to the other virtual consoles results in them all displaying UML, not host, login prompts.

For the security conscious, this sort of redirection and fakery can be valuable. It allows potential attacks on the host to be redirected to a jail, where they can be contained, logged, and analyzed. For the rest of us, it serves as an example of the flexibility of the UML consoles.

Now that we've seen all the ways to access our UML console, it's time to stay logged in on the console and see what we can do inside the UML.

ADDING SWAP SPACE

UML is currently running everything in the memory that it has been assigned since it has no swap space. Normal Linux machines have some swap, so let's fix that now.

We need some sort of disk to swap onto, and since UML disks are generally host files, we need to make a file on the host to be the swap device:

```
host% dd if=/dev/zero of=swap bs=1024 seek=$[ 1024 * 1024 ] count=1
1+0 records in
1+0 records out
host% ls -l swap
-rw-rw-rw-  1 jdike jdike 1073742848 Feb 18 12:31 swap
```

This technique uses `dd` to create a 1GB sparse file on the host by seeking 1 million 1K blocks and then writing a 1K block of zeros there. The use of sparse files is pretty standard with UML since it allows host disk space to be allocated only when it is needed. So, this swap device file consumes only 1K of disk space, even though it is technically 1GB in length.

We can see the true size, that is, the actual disk space consumption, of the file by adding `-s` to the `ls` command line:

```
host% ls -ls swap
12 -rw-rw-r--  1 jdike jdike 1073742848 Oct 27 17:27 swap
```

The `12` in the first column is the number of disk blocks actually occupied by the file. A disk block is 512 bytes, so this file that looks like it's 1GB in length is taking only 6K of disk space.

Now, we need to plug this new file into the UML as an additional block device, which we will do with the management console:

```
host% uml_mconsole debian config ubdb=swap
OK
```

We can check this by asking for the configuration of `ubdb` in the same way we asked about consoles earlier:

```
host% uml_mconsole debian config ubdb
OK /home/jdike/swap
```

Now, back in the UML, we have a brand-new second block device, so let's set it up for swapping, then swap on it, and look at `/proc/meminfo` to check our work:

```
UML# mkswap /dev/ubdb
Setting up swapspace version 1, size = 1073737728 bytes
UML# swapon /dev/ubdb
UML# grep Swap /proc/meminfo
SwapCached:          0 kB
SwapTotal:     1048568 kB
SwapFree:      1048568 kB
```

Let's further check our work by forcing the new swap device to be used. The following command creates a large amount of data by repeatedly converting the contents of `/dev/mem` (the UML's memory) into readable hex and feeds that into a little perl script that turns it into a very large string. We will use this string to fill up the system's memory and force it into swap.

```
UML# while true; do od -x /dev/mem ; done | perl -e 'my $s ; \
while(<STDIN>){ $s .= $_; } print length($s);'
```

At the same time, let's log in on a second console and watch the free memory disappear:

```
UML# while true; do free; sleep 10; done
```

You'll see the system start with almost all of its memory free:

```
             total       used       free     shared    buffers \
cached
Mem:        126696      21624     105072          0        536 \
7808
-/+ buffers/cache:      13280     113416
Swap:      1048568          0    1048568
```

The free memory will start disappearing, until we see a nonzero entry under `used` for the `Swap` row:

```
             total       used       free     shared    buffers     cached
Mem:        126696     124548       2148          0         76       7244
-/+ buffers/cache:     121823       9468
Swap:      1048568       6524    1042044
```

Here UML is behaving exactly as any physical system would—it is swapping when it is out of memory. Note that the host may have plenty of free memory, but the UML instance is confined to the memory we gave it.

PARTITIONED DISKS

You may have noticed another difference between the way we're using disks in UML and the way they are normally used on a physical machine. We haven't been partitioning them and putting filesystems and swap space on the partitions. This is a consequence of the ease of creating and adding new virtual disks to a virtual machine. With a physical disk, it's much less convenient, and sometimes impossible, to add more disks to a system. Therefore, you want to make the best of what you have, and that means being able to slice a physical disk into partitions that can be treated separately.

When UML was first released, there was no partition support for exactly this reason. I figured there was no need for partitions, given that if you want more disk space in your UML, you just create a new host file for it, and away you go.

This was a mistake. I underestimated the desire of my users to treat their UMLs exactly like their physical machines. In part, this meant they wanted to be able to partition their virtual disks. So, partition support for UML block devices ultimately appeared, and everyone was happy.

However, my original mistake resulted in some naming conventions that can be extremely confusing to a UML newcomer. Initially, UML block devices were referred to by number, for example, `ubd0`, `ubd1`, and so on. At first, these numbers corresponded to their minor device numbers, so when you made a device node for `ubd1`, the command was:

```
UML# mknod /dev/ubd1 b 98 1
```

When partition support appeared, this style of device naming was wrong in a couple of respects. First, you want to refer to the partition by number, as with `/dev/hda1` or `/dev/sdb2`. But does `ubd10` refer to block device 10 or partition 0 on device 1? Second, there is support for 16 partitions per device, so each block device gets a chunk of 16 device minor numbers to refer to them. For example, block device 0 has minor numbers 0 through 15, device 1 has minors 16 though 31, and so on. This breaks the previous convention that device numbers correspond to minor numbers, leading people to specify `ubd1` on the UML command line and not realize that it has minor device number 16 inside UML.

These two problems led to a naming convention that should have been present from the start. We name `ubd` devices in the same way as `hd` or `sd` devices—the disk number is specified with a letter (a, b, c, and so on), and the partition is a number. So, partition 1 on virtual disk 1 is `ubdb1`. When you add a second disk on the UML command line or via `mconsole`, it is `ubdb`, not `ubd1`. This eliminates the ambiguity of multidigit device numbers and the naming confusion. In this book, I will adhere to this convention, although my fingers still use `ubd0`, `ubd1`, and so on when I boot UML. In addition, the filesystems I'm using have references to `ubd0`, so commands such as `mount` and `df` will refer to names such as `ubd0` rather than `ubda`.

So, let's partition a `ubd` device just to see that it's the same as on a physical machine. First, let's make another host file to hold the device and plug it into the UML:

```
host% dd if=/dev/zero of=partitioned bs=1024 \
seek=$[ 1024 * 1024 ] count=1
1+0 records in
1+0 records out
host% uml_mconsole debian config ubdc=partitioned
OK
```

Now, inside the UML, let's use `fdisk` to chop this into partitions. Figure 3.2 shows my dialog with `fdisk` to create two equal-size partitions on this disk.

Now, I don't happen to have device nodes for these partitions, so I'll create them:

```
UML# mknod /dev/ubdc1 b 98 33
UML# mknod /dev/ubdc2 b 98 34
```

```
usermode:~# fdisk /dev/ubdc
Device contains neither a valid DOS partition table, nor Sun, SGI, or OSF
     disklabel
Building a new DOS disklabel. Changes will remain in memory only,
until you decide to write them. After that, of course, the previous
content won't be recoverable.


Command (m for help): p

Disk /dev/ubdc: 128 heads, 32 sectors, 512 cylinders
Units = cylinders of 4096 * 512 bytes

    Device Boot    Start       End    Blocks   Id  System

Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-512, default 1):
Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-512, default 512): 256

Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 2
First cylinder (257-512, default 257):
Using default value 257
Last cylinder or +size or +sizeM or +sizeK (257-512, default 512):
Using default value 512

Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.

WARNING: If you have created or modified any DOS 6.x
partitions, please see the fdisk manual page for additional
information.
Syncing disks.
usermode:~#
```

Figure 3.2 Using `fdisk` to create two partitions on a virtual disk

For some variety, let's make one a swap partition and the other a filesystem:

```
UML# mkswap /dev/ubdc1
Setting up swapspace version 1, size = 536850432 bytes
UML# mke2fs /dev/ubdc2
mke2fs 1.18, 11-Nov-1999 for EXT2 FS 0.5b, 95/08/09
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
131072 inodes, 262144 blocks
13107 blocks (5.00%) reserved for the super user
First data block=0
8 block groups
32768 blocks per group, 32768 fragments per group
16384 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376

Writing inode tables: done
```

And let's put them into action to see that they work as advertised:

```
UML# swapon /dev/ubdc1
UML# free
             total       used       free     shared    buffers \
    cached
Mem:        125128      69344      55784          0        448 \
     49872
-/+ buffers/cache:      19024     106104
Swap:      1572832          0    1572832
UML# mount /dev/ubdc2 /mnt
UML# df
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/ubd0              1032056    259444    720132  26% /
none                     62564         0     62564   0% /tmp
/dev/ubdc2              507748        13    481521   0% /mnt
```

So, we do, in fact, have another 512MB of swap space and a brand-new empty 512MB filesystem.

Rather than calling `swapon` by hand whenever we want to add some swap space to our UML, we can also just add the device to the UML's `/etc/fstab`. In our case, the relevant lines would be:

```
/dev/ubdb       swap            swap    defaults        0 0
/dev/ubdc1      swap            swap    defaults        0 0
```

However, if you do this, you must remember to configure the devices on the UML command line since they must be present early in boot when the filesystems are mounted.

UML DISKS AS RAW DATA

Normally, when you add a new block device to a UML, it will be used as either a filesystem or a swap device. However, some other possibilities are also useful with a UML. These work equally well on a physical machine but aren't used because of the lower flexibility of physical disks.

For example, you can copy files into a UML by creating a `tar` file on the host that contains them, plug that `tar` file into the UML as a virtual disk, and, inside the UML, untar the files directly from that device. So, on the host, let's create a tar file with some useful files in it:

```
host% tar cf etc.tar /etc
tar: Removing leading `/' from member names
```

When I did this on my machine, I got a bunch of errors about files that I, as a normal user, couldn't read. Since this is just a demo, that's OK, but if you were really trying to copy your host's `/etc` into a UML, you'd want to become root in order to get everything.

```
host% ls -l etc.tar
-rw-rw-rw-  1 jdike jdike 24535040 Feb 19 13:54 etc.tar
```

I did get about 25MB worth of files, so let's plug this `tar` file into the UML as device number 4, or `ubdd`:

```
host% uml_mconsole debian config ubdd=etc.tar
```

Now we can untar directly from the device:

```
UML# tar xf /dev/ubdd
```

This technique can also be used to copy a single file into a UML. Simply configure that file as a UML block device and use `dd` to copy it from the device to a normal file inside the UML filesystem. The drawback of this approach is that the block device will be an even multiple of the device block size, which is 512 bytes. So, a file whose size is not an even multiple of 512 bytes will have some padding added to it. If this matters, that excess will have to be trimmed in order to make the UML file the same size as the host file.

UML block devices can be attached to anything on the host that can be accessed as a file. Formally, the underlying host file must be seekable. This rules out UNIX sockets, character devices, and named pipes but includes block devices. Devices such as physical disks, partitions, CD-ROMs, DVDs, and floppies can be passed to UML as block devices and accessed from inside as `ubd` devices. If there is a filesystem

on the host block device, it can be mounted inside UML in exactly the same way as on the host, except for the different device name.

The UML must have the appropriate filesystem, either built-in or available as a module. For example, in order to mount a host CD-ROM inside a UML, it must have ISO-9660[2] filesystem support.

The properties of the host file show through to the UML device to a great extent. We have already seen that the host file's size determines the size of the UML block device. Permissions also control what can be done inside UML. If the UML user doesn't have write access to the host file, the resulting device will be only mounted read-only.

NETWORKING

Let's take a quick look at networking with UML. This large subject gets much more coverage in Chapter 7, but here, we will put our UML instance on the network and demonstrate its basic capabilities.

As with all other UML devices, network interfaces are virtual. They are formed from some host network interface that allows processes to send packets either to the host network stack or to another UML instance without involving the host network. Here, we will do the former and communicate with the host.

Processes can send and receive frames from the host in a variety of ways, including TUN/TAP, Ethertap, SLIP, and PPP.[3] All of these, except for PPP, are supported by UML. We will use TUN/TAP since it is intended for this purpose and doesn't have the limitations of the others. TUN/TAP is a driver on the host that creates a pipe, which is essentially a strand of Ethernet, between a process and the host networking system. The host end of this pipe is a network interface, typically named `tap<n>`, which can be seen using `ifconfig` just like the system's normal Ethernet device:

```
host% ifconfig tap0
tap0      Link encap:Ethernet  HWaddr 00:FF:9F:DF:40:D3
          inet addr:192.168.0.254  Bcast:192.168.0.255  \
```

2. The standard filesystem for a CD.
3. SLIP (Serial Line IP) and PPP (Point-to-Point Protocol) are protocols used for dialup Internet access. PPP has largely supplanted SLIP for this purpose. They are useful for UML because they provide virtual network interfaces that allow processes to send and receive network frames.

```
Mask:255.255.255.255
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:61 errors:0 dropped:0 overruns:0 frame:0
          TX packets:75 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:10931 (10.6 Kb)  TX bytes:8198 (8.0 Kb)
          RX bytes:15771 (15.4 Kb)  TX bytes:13466 (13.1 Kb)
```

This output resulted from a short UML session in which I logged in to the UML from the host, ran a few commands, and logged back out. Thus, the packet counters reflect some network activity.

It looks just like a normal network interface, and, in most respects, it is. It is just not attached to a physical network card. Instead, it is attached to a device file, `/dev/net/tun`:

```
host% ls -l /dev/net/tun
crw-rw-rw-  1 root root 10, 200 Sep 15  2003 /dev/net/tun
```

This file and the `tap0` interface are connected such that any packets routed to `tap0` emerge from the `/dev/net/tun` file and can be read by whatever process has opened it. Conversely, any packets written to this file by a process will emerge from the `tap0` interface and be routed to their destination by the host network system. Within UML, there is a similar pipe between this file and the UML Ethernet device. Here is the `ifconfig` output for the UML `eth0` device corresponding to the same short network session as above:

```
UML# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr FE:FD:C0:A8:00:FD
          inet addr:192.168.0.253  Bcast:192.168.0.255  \
Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:75 errors:0 dropped:0 overruns:0 frame:0
          TX packets:61 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          Interrupt:5
```

Notice that the received and transmitted packet counts are mirror images of each other—the number of packets received by the host `tap0` interface is the same as the number of packets transmitted by the UML `eth0` device. This is because these two interfaces are hooked up to each other back to back, with the connection being made through the host's `/dev/net/tun` file.

With this bit of theory out of the way, let's put our UML instance on the network. If we look at the interfaces present in our UML, we see only a loopback device, which isn't going to be too useful for us:

```
UML# ifconfig -a
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
```

Clearly, this needs to be fixed before we can do any sort of real networking. As you might guess from our previous work, we can simply plug a network device into our UML from the host:

```
host% uml_mconsole debian config eth0=tuntap,,,192.168.0.254
OK
```

This `uml_mconsole` command is telling the UML to create a new `eth0` device that will communicate with the host using its TUN/TAP interface, and that the IP address of the host side, the `tap0` interface, will be `192.168.0.254`. The repeated commas are for parameters we aren't supplying; they will be provided default values by the UML network driver.

My local network uses the `192.168.0.0` network, on which only about the first dozen IP addresses are in regular use. That leaves the upper addresses free for my UML instances. I usually use `192.168.0.254` for the host side of my TUN/TAP interface and `192.168.0.253` for the UML side. When I have multiple instances running, I use `192.168.0.252` and `192.168.0.251`, respectively, and so on.

Here, and everywhere else that you put UML instances on the network, you will need to choose IP addresses that work on your local network. They can't already be in use, of course. If suitable IP addresses are in short supply, you may be looking askance at my use of two addresses per UML instance. You can cut this down to one—the UML IP address—by reusing an IP address for the host side of the TUN/TAP interface. You can reuse the IP address already assigned to your host's `eth0` for this and everything will be fine.

Now we can look at the UML network interfaces and see that we have an Ethernet device as well as the previous loopback interface:

```
UML# ifconfig -a
eth0      Link encap:Ethernet  HWaddr 00:00:00:00:00:00
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          Interrupt:5
```

```
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
```

The `eth0` interface isn't running, nor is it configured with an IP address, so we need to fix that:

```
UML# ifconfig eth0 192.168.0.253 up
* modprobe tun
* ifconfig tap0 192.168.0.254 netmask 255.255.255.255 up
* bash -c echo 1 > /proc/sys/net/ipv4/ip_forward
* route add -host 192.168.0.253 dev tap0
* bash -c echo 1 > /proc/sys/net/ipv4/conf/tap0/proxy_arp
* arp -Ds 192.168.0.253 eth1 pub
```

This is more output than you normally expect to see from `ifconfig`, and in fact, it came from the kernel rather than `ifconfig`. This tells us exactly how the host side of the interface was set up and what commands were used to do it. If there had been any problems, the error output would have shown up here, and this would be the starting point for debugging the problem.

This setup enables the UML to communicate with the world outside the host and configures the host to route packets to and from the UML. In order to get UML on the network with the host, only the first two commands, `modprobe` and `ifconfig`, are needed. The `modprobe` command is precautionary since the host kernel may have TUN/TAP compiled or the `tun` module already loaded. Once TUN/TAP is available, the `tap0` interface is brought up and given an IP address, and it is ready to go.

The `bash` command tells the host to route packets rather than just dropping packets it receives that aren't intended for it. The `route` command adds a route to the UML through the `tap0` interface. This tells the host that any packet whose destination IP address is `192.168.0.253` (the address we gave to the UML `eth0` interface) should be sent to the `tap0` interface. Once there, it pops out of the `/dev/net/tun` file, which the UML network driver is reading, and from there to the UML `eth0` interface.

The final two lines set up proxy arp on the host for the UML instance. This causes the instance to be visible, from an Ethernet protocol point of view, on the local LAN. Whenever one Ethernet host wants to send a packet to another, it starts by knowing only the destination

IP address. If that address is on the local network, then the host needs to find out what Ethernet address corresponds to that IP address. This is done using Address Resolution Protocol (ARP). The host broadcasts a request on the Ethernet for any host that owns that IP address. The host in question will answer with its hardware Ethernet address, which is all the source host needs in order to build Ethernet frames to hold the IP packet it's trying to send.

Proxy arp tells the host to answer arp requests for the UML IP address just as though it were its own. Thus, any other machine on the network wanting to send a packet to the UML instance will receive an arp response from the UML host. The remote host will send the packet to the UML host, which will forward it through the `tap0` interface to the UML instance.

So, the host routing and the proxy arp work together to provide a network path from anywhere on the network to the UML, allowing it to participate on the network just like any other machine.

We can start to see this by using the simplest network tool, `ping`. First, let's make sure we can communicate with the host by pinging the `tap0` interface IP, `192.168.0.254`:

```
UML# ping 192.168.0.254
PING 192.168.0.254 (192.168.0.254): 56 data bytes
64 bytes from 192.168.0.254: icmp_seq=0 ttl=64 time=2.7 ms
64 bytes from 192.168.0.254: icmp_seq=1 ttl=64 time=0.2 ms
```

This works fine. For completeness, let's go the other way and ping from the host to the UML:

```
host% ping 192.168.0.253
PING 192.168.0.253 (192.168.0.253) 56(84) bytes of data.
64 bytes from 192.168.0.253: icmp_seq=0 ttl=64 time=0.130 ms
64 bytes from 192.168.0.253: icmp_seq=1 ttl=64 time=0.069 ms
```

Now, let's try a different host on the same network:

```
UML# ping 192.168.0.10
PING 192.168.0.10 (192.168.0.10): 56 data bytes
64 bytes from 192.168.0.10: icmp_seq=0 ttl=63 time=753.2 ms
64 bytes from 192.168.0.10: icmp_seq=1 ttl=63 time=6.3 ms
```

Here the routing and arping that I described above is coming into play. The other system, `192.168.0.10`, believes that the UML host owns the `192.168.0.253` address along with its regular IP and sends packets intended for the UML to it.

Now, let's try something real. Let's log in to the UML from that outside system:

```
host% ssh user@192.168.0.253
user@192.168.0.253's password:
Linux usermode 2.4.27-1um #6 Sun Jan 23 16:00:39 EST 2005 i686 unknown
Last login: Tue Feb 22 23:05:13 2005 from uml
UML%
```

Now, except for things like the fact we logged in as `user`, and the kernel version string and node name, we can't really tell that this isn't a physical machine. This UML is on the network in exactly the same way that all of the physical systems are, and it can participate on the network in all the same ways.

SHUTTING DOWN

The initial exploration of our UML is finished. We will cover everything in much more detail later, but this chapter has provided a taste of how UML works and how to use it. There is one final task: to shut down the UML. Figure 3.3 shows the output of the `halt` command run on the UML.

```
usermode:~# halt

Broadcast message from root (tty0) Wed Feb 23 00:00:32 2005...

The system is going down for system halt NOW !!
INIT: Switching to runlevel: 0
INIT: Sending processes the TERM signal
INIT: Sending processes the KILL signal
Stopping web server: apache.
/usr/sbin/apachectl stop: httpd (no pid file) not running
Stopping internet superserver: inetd.
Stopping MySQL database server: mysqld.
Stopping OpenBSD Secure Shell server: sshd.
Saving the System Clock time to the Hardware Clock...
hwclock: Can't open /dev/tty1, errno=19: No such device.
hwclock is unable to get I/O port access:  the iopl(3) call failed.
Hardware Clock updated to Wed Feb 23 00:00:38 EST 2005.
Stopping portmap daemon: portmap.
Stopping NFS kernel daemon: mountd nfsd.
Unexporting directories for NFS kernel daemon...done.
```

(*continues*)

Figure 3.3 Output from halting a UML

```
Stopping NFS common utilities: lockd statd.
Stopping system log daemon: klogd syslogd.
Sending all processes the TERM signal... done.
Sending all processes the KILL signal... done.
Saving random seed... done.
Unmounting remote filesystems... done.
Deconfiguring network interfaces: done.
Deactivating swap... done.
Unmounting local filesystems... done.
* route del -host 192.168.0.253 dev tap0
* bash -c echo 0 > /proc/sys/net/ipv4/conf/tap0/proxy_arp
* arp -i eth1 -d 192.168.0.253 pub
Power down.
* route del -host 192.168.0.253 dev tap0
* bash -c echo 0 > /proc/sys/net/ipv4/conf/tap0/proxy_arp
* arp -i eth1 -d 192.168.0.253 pub

~ 27056:
```

Figure 3.3 Output from halting a UML (*continued*)

Just as with a physical system, this is a mirror image of the boot. All the services that were running are shut down, followed by the kernel shutting itself down. The only things you don't see on a physical system are the networking messages, which are the mirror images of the ones we saw when bringing up the network. These are cleaning up the routing and the proxy arp that were set up when we configured UML networking.

Once all this has happened, the UML exits, and we are back to the shell prompt from which we started. The UML has simply vanished, just like any other process that has finished its work.

CHAPTER 4

A Second UML Instance

Now that we've seen a single UML instance in action, we will run two of them and see how they can interact with each other. First, we'll boot the two instances from a single filesystem, which should cause them to interact with each other by corrupting it, but we'll use a method that avoids that problem. Then, we'll continue the networking we started in the previous chapter by having the two instances communicate with each other in a couple of different ways. Finally, we'll look at some more unusual ways for UMLs to communicate that take advantage of the fact that, as virtual machines, they can do things that physical machines can't.

COW Files

First, let's fire up our UML instances with basically the same command line as before, with a couple of changes:

```
linux mem=128M ubda=cow1,/home/jdike/roots/debian_22 \
umid=debian1
```

and, in another window:

```
linux mem=128M ubda=cow2,/home/jdike/roots/debian_22 \
umid=debian2
```

The main difference is that we included the `ubda` switch on both command lines to add what is called a COW file to the UML block device. COW stands for Copy-On-Write, a mechanism that allows multiple UML instances to share a host file as a filesystem, mounting it read-write without seeing each others' changes or otherwise interfering with each other.

This has a number of benefits, including saving disk space and memory and simplifying the management of multiple instances.

COW works by attaching a second file to the UML block device that captures all of the changes made to the filesystem. A good analogy for this is a sheet of clear plastic placed over a painting. You can "change" the artwork by painting on the plastic without changing the underlying painting. When you look at it, you see your changes in the places where you painted on the plastic sheet, and you see the underlying work of art in the places you haven't touched. This is shown in Figure 4.1, where we give Mona Lisa a moustache.[1] We paint the mustache on a plastic sheet and place it over the *Mona Lisa*. We have committed artistic blasphemy without breaking any actual laws.

The COW file is the analog of the clear plastic sheet, and the original file that contains the UML filesystem is the analog of the painting.

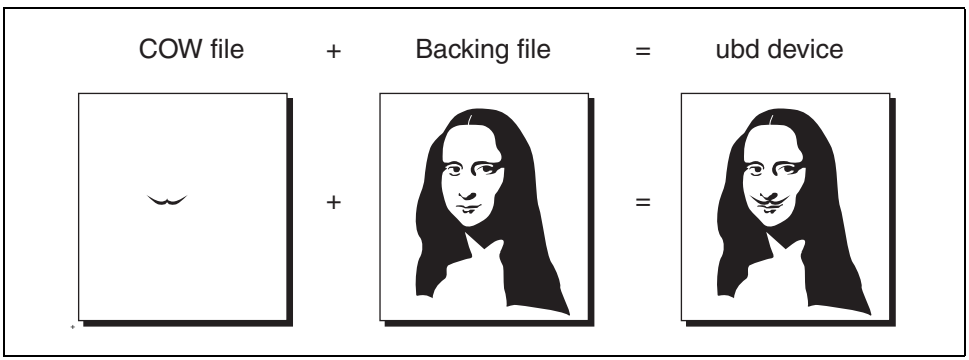


Figure 4.1 Using COW to give Mona Lisa a mustache without getting arrested

1. Which is one of my secret fantasies, and probably one of yours, too.

The COW is placed "over" the filesystem in the same way that the clear sheet is placed over the painting. When you modify a file on a COWed block device, the changed blocks are written to the COW file, not the underlying, "backing" file. This is the equivalent of painting on the sheet rather than on the painting. When you read a modified file, this is like looking at a spot on the painting that you've painted over on the plastic, and the driver reads the data from the COW file rather than the backing file.

Figure 4.2 shows how this works. We start with a COW file with no valid blocks and a fully populated backing file. If a process reads a block from this device, it will get the data that's in the backing file. If it then writes that block back, the new data will be written to the corresponding block in the COW file. At this point, the original block in the backing file is covered and will never be read again. All subsequent reads of that block will get the data from the COW file.

Thus, the backing file is never modified since all changes are stored in the COW file. The backing file can be treated as read-only, but the device as a whole is still read-write.

On a host with multiple UML instances, this has a number of advantages. First, all the instances can boot from the same backing file, as long as they have private COW files. This saves disk space. Since no instance is likely to change every file on its root filesystem, most of the data it uses will come from the shared backing file, and there will be only one copy of that on the host rather than one copy per

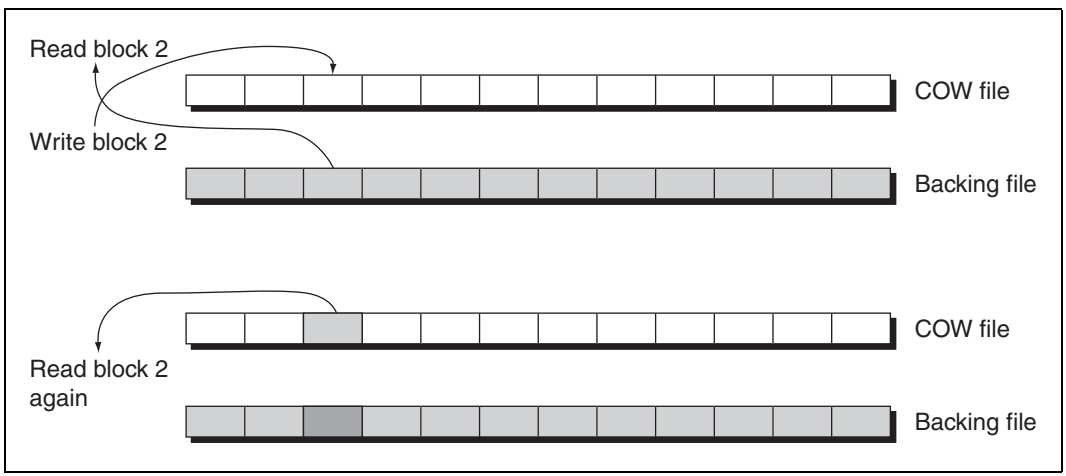


Figure 4.2 COW and backing files

instance. This may not seem like a big deal since disks are so big and so cheap these days, but system memory, as large as it is, is finite. Disk space savings will translate directly into host memory savings since, if there's only one block on disk that's shared by all the instances, it can be present in the host's page cache only once. Host memory is often the factor limiting the number of instances that a host can support, so this memory savings translates directly into greater hosting capacity.

Second, because the data that an instance has changed is in a separate file from the backing file, it is a lot easier to make backups. The only data that needs saving is in the COW file, which is generally much smaller than the backing file. In Chapter 6, we will see how to back up an instance's data in a few seconds for a reasonably-sized filesystem, without having to reboot it.

Third, using COW files for multiple instances on a host can improve the instances' performance. The reason is the elimination of data duplication described earlier. If an instance needs data that another instance has already used, such as the contents of `bash` or `libc`, it will likely already be in the host's memory, in its page cache. So, access to that data will be much faster than when it is still on disk. The first instance to access a certain block from the backing file will have to wait for it to be read from disk, but later instances won't since the host will likely still have it in memory.

Finally, there is a fairly compelling use for COW files even when you're just running a single UML instance. They make it possible to test changes to a filesystem and back them out if they don't work. For example, you can reconfigure a service, storing the changes in a COW file. If the changes were wrong, you can revert them simply by throwing out the COW file. If they are good, you can commit them by merging them into the backing file. We will look at how to do this later in the chapter.

Along with these advantages, there is one major disadvantage, which stems from the fact that the backing file is read-only. If the backing file is modified after it has COW files, those COW files will become invalid. The reason is that if one of the blocks on the backing file that changed was also changed in a COW file, reading that block would result in the COW data being read, rather than the new data in the backing file. This means that this `ubd` device would appear to be a combination of old data and new, resulting in data corruption for blocks

that contain file data and filesystem corruption for blocks that contain filesystem metadata.

The most common reason for wanting to modify the backing file is to upgrade the filesystem on it. This is understandable, but for backing files that have COW files based on them, this can't work. The right way to do upgrades in this case is to upgrade the COW files individually.

Going back to our two UML instances, which we booted from the same backing file, we see that they have almost exactly the same boot sequence. One exception is this from the first instance:

```
Creating "cow1" as COW file for "/home/jdike/roots/debian_22"
```

and this from the second:

```
Creating "cow2" as COW file for "/home/jdike/roots/debian_22"
```

You can specify, as we just did, a nonexistent file for the COW file, and the ubd driver will create the file when it starts up. Now that we have two UMLs booted, on the host, we can look at them:

```
host% ls -l cow*
-rw-r--r--  1 jdike jdike 1075064832 Apr 24 17:33 cow1
-rw-r--r--  1 jdike jdike 1075064832 Apr 24 17:34 cow2
```

Looking at those sizes, you may think I was fibbing when I went on about saving disk space. These files seem about the same size as the backing file. In fact, they look a bit larger than the backing file:

```
host% ls -l /home/jdike/roots/debian_22
-rw-rw-r--  1 jdike jdike 1074790400 Apr 23 21:40
    /home/jdike/roots/debian_22
```

I was not, in fact, fibbing, and therein lies an important fact about UML COW files. They are sparse, which means that even though their size implies that they occupy a certain number of blocks on disk (a disk block is 512 bytes, so the number of blocks occupied by a file is generally its size divided by 512, plus possibly another for the fragment at the end), many of those blocks are not occupied or allocated on disk.

There are two definitions for a file size here, and they conflict when it comes to sparse files. The first is how much data can be read from the file. The second is how much disk space the file occupies. Usually, these sizes are close. They won't be exactly the same because the fragment of the file at the end may occupy a full block. However, for a sparse file,

many data blocks will not be allocated on disk. When they are read, the read operation will produce zeros, but those zeros are not stored on disk. Only when a hitherto untouched block is written is it allocated on disk.

So, for our purposes, the "true" file size is its disk allocation, which you can see by adding the `s` switch to `ls`:

```
host% ls -ls cow*
540 -rw-r--r--  1 jdike jdike 1075064832 Apr 24 17:53 cow1
540 -rw-r--r--  1 jdike jdike 1075064832 Apr 24 17:54 cow2
```

The number in the first column is the number of disk blocks actually allocated to the file. This implies that the two COW files are actually using 270K of disk space, rather than the 1GB implied by the `ls -l` output. This space is occupied by data that the instances modified as they booted, generally log files and the like, which are touched by daemons and other system utilities as they start up.

We will talk more fully about COW file management later in this chapter, but here I will point out that the sparseness of COW files requires us to take some care when dealing with them. Primarily, this means being careful when copying them. The most common methods of copying a sparse file result in it becoming nonsparse—all the parts of the file that were previously unallocated on disk become allocated and that disk space filled with zeros. So, to avoid this, copying a COW file must be done in a sparseness-aware way. The main file copying utilities have switches for preserving sparseness when copying a file. For example, `cp` has `--sparse=auto` and `--sparse=always`, and `tar` has `-S` and `--sparse`.

Also, in order to detect that a backing file has been changed, thus invalidating any COW files based on it, the `ubd` driver compares the current modification time of the backing file to the modification time at the point that the COW file was created (which is stored in the COW file header). If they differ, the backing file has been modified, and a mount of the COW file may result in a corrupt filesystem.

Merely copying the backing file after restoring or moving it for some reason will change the modification time, even though the contents are unchanged. In this case, it is safe to mount a COW file that's based on it, but the `ubd` driver will refuse to do the mount. For this reason, it is important to also preserve the modification time of backing files, as well as sparseness, when copying them. However, everyone will forget once in a while, and later in this chapter, we will discuss some ways to recover from this.

Booting from COW Files

Now, we should look at what these COW files really mean from the perspective of the UML instances. First, we will make some changes in the two filesystems. In the first instance, let's copy `/lib` to `/tmp`:

```
UML1 # cp -r /lib /tmp
```

In the second, let's copy `/usr/bin` to `/tmp`:

```
UML2 # cp -r /usr/bin /tmp
```

In each, let's look at `/tmp` to see that the changes in one instance are not reflected in the other. First, the one where we copied `/lib`:

```
UML1 # ls -l /tmp
total 0
drwxr-xr-x    4 root     root         1680 Apr 25 13:02 lib
```

And next, the one with the copy of `/usr/bin`:

```
UML2 # ls -l /tmp
total 0
drwxr-xr-x    3 root     root         7200 Apr 25 13:07 bin
```

Here we can see that, even though they are booted off the same root filesystem, any changes they make are private. They can't be seen by other instances that have been booted from the same backing filesystem.

We can check this in another way by seeing how the sizes of the COW files on the host have changed:

```
host% ls -ls cow*
936 -rw-r--r--  1 jdike jdike 1075064832 Apr 25 13:22 cow0
1060 -rw-r--r--  1 jdike jdike 1075064832 Apr 25 13:22 cow1
```

Recall that after they booted, they both had 540 blocks allocated on disk. Now, they both have more than that—396 and 520 more, respectively. I chose to copy `/lib` and `/usr/bin` for this example because `/usr/bin` is noticeably larger than `/lib`, and making a copy of it should cause a significantly larger number of blocks to change in the COW file. This is exactly what happened.

So, at this point, we have two instances each booted on a 1GB filesystem, something that would normally take 2GB of disk space. With the use of COW files, this is taking 1GB plus 1MB, since together, the UMLs have made about 1MB worth of changes in this filesystem. There is a

commensurate saving of memory on the host because the data that both instances read from the filesystem will be present only once in the host's page cache instead of twice, as would be the case if they were booted from separate filesystems. Each new UML instance booted from the same filesystem similarly requires only enough host disk space to store its modifications, so the more instances you have booted from the same COWed filesystem, the more host disk space and memory you save.

I have one final remark on the subject of sharing filesystem images. Doing it using COW files is the only safe mechanism for sharing. If you booted two instances on the same filesystem, you would end up with a hopelessly corrupted filesystem. This is basically the same thing as booting two physical machines from the same disk, when both have direct access to the disk, as when it is dual-ported to both machines. Each instance will flush out data from memory to the filesystem file in such a way as to keep its own data consistent, but without regard to anything else that might be doing the same thing.

The only way for two machines to access the same data directly is for them to coordinate with each other, as happens with a clustering filesystem. They have to cooperate to maintain the consistency of the data they are sharing. We will see an example of such a UML cluster in Chapter 12.

In fact, you can't boot two UML instances from the same filesystem because UML locks the files it uses according to the access it needs to those files. It gets exclusive locks on filesystems it is going to write and nonexclusive read-only locks on files it will access but not write. So, when using a COW file, the UML instance will get an exclusive, read-write lock on the COW file and a nonexclusive read-only lock on the backing file. If another instance tries to get any lock on that COW file or a read-write lock on the backing file, it will fail. If that's the UML's root filesystem, the result will be an error message followed by a panic:

```
F_SETLK failed, file already locked by pid 21238
Failed to lock '/home/jdike/roots/debian_22', err = 11
Failed to open '/home/jdike/roots/debian_22', errno = 11
VFS: Cannot open root device "98:0" or unknown-block(98,0)
Please append a correct "root=" boot option
Kernel panic - not syncing: VFS: Unable to mount root fs on
      unknown-block(98,0)
```

This prevents people from accidentally booting two instances from the same filesystem and protects them from the filesystem corruption that would certainly follow.

Moving a Backing File

In order to avoid some basic mistakes, the UML block driver performs some sanity checks on the COW file and its backing file before mounting them. The COW file stores some information about the backing file:

- The filename
- Its size
- Its last modification time

Without these, the user would have to specify both the COW file and the backing file on the command line. If the backing file were wrong, without any checks, the result would be a hopelessly corrupted filesystem. The COW file is a block-level image of changes to the backing file. As such, it is tightly tied to a particular backing file and makes no sense with any other backing file.

If the backing file were modified, that would invalidate any already-existing COW files. This is the reason for the check of the modification time of the backing file.

However, this check gets in the way of moving the backing file since the file, in its new location, would normally have its modification time updated. So, it is important to preserve the timestamp on a backing file when moving it. A number of utilities have the ability to do this, including

- `cp` with the `-a` or `-p` switch
- `tar` with the `-p` switch

After you have carefully moved the backing file, you still need to get the COW file header to contain the new location. You do this by booting an instance on the COW file, specifying both filenames in the device description:

```
ubda=cow-file,new-backing-file
```

The UML block driver will notice the mismatch between the command line and the COW file header, make sure the size and timestamp of the new location are what it expects, and update the backing file location. When this happens, you will see a message such as this:

```
Backing file mismatch - "debian30" requested,
"/home/jdike/linux/debian30" specified in COW header of "cow2"
Switching backing file to 'debian30'
```

However, at some point, you will forget to preserve the timestamp, and the COW file will appear to be useless. If it's a UML root device, the boot will fail like this:

```
mtime mismatch (1130814229 vs 1130970724) of COW header vs \
    backing file
Failed to open 'cow2', errno = 22
VFS: Cannot open root device "98:0" or unknown-block(98,0)
Please append a correct "root=" boot option
```

All is not lost. You need to restore the timestamp on the new backing file by hand, taking the proper timestamp from the error message above:

```
host% date --date="1970-01-01 UTC 1130814229 seconds"
Mon Oct 31 22:03:49 EST 2005
host% touch --date="Mon Oct 31 22:03:49 EST 2005" debian30
```

The `date` command converts the timestamp, which is the number of seconds since January 1, 1970, into a form that `touch` will accept. In turn, the `touch` command applies that timestamp as the modification time of the backing file.

To minimize the amount of typing, you can abbreviate this operation as follows:

```
touch --date="`date --date='1970-01-01 UTC 1130814229 seconds'`" \
      debian30
```

You may wonder why this isn't automated like the filename operation. When both the backing filename and timestamp don't match the information in the COW header, the only thing left is the file size. And there aren't enough common file sizes to have any sort of reasonable guarantee that you're associating the COW file with the correct backing file. I require that you update the timestamp by hand so you look at the file in question and can catch a mistake before it happens.

Merging a COW File with Its Backing File

Sometimes you want to merge the modified data in a COW file back into the backing file. For example, you may have created a COW file in order to test a modification of the filesystem, such as the installation or modification of a service. If the results are bad, you can back out to the original filesystem merely by throwing out the COW file. If the results

are good, you want to keep them by merging them back into the backing file—in essence, committing them.

The tool used to do this is called `uml_moo`.[2] Using it is simple. You just need to decide whether you want to do an in-place merge or create a new file, leaving the original COW and backing files unchanged. The second option is recommended if you're feeling paranoid, although making a copy of the backing file before doing an in-place merge is just as safe. Most often, people choose based on the amount of disk space available on the host—if it's low, they do an in-place merge.

Create a new file by doing this:

```
host% uml_moo COW-file new-backing-file
```

Do an in-place merge like this:

```
host% uml_moo -d COW-file
```

You can use the `-b` switch to specify the true location of the backing file in the event that the name stored in the COW file header is incorrect. This happens most often when the COW file was created inside a `chroot` jail. In this case, the backing file specified in the COW file will be relative to the jail and thus wrong outside the jail. For example, if you had a COW file created by a UML instance that was jailed to `/jail` and contains `/rootfs` as the backing file, you would do an in-place merge like this:

```
host% uml_moo -b /jail/rootfs -d /jail/cow-file
```

NETWORKING THE UML INSTANCES

After seeing the example of two UML instances not interacting (i.e., not corrupting each other's filesystems) when you might expect them to, let's make them interact when we want them to. We will create a small private network with just these two instances on it and see that they can use it to communicate with each other in the same way that physical machines communicate on a physical network.

2. I can only offer my deep and humble apologies for the name—a bovine theme pervades the COW file support in UML.

For a pair of virtual machines, the basic requirement for setting up a network between them is some method of exchanging packets. Since packets are just hunks of data, albeit specially formatted ones, in principle, any interprocess communication (IPC) mechanism will suffice. All that's needed in UML is a network driver that can send and receive packets over that IPC mechanism.

This is enough to set up a private network that the UML instances can use to talk to each other, but it will not let them communicate with anything else, such as the host or anything on the Internet. Communicating with the outside world, including the host, requires root privileges at some point. The instance needs to send packets to the host and have them be handled by its network subsystem. This ability requires root privileges because it implies that the instance is a networking peer of the host and could foul up the network through misconfiguration or malice.

Here, we will introduce UML networking by setting up a two-machine private network with no access to the outside world. We will cover networking fully in Chapter 7, including access to the host and the Internet.

As I said earlier, in principle, any IPC mechanism can be used to construct a virtual network. However, they differ in their convenience, which is strongly related to how well they map onto a network. Fundamentally, Ethernet is a broadcast medium in which a message sent by one host is seen by all the others on the same Ethernet, although, in practice, the broadcasting is often suppressed by intelligent hardware such as switches. Most IPC mechanisms, on the other hand, are point to point. They have two ends, with one process at each end, and a message sent by a process at one end is seen by the host at the other.

This mismatch makes most IPC mechanisms not well suited for setting up a network. Each host would need a connection to each other host, including itself, so the total number of connections in the network would grow quadratically with the number of hosts. Further, each packet would need to be sent individually to each host, rather than having it sent once and received by all the other hosts.

However, one broadcast IPC mechanism is available: multicasting. This little-used networking mechanism allows processes to join a group, called a multicast group. When a message is sent to this group, it is received by all the processes that have joined the group. This nicely matches the semantics needed by a broadcast medium, with one caveat—it matches an Ethernet device that's connected by a hub, not a switch. A hub repeats every packet to every host connected to it, while a switch knows which Ethernet MAC addresses are associated with

each of its ports and sends each packet only to the hosts it's intended for. With a multicast virtual network, as with a hub, each host will see all of the packets on the network and will have to discard the ones not addressed to it.

To start things off, we need Ethernet interfaces in our UML instances. To do this, we need to plug them in:

```
host% uml_mconsole debian1 config eth0=mcast
OK
host% uml_mconsole debian2 config eth0=mcast
OK
```

This hot-plugs an Ethernet device into each instance. If you were starting them from the shell here, you would simply add `eth0=mcast` to their command lines.

Now, if you go back to one of the instances and run `ifconfig`, you will notice that it has an `eth0`:

```
UML1# ifconfig -a
eth0      Link encap:Ethernet  HWaddr 00:00:00:00:00:00
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          Interrupt:5

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
```

You'll see the same thing has happened in the other UML.

Now we need to bring them up, so we'll assign IP addresses to them. We'll use `192.168.0.1` for one instance:

```
UML1# ifconfig eth0 192.168.0.1 up
```

and similarly in the other instance, we'll assign `192.168.0.2`:

```
UML2# ifconfig eth0 192.168.0.2 up
```

Don't worry if you are already using these addresses on your own network—we have set up an isolated network, so there can't be any conflicts between IP addresses if they can't exchange packets with each other.

Running `ifconfig` again shows that both interfaces are now up and running:

```
UML1# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr FE:FD:C0:A8:00:01
          inet addr:192.168.0.1  Bcast:192.168.0.255
    \Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          Interrupt:5
```

No packets have been transmitted or received, so we need to fix that. Let's ping the second UML from the first:

```
UML1# ping 192.168.0.2
PING 192.168.0.2 (192.168.0.2): 56 data bytes
64 bytes from 192.168.0.2: icmp_seq=0 ttl=64 time=9.3 ms
64 bytes from 192.168.0.2: icmp_seq=1 ttl=64 time=0.2 ms
64 bytes from 192.168.0.2: icmp_seq=2 ttl=64 time=0.2 ms

--- 192.168.0.2 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.2/3.2/9.3 ms
```

This establishes that we have basic network connectivity. To see some more interesting network action, let's request a Web page from the other UML. Since we don't have any ability to run a graphical Web browser inside the UML yet, we'll use the command-line tool `wget`:

```
UML1# wget -O - http://192.168.0.2
--15:51:10--  http://192.168.0.2:80/
           => `-'
Connecting to 192.168.0.2:80... connected!
HTTP request sent, awaiting response... 200 OK
Length: 4,094 [text/html]

    0K -><!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
<HEAD>
```

Following that snippet, you'll see the rest of the default Apache home page as shipped by Debian. If you want a more interactive Web experience at this point, you can just run `lynx`, the text-mode Web browser, with the same URL, and you'll see a pretty good text representation of that page. The external links (those that point to debian.org, apache.org, and the like) will not work because these instances don't

have access to the outside network. However, any links internal to the other UML instance, such as the Apache documentation, should work fine.

Now that we have basic networking between the two instances, I am going to complicate the configuration as much as possible, given that we have only two hosts, and add them both to what amounts to a second Ethernet network. I'm going to keep this network separate from the current one, and to do so, I need to specify a different port from the default. We specified no multicast parameters when we set up the first network, so the UML network driver assigned default values. To keep this new network separate from the old one, we will provide a full specification of the multicast group:

```
host% uml_mconsole debian1 config eth0=mcast,,239.192.168.1,1103,1
OK
host% uml_mconsole debian2 config eth0=mcast,,239.192.168.1,1103,1
OK
```

We are separating this network from the previous one by using the next port. You can see how things are set up by looking at the kernel message log:

```
UML# dmesg | grep mcast
Configured mcast device: 239.192.168.1:1102-1
Netdevice 0 : mcast backend multicast address: \
    239.192.168.1:1102, TTL:1
Configured mcast device: 239.192.168.1:1103-1
Netdevice 1 : mcast backend multicast address: \
    239.192.168.1:1103, TTL:1
```

We used the same default IP address, but used port 1103 instead of the default 1102. We are still defaulting the second parameter, which is the hardware MAC address that will be assigned to the adapters. Since we're not providing one, it will be derived from the first IP assigned to the interface.

Again, if you run `ifconfig`, you will see that another interface has materialized on the system:

```
UML1# ifconfig -a
eth0      Link encap:Ethernet  HWaddr FE:FD:C0:A8:00:01
          inet addr:192.168.0.1  Bcast:192.168.0.255
     \Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1363 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1117 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          Interrupt:5
```

```
eth1      Link encap:Ethernet  HWaddr 00:00:00:00:00:00
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          Interrupt:5

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:546 errors:0 dropped:0 overruns:0 frame:0
          TX packets:546 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
```

We'll bring these up with IP addresses on a different subnet:

```
UML1# ifconfig eth0 192.168.1.1 up
```

and:

```
UML2# ifconfig eth0 192.168.1.2 up
```

As before, we can verify that we have connectivity by pinging one from the other:

```
UML# ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: icmp_seq=0 ttl=64 time=18.6 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=0.4 ms

--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.4/9.5/18.6 ms
```

Now that we have two networks, we can do some routing experiments. We have two interfaces on each UML instance, on two different networks, with correspondingly different IP addresses. We can pretend that the `192.168.1.0/24` network is the only one working and set up one instance to reach the `192.168.0.0/24` interface on the other. So, let's first look at the routing table on one of the instances:

```
UML# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    \
     Use Iface
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0      \
     0 eth1
192.168.0.0     0.0.0.0         255.255.255.0   U     0      0      \
     0 eth0
```

We will delete the `192.168.0.0/24` route on both instances to pretend that network doesn't work any more:

```
UML1# route del -net 192.168.0.0 netmask 255.255.255.0 dev eth0
```

and identically on the other:

```
UML2# route del -net 192.168.0.0 netmask 255.255.255.0 dev eth0
```

Now, let's add the route back in, except we'll send those packets through `eth1`:

```
UML1# route add -net 192.168.0.0 netmask 255.255.255.0 dev eth1
```

and on the other:

```
UML2# route add -net 192.168.0.0 netmask 255.255.255.0 dev eth1
```

Now, the routing table looks like this:

```
UML# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref
     Use Iface
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0
     0 eth1
192.168.0.0     0.0.0.0         255.255.255.0   U     0      0
     0 eth1
```

Before we ping the other side to make sure that the packets are traveling the desired path, let's look at the packet counts on `eth0` and `eth1` before and after the ping. Running `ifconfig` shows this output for `eth0`:

```
RX packets:3597 errors:0 dropped:0 overruns:0 frame:0
TX packets:1117 errors:0 dropped:0 overruns:0 carrier:0
```

and this for `eth1`:

```
RX packets:8 errors:0 dropped:0 overruns:0 frame:0
TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
```

The rather large packet count for `eth0` comes from my playing with the network without recording here everything I did. Also, notice that the receive count for `eth1` is double the transmit count. This is because of the hublike nature of the multicast network that I mentioned earlier. Every packet is seen by every host, including the ones the host

itself sent. The UML received its own transmitted packets and the replies. Since there was one reply for each packet sent out, the number of packets received will be exactly double the number transmitted.

Now, let's test our routing by pinging one instance from the other:

```
UML# ping 192.168.0.251
PING 192.168.0.251 (192.168.0.251): 56 data bytes
64 bytes from 192.168.0.251: icmp_seq=0 ttl=64 time=19.9 ms
64 bytes from 192.168.0.251: icmp_seq=1 ttl=64 time=0.4 ms
64 bytes from 192.168.0.251: icmp_seq=2 ttl=64 time=0.4 ms

--- 192.168.0.251 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.4/6.9/19.9 ms
```

This worked, so we didn't break anything. Let's check the packet counters for `eth0` again:

```
RX packets:3597 errors:0 dropped:0 overruns:0 frame:0
TX packets:1117 errors:0 dropped:0 overruns:0 carrier:0
```

and for `eth1`:

```
RX packets:18 errors:0 dropped:0 overruns:0 frame:0
TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
```

Nothing went over `eth0`, as planned, and the pings went over `eth1` for both UMLs. So, even though the `192.168.0.0/24` network is still up and running, we persuaded the UMLs to pretend it wasn't there and to use the `192.168.1.0/24` network instead.

Although this is a simple demonstration, we just simulated a scenario you could run into in real life, and dealing with it incorrectly in real life could seriously mess up a network.

For example, say you have two parallel networks, with one acting as a backup for the other. If one goes out of commission, you want to fail over to the other. Our scenario is similar to having the `192.168.0.0/24` network fail. Leaving the `eth0` interfaces running is consistent with this because they would remain up on a physical machine on a physical Ethernet—they would just have 100% packet loss. Having somehow seen the network fail, we reset the routes so that all traffic would travel over the backup network, `192.168.1.0/24`. And we did it with no extra hardware and no Ethernet cables, just a standard Linux box and some software.

Setting this up and doing the failover without having tested the procedure ahead of time would risk fouling up an entire network, with

its many potentially unhappy users, some of whom may have influence over the size of your paycheck and the duration of your employment. Developing the procedure without the use of a virtual network would involve setting up two physical test networks, with physical machines and cables occupying space somewhere. Simply setting this up to the point where you can begin simulating failures would require a noticeable amount of time, effort, and equipment. In contrast, we just did it with no extra hardware, in less than 15 minutes, and with a handful of commands.

A VIRTUAL SERIAL LINE

We are going to round out this chapter with another example of the two UML instances communicating over simulated hardware. This time, we will use a virtual serial line running between them to log in from one to the other.

This serial line will be constructed from a host pseudo-terminal, namely, a UNIX 98 pts device. Pseudo-terminals on UNIX are pipes—whatever goes in one end comes out the other, possibly with some processing in between, such as line editing. This processing distinguishes pseudo-terminals from normal UNIX pipes. The end that's opened first is the pty end, and it's the master side—the device doesn't really exist until this side is opened. So, the instance to which we are going to log in will open the master side of the device, and later, the slave side will be opened by the other instance when we log in over it.

We are going to make both ends of the device appear inside the instances as normal, hardwired terminals. One instance is going to run a getty on it, and we will run a screen session inside the other instance attached to its terminal.

To get started, we need to identify an unused terminal in both instances. There are two ways to do this—read `/etc/inittab` to find the first terminal that has no getty running on it, or run `ps` to discover the same thing. The relevant section of `inittab` looks like this:

```
# /sbin/getty invocations for the runlevels.
#
# The "id" field MUST be the same as the last
# characters of the device (after "tty").
#
# Format:
#  <id>:<runlevels>:<action>:<process>
0:2345:respawn:/sbin/getty 38400 tty0
1:2345:respawn:/sbin/getty 38400 tty1
```

```
2:2345:respawn:/sbin/getty 38400 tty2
3:2345:respawn:/sbin/getty 38400 tty3
4:2345:respawn:/sbin/getty 38400 tty4
5:2345:respawn:/sbin/getty 38400 tty5
6:2345:respawn:/sbin/getty 38400 tty6
7:2345:respawn:/sbin/getty 38400 tty7
c:2345:respawn:/sbin/getty 38400 ttyS0
```

It appears that `tty8` is unused. `ps` confirms this:

```
UML1# ps uax | grep getty
root       153  0.0  0.3  1084  444 tty1     S    14:07   0:00
     /sbin/getty 38400 tty1
root       154  0.0  0.3  1088  448 tty2     S    14:07   0:00
     /sbin/getty 38400 tty2
root       155  0.0  0.3  1084  444 tty3     S    14:07   0:00
     /sbin/getty 38400 tty3
root       156  0.0  0.3  1088  448 tty4     S    14:07   0:00
     /sbin/getty 38400 tty4
root       157  0.0  0.3  1088  452 tty5     S    14:07   0:00
     /sbin/getty 38400 tty5
root       158  0.0  0.3  1088  452 tty6     S    14:07   0:00
     /sbin/getty 38400 tty6
root       159  0.0  0.3  1088  452 tty7     S    14:07   0:00
     /sbin/getty 38400 tty7
root       160  0.0  0.3  1084  444 ttyS0    S    14:07   0:00
     /sbin/getty 38400 ttyS0
```

This is the same on both instances, as you would expect, so we will use `tty8` as the serial line on both.

First we need to plug in a properly configured `tty8` to the master UML instance, the one to which we will be logging in. We do this with `uml_mconsole` on the host, configuring `con8`, which is the `mconsole` name for the device that is `tty8` inside UML:

```
host% uml_mconsole debian2 config con8=pts
OK
```

Now, the master UML instance has a `tty8`, and we need to know which pseudo-terminal on the host it allocated so that we can connect the other instance's `tty8` to the other end of it. Right now, it's not connected to anything, as it waits until the device is opened before allocating a host terminal. So, to get something to open it, we'll run `getty` on it:

```
UML2# /sbin/getty 38400 tty8
```

Now we need to know what the other end of the pts device is, since that's determined dynamically for these devices:

```
host% uml_mconsole debian2 config con8
OK pts:/dev/pts/28
```

This tells us how to configure `con8` on the slave UML:

```
host% uml_mconsole debian1 config con8=tty:/dev/pts/28
OK
```

Here we are using `tty` instead of `pts` as the device type because the processes of opening the two sides of the device are slightly different, and we are opening the slave side here.

This will just sit there, so we now go to the slave UML instance and attach `screen` to its `tty8`:

```
UML1# screen /dev/tty8
```

Figure 4.3 shows what we have constructed. The two UML consoles are connected to opposite ends of the host's `/dev/pts/28` and communicate through it. From inside the UML instances, it appears that the two UML `/dev/tty8` devices are connected directly to each other.

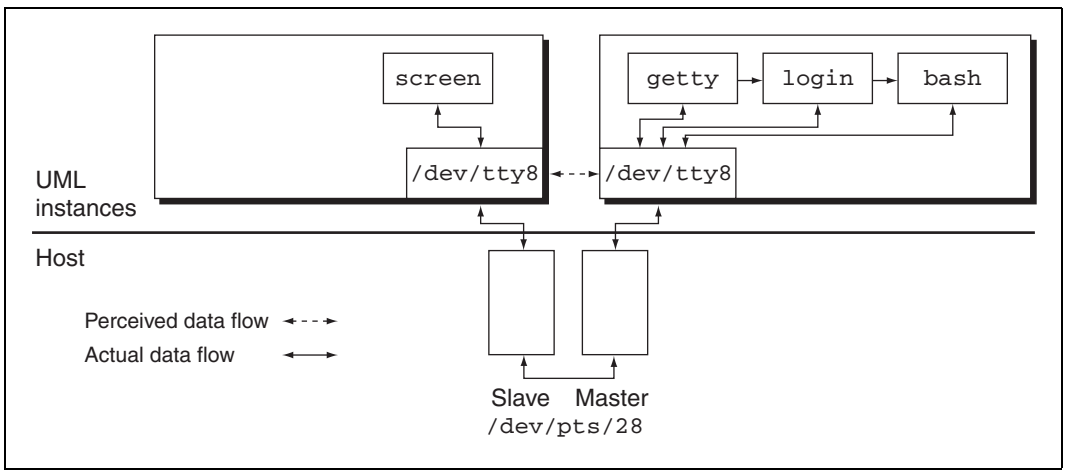


Figure 4.3 A virtual serial line. The two UML `/dev/tty8` devices are connected to the host's `/dev/pts/28` pseudo-terminal, the master side connected to the UML instance that will be logged into and the slave side connected to the UML instance that will be logging in. The master side is connected to the UML instance's `getty`, `login`, and `bash` as the login proceeds. On the other side, `screen` is connected to the UML instance's `/dev/tty8`, which is attached to the slave side of the host pseudo-terminal. The solid lines show the actual flow of data through the UML consoles and the host pseudo-terminal. The dashed line shows the flow of data apparent to the UML users, who see the two UML consoles directly connected to each other.

Now you should see a login prompt in the `screen` session. Log in and determine that it really is the other instance. During the examples in this chapter, we've copied different things into `/tmp`, assigned different IP addresses to their network interfaces, and played with their routing tables, so this should not be hard to verify.

Once you log out, you'll notice that the getty exits back to the shell, and you get no login prompt in the `screen` session. This is the purpose of the `respawn` on the `getty` lines in `/etc/inittab`. If you wrapped the `getty` command in an infinite loop, you would be doing a passable imitation of `init`. However, we will just exit the screen session (`^A K`) to get back to the prompt in the other instance.

We are done with these UMLs, so you can just halt them and remove their COW files if you want to reclaim whatever disk space they consumed.

The point of this exercise was not to demonstrate that two UML instances can be used to simulate a serial line—physical serial lines are not hard to come by and not that hard to set up. Rather, it was to demonstrate how easily a virtual device on the host can be pressed into service as a physical device inside UML. A serial line is probably the simplest example of this, which is why I used it. Out of the box, UML can emulate many other sorts of hardware, and for other types, it is fairly simple to write a UML driver that emulates the device. Other examples include using shared memory on the host to emulate a device with memory-mapped I/O, which some embedded systems developers have done, and using shared memory to emulate a cluster interconnect, with multiple UML instances on the host being the emulated cluster.

More prosaic, and more common, is the need to emulate a network environment for purposes such as setting it up, reconfiguring it, and testing fault handling. We saw an example of testing failover from a failed network to a hot spare network. This only scratches the surface of what can be done with a virtual network. A network of UMLs can be configured in any way that a physical network can and a lot of ways that a physical network can't, making UML an ideal way to set up, develop, and test networks before physically building them.

CHAPTER 5

Playing with a UML Instance

By now, you have at least a basic idea of what UML is and how it can be used. In this chapter, we will see a wider variety of things we can do with UML. We will set up a basic network and use it to gain access to the host and to the outside network, and also access the UML instance from the outside. We will continue playing with virtual devices, seeing how they can be used like physical devices and what they can do that physical devices can't.

USE AND ABUSE OF UML BLOCK DEVICES

First, let's look at ways to copy data into a UML instance from the host without using the network. We will use UML block devices for this, tying them to files containing the data that we want to access inside the instance. Until now, we have used UML block devices only for filesystems. However, like physical disks, block devices can contain any data whatsoever, and that data can be accessed by anything that understands its format. Putting a filesystem on a disk formats the disk's data in a particular way that can be understood by the filesystem

that will mount it. However, if you don't need to mount the disk as a filesystem, the data on it can be anything you want.

For example, let's say that we want to copy a directory from the host to the instance. In this example, we will create a `tar` file on the host containing the directory, attach a UML block device to the file, and untar the directory inside the instance. We saw this in Chapter 3, but I will go into more depth here.

To start, we need the `tar` file. I will use the host's `/etc` here:

```
host% tar cpf etc.tar /etc
host% ls -l etc.tar
-rw-rw-rw-  1 jdike jdike 25149440 May 13 22:28 etc.tar
```

I ran `tar` as a normal user and got a bunch of errors from files I didn't have permission to read. That's fine for an example, but if you really wanted all those files, you would run `tar` as root to ensure that they all end up in the `tar` file.

At the end of the previous chapter, we shut down our UML instances, so if you don't have one running now, start one up.

We now have a 25MB file that we will turn into a UML block device, using `uml_mconsole`:

```
host% uml_mconsole debian config ubdb=etc.tar
OK
```

This causes a second block device, `/dev/ubdb`, to come into existence inside our instance. Rather than mounting it as a filesystem, as we have done before, we will treat it as a tape drive. These days, `tar` is often used to archive data in files and to retrieve files from those files. It used to be more common to use `tar` to write the data directly to a device, usually a tape drive. We are going to treat our new block device similarly and tar the directory off the device directly.

First, let's go to the UML, see if `/dev/ubdb` contains something that `tar` recognizes, and ask it to show us what's on the device:

```
UML# tar tf /dev/ubdb | head
etc/
etc/sysconfig/
etc/sysconfig/network-scripts/
etc/sysconfig/network-scripts/ifdown-aliases
etc/sysconfig/network-scripts/ifcfg-lo
etc/sysconfig/network-scripts/ifdown
etc/sysconfig/network-scripts/ifdown-ipsec
etc/sysconfig/network-scripts/ifdown-ippp
etc/sysconfig/network-scripts/ifup-aliases
etc/sysconfig/network-scripts/ifdown-ipv6
```

That looks a lot like a `/etc`, so let's pull it off for real:

```
UML# tar xpf /dev/ubdb
```

Now you will see an `/etc` directory in your current directory. If you run `ls` on it, you will see that it is, in fact, the same as what you copied on the host.

This should make it clear that the data on a UML block device can be any format and that all you need to pull the data off the device inside the UML instance is a utility that understands the format.

Let's pull the data off in a way that assumes nothing about the format. We will just make a raw copy of the device in a file inside UML and see that it contains what we expect. To start, remove the `/etc` directory we just made:

```
UML# rm -rf etc
```

Now, let's use `dd` to copy the device into a file:

```
UML# dd if=/dev/ubdb of=etc.tar
49120+0 records in
49120+0 records out
```

We can check whether `tar` still thinks it contains a copy of `/etc`:

```
UML# tar xpf etc.tar
```

That finishes successfully, and you can again check with `ls` that you extracted the same directory as before.

As a final example using this `tar` file, we will compress the file before attaching it to a block device, and then uncompress and untar it inside UML. Again, let's remove our copy of `/etc`:

```
UML# rm -rf etc
```

Now, back on the host, we compress the `tar` file and attach it to UML block device 2:

```
host% gzip etc.tar
host% uml_mconsole debian config ubdc=etc.tar.gz
OK
```

Back inside the UML instance, we now uncompress the compressed `tar` file to `stdout` and pipe that into `tar`, hopefully extracting the same directory that we did before:

```
UML# gunzip -c < /dev/ubdc | tar xf -
```

Again, you can check that `etc` is the same as in the previous examples.

When copying files into UML using this method, you need to be careful about lengths. We are mapping a host file, which can be any length, onto a block device, which is expected to be a multiple of 512 bytes long—the size of a disk sector in Linux. Block devices are expected to contain sectors, with no bytes left over. To see how this affects files copied into UML, let's copy a single, odd-length file through a block device.

Locate a file on the host whose length is not an even multiple of 512 bytes.

On my system, `/etc/passwd` is suitable:

```
host% ls -l /etc/passwd
-rw-r--r--  1 root root 1575 Dec 10 18:38 /etc/passwd
```

Let's attach this file to UML:

```
host% uml_mconsole debian config ubdd=/etc/passwd
OK
```

Here's what we get when we copy it inside UML:

```
UML# dd if=/dev/ubdd of=passwd
4+0 records in
4+0 records out
UML# ls -l passwd
-rw-r--r--  1 root root 2048 May 13 23:48 passwd
```

Notice that the size changed. If you look at the file with a suitable utility, such as `od`, you will see that the extra bytes are all zeros.

There is a mismatch between a file with no size restrictions being mapped to a device that must be an even number of sectors. The UML block driver has to bridge this gap somehow. When it reaches the end of a host file and has read only a partial sector, it pads the sector with zeros and pretends that it read the entire thing. This is a necessary fiction in order to generally handle the end of a file, but it results in the block driver copying more data than it should.

To deal with this problem, we need to tell `dd` exactly how much data to copy. The UML block driver will still pad the last sector with zeros, but `dd` won't copy them.

My `/etc/passwd` is 1575 bytes long, so this is what we will tell `dd` to copy:

```
UML# dd if=/dev/ubdd of=passwd bs=1 count=1575
1575+0 records in
1575+0 records out
UML# ls -l passwd
-rw-r--r--  1 root root 1575 May 13 23:56 passwd
```

The `bs=1` argument to `dd` tells it to copy data in units of a single byte, and the `count` argument tells it to copy 1575 of those units.

Now, the size is what we expect, and if you check the contents, you will see that they are, in fact, the same.

Some file formats are self-documenting in terms of their length—it is possible to tell whether extra data has been added to the length of the file. `tar` and `bzip` files are two examples—`tar` and `bunzip2` can tell when they've reached the end of the data they are supposed to process, and `bunzip2` will complain about the extra data. If you are copying data from a `ubd` device that is in one of these formats, you can use the device directly as the data source. You don't need to copy the correct number of bytes from the device into a file in order to recreate the original data.

NETWORKING AND THE HOST

Now, let's move to a more conventional method of transferring data between machines. Pluggable block devices are cute and sometimes invaluable, but a network is more conventional, more flexible, and usually easier to use. We saw a bit of UML networking in the previous chapter, which showed that UML instances can be used to construct an isolated network. But the value of networking lies in accessing the outside world. Let's do this now.

We will plug a network interface into the UML as we did before, but we are going to use a different host mechanism to transfer the packets:

```
host% uml_mconsole debian config eth0=tuntap,,,192.168.0.254
OK
```

At this point, the IP addresses we use will be visible to the outside network, so choose ones that aren't used on your network. If you are using `192.168.0.254` already, change the `uml_mconsole` command to specify an unused IP address (or one that is already used by a different

interface on the host, if IP addresses are scarce on your network). Inside the UML, you should see a message similar to this:

```
Netdevice 0 : TUN/TAP backend - IP = 192.168.0.254
```

If it doesn't appear on the main console, you will be able to see it by running `dmesg`. Some distributions don't have their logging configured so that kernel messages appear on the main console. In this case, running `dmesg` is the most convenient way to see the recent kernel log.

In this example, we are setting up a virtual network interface on the host that will be connected to the UML's `eth0`. There are a number of mechanisms for doing this, such as SLIP, PPP, Ethertap, and TUN/TAP. TUN/TAP is the newest and most general-purpose one of the bunch. It provides a file that, when opened by a process, allows that process to receive and transmit Ethernet frames to the system's network stack.

Figure 5.1 shows an example of using TUN/TAP. Here we have a system with `eth0`, a wired Ethernet interface; `eth1`, a wireless interface; and `tap0`, a TUN/TAP interface. Frames that are routed to `eth0` or `eth1` are sent to a hub or switch, or to a wireless router, respectively, for delivery to their destination. In contrast, a frame that's routed to the `tap0` device is sent to whatever process on the system opened the `/dev/net/tun` file and associated that file descriptor with the `tap0` interface.

This process may do anything with the frames it receives. UML will send the frames through its own network stack, which could do almost anything with them, including delivering the data to a local pro-

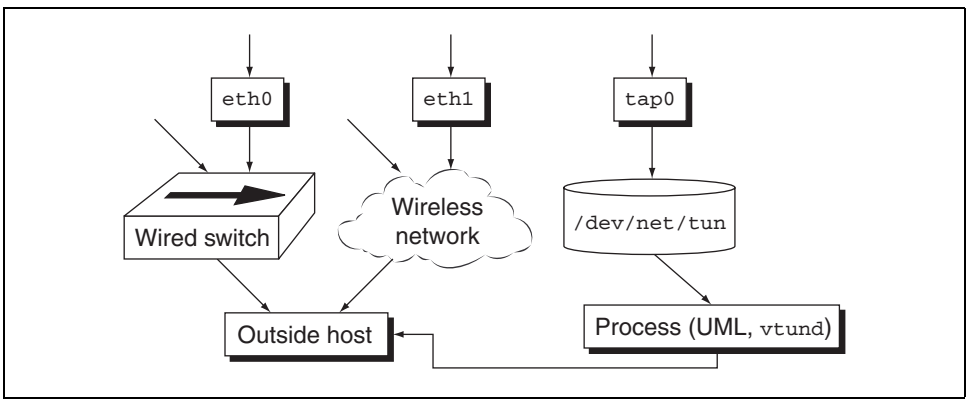


Figure 5.1 TUN/TAP provides an interface for processes to receive and transmit network frames

cess, forwarding them to another host that it's connected to, or just dropping them.

UML isn't the only process that can attach itself to a TUN/TAP interface—`vtund` is another example. `vtund` is used to construct a Virtual Private Network (VPN)—it will read frames from a TUN/TAP interface, encrypt them, and forward them to another `vtund` instance on a remote host. The remote `vtund` will decrypt the frames and inject them into the network on its host by writing them to its TUN/TAP interface. So, the TUN/TAP interface provides a general-purpose mechanism for processes to receive and transmit network traffic and do any sort of processing on it.

This is unlike the `mcast` transport we saw in the previous chapter in that frames sent to a TUN/TAP device are interpreted and routed by the host's network stack. With `mcast`, the frames were simply hunks of data to be sent from one process to another. Since frames sent to a TUN/TAP device are seen as network frames by the host, they will be routed to whatever host owns the frame's destination IP address. That could be the host itself, another machine on the local network, or a host on the Internet.

Similarly, if the IP address given to the UML is visible to the outside world (the ones we're using will be visible on the local net, but not to the Internet as a whole), people in the outside world will be able to make network connections to it. It will appear to be a perfectly normal network host.

A TUN/TAP device is very similar to a strand of Ethernet connecting the host and the UML. Each end of the strand plugs into a network device on one of the systems. As such, the device at each end needs an IP address, and the two ends need different IP addresses. The address we specified on the `uml_mconsole` command line, `192.168.0.254`, is the address of the host end of the TUN/TAP device.

The fact that a TUN/TAP interface is like a strand of Ethernet has an important implication: The TUN/TAP interface and the UML `eth0` form their own separate Ethernet broadcast domain, as shown in Figure 5.2. This means that any Ethernet broadcast protocols, such as ARP and DHCP, will be restricted to those two interfaces and the hosts they belong to. In contrast, the local Ethernet on which the host resides is a broadcast domain with many hosts on it. Without some extra work, protocols like ARP and DHCP can't cross between these two domains. This means that a UML instance can't use the DHCP server on your local network to acquire an IP address and that it can't use ARP to figure

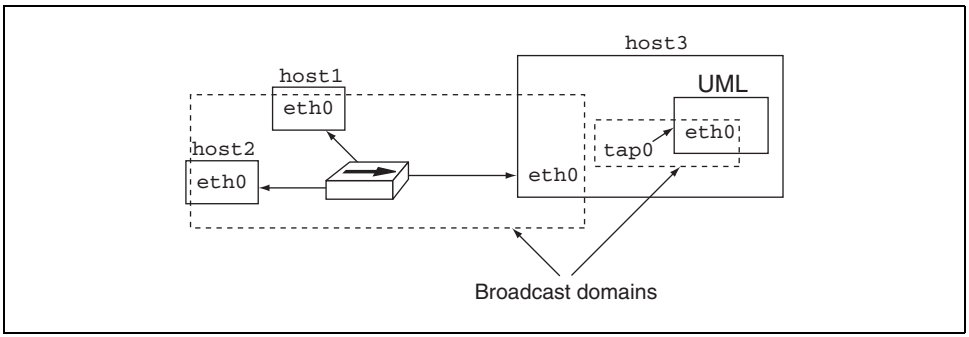


Figure 5.2 TUN/TAP interfaces form their own Ethernet networks

out the MACs of other hosts in order to communicate with them. In Chapter 7, we will discuss this problem in detail and see several methods for fixing it.

Returning to our exercise, the next step is to bring up the UML `eth0`, which is the other end of the strand, and to assign an IP address to it. Since it's a different system, it will need a different unused IP address. Here, I will use `192.168.0.253`:

```
UML# ifconfig eth0 192.168.0.253 up
* modprobe tun
* ifconfig tap0 192.168.0.254 netmask 255.255.255.255 up
* bash -c echo 1 > /proc/sys/net/ipv4/ip_forward
* route add -host 192.168.0.253 dev tap0
* bash -c echo 1 > /proc/sys/net/ipv4/conf/tap0/proxy_arp
* arp -Ds 192.168.0.253 eth1 pub
```

Again, if the output above doesn't appear on your console, run `dmesg` to see it. The UML network driver is running a helper process on the host in order to set up the host side of the network, and this output shows what commands the helper is running. We will go into much more detail in Chapter 7, but, briefly, these are making sure that TUN/TAP is available on the host, configuring the TUN/TAP interface, `tap0`, and setting up routing and proxy arp so that the instance will be visible on your local network.

At this point, we have the network running between the host and the UML instance, and you should be able to ping back and forth:

```
UML# ping 192.168.0.254
PING 192.168.0.254 (192.168.0.254): 56 data bytes
64 bytes from 192.168.0.254: icmp_seq=0 ttl=64 time=19.6 ms
```

```
--- 192.168.0.254 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 19.6/19.6/19.6 ms

host% ping 192.168.0.253
PING 192.168.0.253 (192.168.0.253) 56(84) bytes of data.
64 bytes from 192.168.0.253: icmp_seq=0 ttl=64 time=0.209 ms

--- 192.168.0.253 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.209/0.209/0.209/0.000 ms, pipe 2
```

This is the most basic level of networking. The next step is to access the outside world. We need to do two things—give our UML instance a route to the outside, and give it a name server. First, look at the UML routing table:

```
UML# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref \
    Use Iface
192.168.0.0     0.0.0.0         255.255.255.0   U     0      0   \
    0 eth0
```

This tells us we have a route to the local network, but nothing else. So, we need a default route:

```
UML# route add default gw 192.168.0.254
UML# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref \
    Use Iface
192.168.0.0     0.0.0.0         255.255.255.0   U     0      0   \
    0 eth0
0.0.0.0         192.168.0.254   0.0.0.0         UG    0      0   \
    0 eth0
```

This new route just tells the UML network to send all packets it doesn't know what to do with (in this case, those not destined for the local network) to the host `tap` interface and let it deal with them. Presumably, the host has access to the Internet and knows how to route packets to it.

Second, we need to give the UML instance an `/etc/resolv.conf`. This is normally set up by DHCP, but since we are setting up the network by hand, this must be done by hand, too. I normally just copy the host's `/etc/resolv.conf`:

```
UML# cat > /etc/resolv.conf
; generated by /sbin/dhclient-script
```

```
search user-mode-linux.org
nameserver 192.168.0.3
```

I just cut the host's `resolv.conf` from one xterm window and pasted it into another that has UML running in it. You should do something similar here. Definitely don't use my `resolv.conf` since that won't work on your network.

We need to set up one thing on the host. Since I am using the `192.168.x.x` address block, the host will need to do masquerading for it. This address block and the `10.x.x.x` one are nonroutable, so they can't be used by any machine that has direct access to the Internet. Masquerading, or Network Address Translation (NAT), on the host will solve this problem by having the host use its own address for outgoing UML network traffic.

If you are using normal, routable IP addresses, you don't need to worry about masquerading. But if you are using a nonroutable IP address for your instance, you need to NAT it. So, as root on the host, run:

```
host# iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

If `eth0` is not the device used to access the Internet, change that part of the code to the correct device. For example, if you're a dialup user with Internet access through the `ppp0` device, you would use `ppp0` instead of `eth0`.

It is common for the firewall on the host to interfere with UML's ability to communicate with any machines other than the host. To see if your host has a potentially interfering firewall, run `iptables -L` as root. If you have no firewall, you will see this:

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

Chain RH-Firewall-1-INPUT (0 references)
target     prot opt source               destination
```

If you see anything else, it's a good idea to poke a hole in the firewall for the UML instance's traffic like this, again as root:

```
host# iptables -I FORWARD -d 192.168.0.253 -j ACCEPT
host# iptables -I FORWARD -s 192.168.0.253 -j ACCEPT
```

This tells the host to allow all traffic being forwarded to and from `192.168.0.253`, the IP address that we've assigned to the UML instance, through the firewall. You might imagine that there are security implications to this, since a firewall is an important part of any network's security infrastructure, and there are. This means that any attacks aimed at the UML IP address will reach it. Whether they succeed is a different question altogether. But this is simply making a new host accessible to the outside network, including anything malicious out there.

If the host is masquerading the UML IP, that provides a degree of protection because it is invisible to the outside network, except for connections that it initiates itself. For example, if you ran a Web browser in the UML instance and loaded a Web page from a malicious Web site, the instance could be attacked by that site, through the browser. However, the instance is invisible to everything else, including worms and other malicious software scanning the network for victims. The downside of this is that it would be unusable as a server, since that would require that it be visible enough for outside clients to make connections to it.

The situation is somewhat different if you are using a real, routable IP for your UML. In this case, it is visible on the outside network and to whatever nasty things are scanning it.

In either case, you want the UML to be a full-fledged member of the network, so you need to take the same care with its security as you do with any physical machine. Make sure that the distribution you are booting in it is reasonably up to date and that you take the same precautions here as you take elsewhere.

With masquerading set up, and a hole poked in the host firewall if necessary, the UML should now be a full member of your network. We can do another simple test to make sure the UML has name service:

```
UML# host www.user-mode-linux.org
www.user-mode-linux.org A       66.59.111.166
```

Now that this works, we can check that we have full Internet access by pinging http://www.user-mode-linux.org:

```
UML# ping www.user-mode-linux.org
PING www.user-mode-linux.org (66.59.111.166): 56 data bytes
64 bytes from 66.59.111.166: icmp_seq=0 ttl=52 time=223.7 ms
64 bytes from 66.59.111.166: icmp_seq=1 ttl=52 time=38.9 ms

--- www.user-mode-linux.org ping statistics ---
3 packets transmitted, 2 packets received, 33% packet loss
round-trip min/avg/max = 38.9/131.3/223.7 ms
```

At this point, we can start playing with the UML from the outside. There should be an Apache running inside it, and you should now be able to access it at http://192.168.0.253 with your favorite browser. This is what `wget` shows from the host:

```
host% wget http://192.168.0.253
--16:40:43--  http://192.168.0.253/
          => `index.html'
Connecting to 192.168.0.253:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 4,110 [text/html]

100%[====================================>] 4,110    3.92M/s \
    ETA 00:00

16:40:43 (3.92 MB/s) - `index.html' saved [4110/4110]
```

This is the default Apache install page. You can now turn your UML instance into a Web server by installing some real content into Apache's `html` directory. You can figure out where this is by finding `httpd.conf` under either `/etc/apache` or `/etc/httpd` and looking at how it defines `DocumentRoot`:

```
UML# grep DocumentRoot /etc/apache/conf/httpd.conf
# DocumentRoot: The directory out of which you will serve your
DocumentRoot /var/www
# This should be changed to whatever you set DocumentRoot to.
#    DocumentRoot /www/docs/host.some_domain.com
```

In this case, Apache expects HTML content to be put in `/var/www`. If you put some Web pages there, they would be visible in your browser on the host.

The next thing many people like to do is remotely log in to their instance. `ssh` is the usual way to do this, and it works exactly as you'd expect:

```
host% ssh root@192.168.0.253
Password:
Last login: Mon May 23 19:56:28 2005
```

Here, I logged in as root since that's how I normally log in to a UML instances. If you create an account for yourself and put your `ssh` public key in it, you'll be able to log in to your instance as yourself, without needing a password, just as you do with any physical machine.

Obviously `ssh` in the other direction will work just as well. I continue this session by logging back in on the host as myself:

```
UML# ssh jdike@192.168.0.254
jdike@192.168.0.254's password:
host%
```

You'll want to substitute your own username for mine in the `ssh` command line.

With Web and `ssh` access working, it should be clear that the network is operating just as it does with any Linux machine. Now, let's look at another use of the network, X, and how it can be virtualized.

First, let's see that X clients running inside UML can be displayed on the host display. There are several authorization mechanisms in use by X and Xlib to ensure that X applications can connect only to displays they're allowed on. The two most common are `xhost` and Xauthority. Xauthority authorization relies on a secret (a magic cookie) stored in `~/.Xauthority` by the session manager when you log in. A client is expected to read that file; if it can, that is considered evidence that it has sufficient permissions to connect to your display. It presents the contents of the file to the X server, which checks that it really is the contents of your `.Xauthority` file.

The other mechanism is `xhost`, which is a simple access control list (ACL) naming remote machines that are allowed to connect to your display. This is less fine-grained than Xauthority since it would allow someone else logged in to a remote machine in your `xhost` list to open windows on your display. Despite this disadvantage, I will use `xhost` authorization here.

First, on whatever machine you're sitting in front of (which may not be the UML host, as it is not for me), run `xhost`:

```
X-host% xhost
access control enabled, only authorized clients can connect
```

This tells us that `xhost` access is enabled, which is good, and that no one has `xhost` access to this display. So, let's give access to the UML:

```
X-host% xhost 192.168.0.253
192.168.0.253 being added to access control list
X-host% xhost
access control enabled, only authorized clients can connect
INET:192.168.0.253
```

Your X server may be configured to not accept remote connections. You can check this by running `ps` to see the full X server command line:

```
X-host% ps uax | grep X
root      4215  1.6  2.4 59556 12672 ?       S    10:29   7:44 \
    /usr/X11R6/bin/X :0 -audit 0 -auth /var/gdm/:0.Xauth \
    -nolisten tcp vt7
```

`-nolisten tcp` causes any attempts to make X client connections fail with "connection refused." This causes the X server to accept local connections only by accepting them through a UNIX-domain socket. It is not listening to a TCP socket, and it is inaccessible to the network.

To change this, I ran `gdmsetup`, selected the Security tab, and turned off the "Always disallow TCP connections to X server" option. Other display managers, such as `xdm` or the KDE display manager, probably have similar setup applications. Then, it's necessary to restart the X server. Logging out usually suffices. You should recheck the X command line after logging back in to see that `-nolisten` has disappeared. If it hasn't, it may be necessary to kill the display manager to force it to restart.

Now, we need to set our `DISPLAY` environment variable inside UML to point at our display:

```
UML# export DISPLAY=192.168.0.254:0
```

Here I'm assuming that the machine you're using is the UML host and that you chose the IP address we assigned to the host end of the `tap` device. Any IP address associated with that host, or its name, would work equally well.

Now you should be able to run any X client on the UML and see it display on your screen. Starting with `xdpyinfo` or `xload` to see that it basically works is a good idea. What's more fun is `xterm`. This gives you a terminal on the UML without needing to log into it anymore.

Now that we have X working between the UML and the rest of the world, let's introduce a new sort of virtualization, in keeping with the spirit of this book. It is possible to virtualize X by running a process that appears to be an X server to X clients, and a client to an X server. This application is called Xnest,[1] and its name pretty well describes it. It creates a window on its own X server, which is its own root window,

1. You likely don't have Xnest (or other X packages) installed in your UML filesystem. On Fedora, Xnest comes in the `xorg-x11-Xnest` package and requires the `fonts-xorg-75dpi` package, which doesn't get pulled in automatically because of a missing dependency. You also will likely want the `xorg-x11-tools` package, which has the common X11 utilities in it.

then accepts connections from other clients, displaying their windows on this root window.

This little root window is a totally different display from the main one. There can and generally will be separate session and window managers running on it. They will be completely confined to that window and will not be able to tell that there's anything outside it.

Xnest has nothing to do with UML, but it's easy to draw parallels between them. A good one is between this little root window inside the main one, on the one hand, and the UML filesystem in a host file on the host filesystem, on the other. In both cases, the host sees something, either a file or a window, that, unbeknownst to it, contains a full universe, either the filesystem run by UML or the X session run by Xnest.

Running Xnest is simple:

```
UML# Xnest :0 &
[2] 1785
```

`:0` tells Xnest to become display 0 on the UML instance. On a physical machine with a display already attached to it, you would normally use display 1 or greater. The instance has no incumbent displays, so Xnest can use display 0.

You should now see a largish blank window on your screen. This is the new virtual display. Next, set the `DISPLAY` environment variable inside UML to use the Xnest display:

```
UML# export DISPLAY=:0
```

At this point, you can run some X clients, and you will see their windows appear within this virtual X display. You will also notice that they have no borders and can't be moved, resized, or otherwise adjusted. The thing to do now is run a window manager so that these windows become controllable. Here, I'm using `fvwm`, a lightweight, minority window manager:

```
UML# fvwm &
[5] 2067
```

Now you should see borders around the windows within the Xnest display, and you should be able to move them around just as on your normal display.

To get a bit surreal, let's run an X client on the host, displaying over the network to the UML Xnest, which is displaying back over the network to the host. First, we need to do the same X security things

inside the UML instance as we did on the host earlier. For this step, make sure there is some X client attached to the Xnest display through the entire process. The X server reinitializes itself whenever the last client disconnects, and this reinitialization includes resetting the `xhost` list. This is a problem because `xhost` itself is a client, and if it is the only one, when it disconnects, it triggers this reinitialization, which unfortunately throws out the permission changes it added.

Running `xhost` on the UML now shows:

```
UML# xhost
access control enabled, only authorized clients can connect
INET:192.168.0.253
LOCAL:
```

So, right now, the UML allows connections from local clients, either connecting over a UNIX domain socket (`LOCAL:`) or over a local TCP connection. We need to add the host from which we will be running clients and to which we will be ultimately displaying them back:

```
UML# xhost 192.168.0.254
192.168.0.254 being added to access control list
UML# xhost
access control enabled, only authorized clients can connect
INET:192.168.0.254
INET:192.168.0.253
LOCAL:
```

Now, if we go back to the host and display to the UML display 0, we should see those clients within the virtual X display:

```
host% export DISPLAY=192.168.0.253:0
host% xhost
access control enabled, only authorized clients can connect
INET:192.168.0.254
INET:192.168.0.253
LOCAL:
host%  xterm &
[1] 7535
host% fvwm &
[2] 7654
```

Now, as expected, we have a host window manager and xterm window displaying within the virtual X display running on the UML. Figure 5.3 shows a partial screenshot of a display with an Xnest running from a UML instance. I have `xhost` set up as described earlier, and the xterm window inside the Xnest is running over the network from a

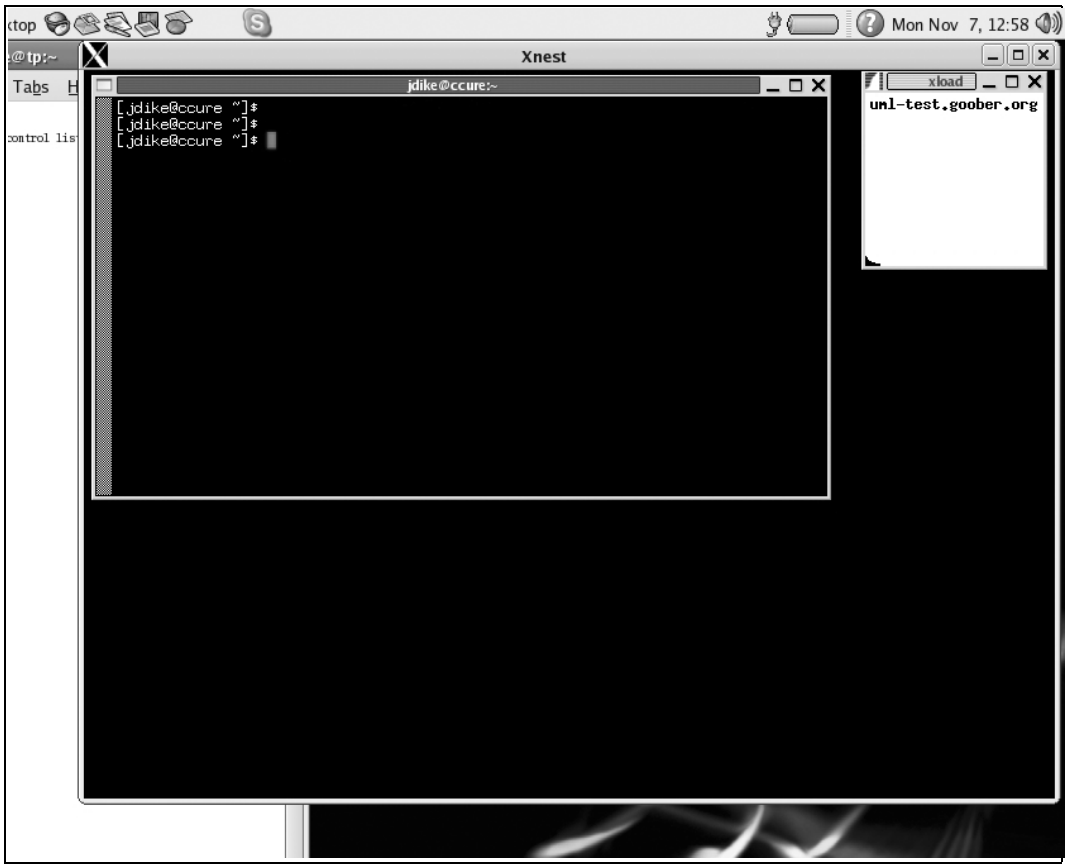


Figure 5.3 UML running Xnest

third host. There is also a local xload window and window manager running inside the Xnest.

Now, strictly speaking, none of this Xnest stuff required UML. Everything we just did could have been done on the host, with Xnest providing a second display that happens to be shown on part of the first. However, it is a nice example of providing a virtual machine with a new piece of virtualized hardware that behaves just like the equivalent piece of physical hardware. It also shows another instance of constructing this virtual device with part of the equivalent physical hardware. As I pointed out earlier, the analogy of Xnest with other UML devices is more than skin deep. Xnest does have a role to play with UML, even if it was created independent of and earlier than UML, and even if it is rarely required for day-to-day work.

CHAPTER 6

UML Filesystem Management

In this chapter, we will talk about filesystems from the perspectives of both the UML instance and the host. There are a few different ways to store files on the host so they can be mounted as a filesystem inside a UML instance. The one we've already seen, and the most popular, is to use a block device within a host file. The others involve mounting a host directory hierarchy inside the instance. This has the advantage that the UML files are visible on the host and can be managed from there. The usefulness of this becomes obvious when you have a UML user who lost the root password and needs it reset. It's much easier to do that when the UML instance's `/etc/passwd` is a normal file on the host than when the `/etc/passwd` is hidden inside a filesystem image in a large host file.

MOUNTING HOST DIRECTORIES WITHIN A UML

There are two ways to mount a host directory as a UML directory—`hostfs` and `humfs`. `hostfs` is the older and more limited method, but it does have the advantage of greater convenience. Both are virtual file-

systems, in the sense that they are not stored within a UML block device. You can think of them as nondevice filesystems whose data is maintained without benefit of a storage device that's known to UML. In many cases, the data is simply stored inside the kernel. If you look at `/proc/filesystems` on any modern Linux system, UML or physical, you will see a great number of these:

```
host% cat /proc/filesystems
nodev   sysfs
nodev   rootfs
nodev   bdev
nodev   proc
nodev   sockfs
nodev   binfmt_misc
nodev   debugfs
nodev   usbfs
nodev   pipefs
nodev   futexfs
nodev   tmpfs
nodev   eventpollfs
nodev   devpts
        ext2
nodev   ramfs
nodev   hugetlbfs
        iso9660
nodev   mqueue
nodev   selinuxfs
        ext3
nodev   rpc_pipefs
nodev   autofs
```

All of the `nodev` entries are virtual filesystems. Most of these make internal kernel information available as a filesystem. Probably the most familiar example is `proc`, which is normally mounted on `/proc`. This filesystem makes internal kernel variables and data structures visible as files. Most of the others do, as well, with the exception of `tmpfs`. This is a normal filesystem, in the sense that it can be mounted and arbitrary files created within it. Those files are temporary, disappearing when the filesystem is unmounted, rather than being stored permanently on a disk, and written to the system's swap when memory is tight, rather than to a dedicated disk partition. Figure 6.1 illustrates the differences between the various kinds of Linux filesystems.

`hostfs` and `humfs` are similar to these in that the filesystem data seems to be fabricated from within the kernel, but different in that the data is permanently stored. They are conceptually most similar to a network filesystem, such as NFS. In both cases, the data is stored outside

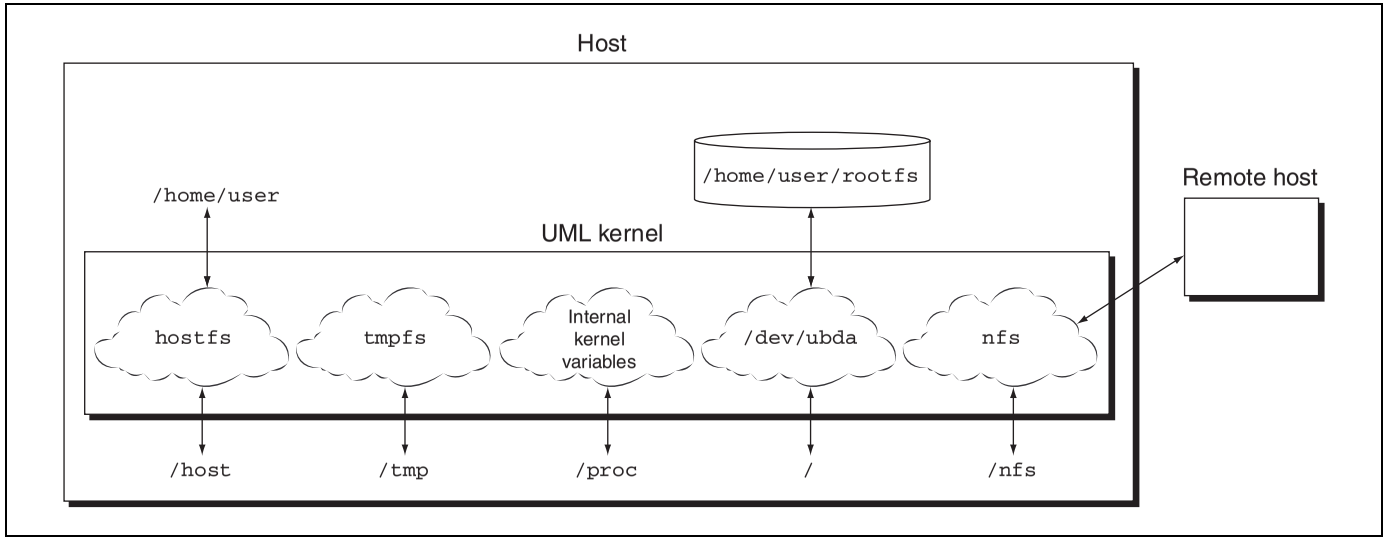


Figure 6.1 The various types of filesystems available in UML. `/` is a traditional disk-based filesystem contained in the UML device `/dev/ubda`, which itself is contained in a file on the host. `/proc` and `/tmp` are virtual filesystems whose data is contained entirely within the UML kernel. `/proc` exports internal kernel data structures, so its data is always within the kernel. `/tmp` is a `tmpfs` mount, which looks like a normal filesystem, but its data is stored in the kernel's filesystem cache and swapped to the UML instance's swap space if necessary. `/nfs` is also virtual in the sense that the data contained in the filesystem isn't stored on this system. It resides on a remote system and remote procedure calls (RPCs) are made in order to access it. `/host` is another sort of virtual filesystem, except that its data comes from a directory hierarchy on the host. This is fairly similar to an `nfs` mount, except that the remote files are accessed using system calls to the host rather than a network protocol.

the machine and transparently made available by a filesystem that knows how to access it.

With a network filesystem, file accesses are translated into network requests to the server, which sends data and status back. With `hostfs` and `humfs`, file accesses are translated into file accesses on the host. You can think of this as a one-to-one translation of requests—a read, write, or `mkdir` within UML translates directly into a read, write, or `mkdir` to the host. This is actually not true in the most literal sense. An operation such as `mkdir` within one of these filesystems must create a directory on the host; therefore, it must translate into a `mkdir` there, but won't necessarily do so immediately. Because of caching with the filesystem, the operation may not happen until a long time later. Operations such as a read or write may not translate into a host read or write at all. They may, in fact translate into an `mmap` followed by directly reading or writing memory. And in any case, the lengths of the read and write operations will certainly change when they reach the host. Linux filesystem operations typically have page granularity—the minimum I/O size is a machine page, 4K on most extant systems. For example, a sequence of 1-byte reads will be converted into a single page-length read to the host followed by simply passing out bytes one at a time from the buffer into which that page was read.

So, while it is conceptually true that `hostfs` and `humfs` operations correspond one-to-one to host operations, the reality is somewhat different. This difference will become relevant later in this chapter when we look at simultaneous access to data from a UML and the host, or from two UMLs.

hostfs

`hostfs` is the older and simpler of the two ways to mount a host directory as a UML directory. It uses the most obvious mapping of UML file operations to host operations in order to provide access to the host files. This is complicated only by some technical aspects, such as making use of the UML page cache. This simplicity results in a number of limitations, which we will see shortly and which I will use to motivate `humfs`.

So, let's get a UML instance and make a hostfs mount inside it:

```
UML# mount none /mnt -t hostfs
```

Now we have a new filesystem mounted on `/mnt`:

```
UML# mount
/dev/ubd0 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=4,mode=620)
shm on /dev/shm type tmpfs (rw)
none on /mnt type hostfs (rw)
```

Its contents show that it looks a lot like the host's root filesystem:

```
UML# ls /mnt
bin   etc     lib         media  opt   sbin     sys    usr
boot  home    lib64       misc   proc  selinux  tmp    var
dev   initrd  lost+found  mnt    root  srv      tools
```

You can do the same `ls` on the host's `/` to verify this. Basically, we have mounted the host's root on the UML instance's `/mnt`, creating a completely normal Linux filesystem within the UML. For getting access to files on the host within UML, this is very convenient. You can do anything within this filesystem that you can do with a disk-based filesystem, with some restrictions that we will talk about later.

By default, when you make a `hostfs` mount, you get the host's root filesystem. This isn't always desirable, so there is an option to mount a different host directory:

```
UML# mkdir /mnt-home
UML# mount none /mnt-home/ -t hostfs -o /home
UML# ls /mnt-home/
jdike  lost+found
```

The `-o` option specifies the host directory to mount. From that mount point, it is impossible to access any files outside that directory. In our case, the `/mnt-home` mount point gives us access to the host's `/home`, but, from there, we can't access anything outside of that. The obvious trick of using `..` to try to access files outside of `/home` won't work because it's the UML that will interpret the `..`, not the host. Trying to "dotdot" your way out of this will get you to the UML instance's `/`, not the host's `/`.

Using `-o` is at the option of the user within the instance. Many times, the host administrator wants all `hostfs` mounts confined to a host subdirectory and makes it impossible to access the host's `/`. There is a command-line option to UML to allow this, `hostfs=/path/to/UML/jail`. With this enabled, `hostfs` mounts within the UML will be

restricted to the specified host subdirectory. If the UML user does a mount specifying a mount path with `-o`, that path will be appended to the directory on the command line. So, `-o` can be used to mount subdirectories of whatever directory the UML's `hostfs` has been confined to, but can't be used to escape it.

Now, let's create a file within the host mount:

```
UML# touch /mnt/tmp/uml-file
UML# ls -l /mnt/tmp/uml-file
-rw-r--r--  1 500 500 0 Jun 10 13:02 /mnt/tmp/uml-file
```

The ownerships on this new file are somewhat unexpected. We are root inside the UML, and thus expect that any new files we create will be owned by root. However, we are creating files on the host, and the host is responsible for the file, including its ownerships. The UML instance is owned by user ID (UID) 500, so from its point of view, a process owned by UID 500 created a file in `/tmp`. It's perfectly natural that it would end up being owned by that UID. The host doesn't know or care that the process contains another Linux kernel that would like that file to be owned by root.

This seems perfectly reasonable and innocent, but it has a number of consequences that make `hostfs` unusable for a number of purposes. To demonstrate this, let's become a different, unprivileged user inside UML and see how `hostfs` behaves:

```
UML# su user
UML% cd /mnt/tmp
UML% echo foo > x
UML% ls -l x
-rw-r--r--  1 500 500 4 Jun 10 14:31 x
UML% echo bar >> x
sh: x: Permission denied
UML% rm x
rm: remove write-protected regular file `x'? y
rm: cannot remove `x': Operation not permitted
UML% chmod 777 x
chmod: changing permissions of `x': Operation not permitted
```

Here we see a number of unexpected permission problems arising from the ownership of the new file. We created a file in the host's `/tmp` and found that we couldn't subsequently append to it, remove it, or change its permissions.

It is created with the owner UID 500 on the host and is writable by that UID. However, I became `user`, with UID 1001, inside the UML instance, so my attempts to modify the file don't even make it past the

UML's permission checking. When the file was created on the host, it was given its ownership and permissions by the host. `hostfs` shows those permissions, rather than the ones the UML instance provided, because they are more "real."

The ownership and permissions are interpreted locally by the UML when seeing whether a file operation should succeed. The fact that the file ownerships are set by the host to something different from what the UML expects can cause files to be unmodifiable by their owner within UML.

This isn't a problem for the root user within UML because the superuser doesn't undergo the same permission checks as a normal user, so the permission checks occur on the host.

However, this issue does make it impossible for multiple users within the UML to use `hostfs`. In fact, only root within the UML can realistically use it. The only way for a normal UML user to use `hostfs` is for its UID to match the host UID that the UML is running as. So, if `user` within UML had UID 500 (matching the UML instance's UID on the host), the previous example would have been more successful.

Let's look at another problem, in which root within the UML doesn't have permission to do some things that it should be able to do:

```
UML# mknod ubda b 0 98
mknod: `ubda': Operation not permitted
```

Here, creating a device node for `ubda` doesn't work, even for root. Again, the reason is that the operation is forwarded to the host, where it is attempted as the nonroot UML user, and fails because this operation requires root privileges. You will see similar problems with creating a couple of other types of files.

If you experiment long enough with `hostfs`, you will discover other problems, such as accessing UNIX sockets. If the `hostfs` mount contains sockets, they were created by processes on the host. When one is opened on the host, it can be used to communicate with the process that created it. However, they are visible within a `hostfs` mount, but a UML process opening one will fail to communicate with anything. The UML kernel, not the host kernel, will interpret the open request and attempt to find the process that created it. Within the UML kernel, this will fail because there is no such process.

Creating a directory on the host with a UML root filesystem in it, and booting from it, is also problematic. The filesystem, by and large, should be owned by root, and it won't be. All of the files are owned by whoever created them on the host. At this writing, there is a kludge in

the `hostfs` code that changes (internally to the UML kernel) the ownerships of these files to root when the `hostfs` filesystem is the UML root filesystem. This makes booting from `hostfs` work, more or less, but all the problems described above are still there. Other kernel developers have objected to this ownership changing, and this kludge likely won't be available much longer. When this "feature" does disappear, booting from a `hostfs` root filesystem likely won't work anymore.

I've spent a good amount of time describing the deficiencies of `hostfs`, but I'd like to point out that, for a common use case, `hostfs` is exactly what you want. If you have a private UML instance, are logged in to it as root, and want access to your own files on the host, `hostfs` is perfect. The filesystem semantics will be exactly what you expect, and no prior host setup is needed. Just run the `hostfs` mount command, and you have all of your files available.

Most of the problems with `hostfs` that I've described stem from the fact that all `hostfs` file operations go through both the UML's and the host's permission checking. This is because both systems look at the same data, the file metadata on the host, in order to decide what's allowed and what's not.

UNIX domain sockets and named pipes are sort of a reflection of a process within the filesystem—there is supposed to be a process at the other end of it. When the filesystem (including the sockets) is exported to another system, whether a UML instance with a `hostfs` mount or another system with an NFS mount, the process isn't present on the other system. In this case, the file doesn't have the meaning it does on its home system.

humfs

We can fix these problems by making sure we see, inside UML, distinct file ownerships, permissions, and types from the host. To achieve this, UML can store these in a separate place, freeing itself from the host's permission checks. This is what `humfs` does. The actual file data is stored in exactly the same way that `hostfs` does—in a directory hierarchy on the host. However, permissions information is stored separately, by default, in a parallel directory hierarchy.

For example, here are the data and metadata for a file stored in this way:

```
host% ls -l data/usr/bin/ls
```

```
-rwxr-x--x  1 jdike jdike 201642 May  1 10:01 data/usr/bin/ls
host% ls -l file_metadata/usr/bin/ls
-rw-r--r--  1 jdike jdike 8 Jun 10 18:04 file_metadata/usr/bin/ls
host% cat file_metadata/usr/bin/ls
493 0 0
```

The actual `ls` binary is stored in `data/usr/bin/ls`, while its ownership and permissions are stored in `file_metadata/usr/bin/ls`. Notice that the permissions on the binary are wide open for the file's owner. This, in effect, disables permission checking on the host, allowing UML's ideas about what's allowed and what's not to prevail.

Next, notice the contents of the metadata file. For a normal file, such as `/usr/bin/ls`, the permissions and ownerships are stored here. In the last line of the output, 493 is the decimal equivalent of 0755, and the zeros are UID root and group ID (GID) root.

We can see this by looking at this file inside UML:

```
UML# ls -l usr/bin/ls
-rwxr-xr-x  1 root root 201642 May  1 10:01 usr/bin/ls
```

The `humfs` filesystem has taken the file size and date from `data/usr/bin/ls` and merged the ownership and permission information from `file_metadata/usr/bin/ls`.

By storing this metadata as the contents of a file on the host, UML may modify it in any way it sees fit. We can go through the list of `hostfs` problems I described earlier and see why this approach fixes them all.

In the case of a new file having unexpected ownerships, we can see that this just doesn't happen in `humfs`. The data file's ownership will, in fact, be determined by the UID and GID of the UML process, but this doesn't matter since the ownerships you will see inside UML will be determined by the contents of the `file_metadata` file.

So, you will be able to create a file on a `humfs` mount and do anything with it, such as append to it, remove it, or change permissions.

Now, let's try to make a block device:

```
UML# mknod ubda b 98 0
UML# ls -l ubda
brw-r--r--  2 root root 98, 0 Jun 10 18:46 ubda
```

This works, and it looks as we would expect. To see why, let's look at what occurred on the host:

```
host% ls -l data/tmp/ubda
-rwxrw-rw-  1 jdike jdike 0 Jun 10 18:46 data/tmp/ubda
```

```
host% ls -l file_metadata/tmp/ubda
-rw-r--r--  1 jdike jdike 15 Jun 10 18:46 file_metadata/tmp/ubda
host% cat file_metadata/tmp/ubda
420 0 0 b 98 0
```

The file is empty, just a token to let the UML filesystem know a file is there. Almost all of the device's data is in the metadata file. The first three elements are the same permissions and ownership information that we saw earlier. The rest, which don't appear for normal files, describe the type of file, namely, a block device with major number 98 and minor number 0.

The host definitely won't recognize this as a block device, which is why this works. Creating a device requires root privileges, so `hostfs` can't create one unless the UML is run by root. Under `humfs`, creating a device is simply a matter of creating this new file with contents that describe the device.

It is apparent that the host socket and named pipe problem can't happen on this filesystem. Everything in this directory on the host is a normal file or directory. Host sockets and named pipes just don't exist. If a UML process makes a UNIX domain socket or a named pipe, that will cause the file's type to appear in the metadata file.

Along with these advantages, `humfs` has one disadvantage: It needs to be set up beforehand. You can't just take an arbitrary host subdirectory and mount it as a `humfs` filesystem. So, `humfs` is not really useful for quick access to your files on the host.

In order to set up `humfs`, you need to decide what's going to be in your `humfs` mount, create an empty directory, copy the files to the `data` subdirectory, and run a script that will create the metadata. As a quick example, here's how to create a `humfs` version of your host's `/bin`.

```
host% mkdir humfs-test
host% cd humfs-test
host# cp -a /bin data
host# perl ..humfsify.pl jdike jdike 100M
host% ls -al
total 24
drwxrw-rw-   5 jdike jdike 4096 Jun 10 19:40 .
drwxrw-r--  16 jdike jdike 4096 Jun 10 19:40 ..
drwxrwxrwx   2 jdike jdike 4096 May 23 12:12 data
drwxr-xr-x   2 jdike jdike 4096 Jun 10 19:40 dir_metadata
drwxr-xr-x   2 jdike jdike 4096 Jun 10 19:40 file_metadata
-rw-r--r--   1 jdike jdike   58 Jun 10 19:40 superblock
```

Two of the commands, the creation of the data subdirectory and the running of `humfsify`, have to be run as root. The copying of the

directory needs to preserve file ownerships so that `humfsify` can record them in the metadata, and `humfsify` needs to change those ownerships so that you own all the files.

We now have two metadata directories, one for files and one for directories, and a `superblock` file. This file contains information about the filesystem as a whole, rather like the superblock on a disk-based filesystem:

```
host% cat superblock
version 2
metadata shadow_fs
used 6877184
total 104857600
```

This tells the UML filesystem:

- What version of `humfs` it is dealing with
- What metadata format is being used
- How much disk space is used
- How much total disk space is available

The `shadow_fs` metadata format describes the parallel metadata directories. There are some other possibilities, which will be described later in this section. The total disk space amount is simply the number given to `humfsify`. This number is used by the filesystem within UML to enforce the limit on disk consumption. Quotas on the host can be used, but they are not necessary.

You may have noticed that it would be particularly easy to change the amount of disk space in this filesystem. Simply changing the `total` field by editing this file would seem to do the trick, and it does. At this writing, this ability is not implemented, but it is simple enough and easy enough to do that it will be implemented at some point.

Now, having created the `humfs` directory, we can mount it within the UML:

```
UML# mkdir /mnt-test
UML# mount none /mnt-test -t humfs -o \
    path=/home/jdike/linux/humfs-test
UML# cd /mnt-test
```

If you do an `ls` at this point, you'll see your copy of the host's `/bin`.

Note that the `mount` command is very similar to the `hostfs` `mount` command. It's a virtual filesystem, so we're telling it to mount

none since there is no block device associated with it, and we specify the filesystem type and the host mount point. In the case of humfs, specifying the host mount point is mandatory because it must be prepared ahead of time. humfs is passed the root of the humfs tree, which is the directory in which the data and metadata directories were created.

You can now do all the things that didn't work under humfs and see that they do work here. humfs works as expected in all cases, with no interference from the host's permission checking. So, humfs is usable as a UML root filesystem, whereas hostfs can be used only with some trickery.

Now I'll cover some aspects of humfs that I didn't explain earlier. First, version 2 of humfs was created because version 1 had a bug, and fixing that bug led to the separate file_metadata and dir_metadata directories. As we've seen, the metadata files for files are straightforward. Directories have ownerships and permissions and need metadata files, but they introduce problems in some corner cases.

The initial shadowfs design required a file called metadata in each directory in the metadata tree that would hold the ownerships and permissions for the parent directory. Of course, each file in the original directory would have a file in the metadata tree with the same name. But I missed this case: What metadata file should be used for a file called metadata? Both the file and the parent directory would want to use the same metadata file, metadata.

Another problem occurs with a subdirectory called metadata. In this case, the metadata file will want to be both a directory (because the metadata directory structure is identical to the data directory structure) and a file (because the parent directory will want to put its metadata there.

The solution I chose was to separate the file and directory metadata information from each other. With them in separate directory trees, the first collision I described doesn't exist. However, the second does. The solution to that is to allow the metadata directory to be created, and rename the parent directory's metadata file. It turns out that it can be renamed to anything that doesn't collide with a subdirectory. The reason is that in the dir_metadata tree, there will be only one normal file in each directory. If metadata is a directory, the humfs filesystem will need to scan the directory for a normal file, and that will be the metadata file for the parent directory.

The next question is this: Why do we specify the metadata format in the superblock file? When I first introduced humfs, with the version 1 shadow_fs format, there were a bunch of suggestions for alter-

nate formats. They generally have advantages and disadvantages compared to the `shadow_fs` format, and I thought it would be interesting to support some of them and let system administrators choose among them.

These proposals came in two classes—those that preserved some sort of shadow metadata directory hierarchy, and those that put the metadata in some sort of database. An interesting example of the first class was to make all of the metadata files symbolic links, rather than normal files, and store the metadata in the link target. This would make them dangling links, as the targets would not exist, but it would allow somewhat more efficient reading of the metadata.

Reading a file requires three system calls: an open, a read, and a close. Reading the target of a symbolic link requires one—a readlink. Against this slight performance gain, there would be some loss of manageability, as system administrators and their tools expect to read contents of files, not targets of symbolic links.

The second class of proposals, storing metadata in databases of various sorts, is also interesting. Depending on the database, it could allow for more efficient retrieval of metadata, which is nice. However, what makes it more interesting to me is that the database could be used on the host to do queries much more quickly than with a normal filesystem. The host administrator could ask questions about what files had been modified recently or what files are `setuid` root and could get answers very quickly, without having to search the entire filesystem.

Even more interesting would be the ability to import this capability into the UML, where the UML administrator, who probably cares about the answers more than the host administrator does, could ask these questions. I'm planning to allow this through yet another filesystem, which would make a database look like a filesystem. The UML admin would mount this filesystem inside the UML and query the database underneath it this like:

```
UML# cat /sqlfs/"select name from root_fs where setuid = 1"
/usr/bin/newgrp
/usr/bin/traceroute6
/usr/bin/chfn
/usr/bin/chsh
/usr/bin/gpasswd
/usr/bin/passwd
```

The "file" associated with a query would contain the results of that query. In the example above, we searched the database for all `setuid` files, and the results came back as the contents of a file.

With `humfs`, only the file metadata would be indexed in the database. It is possible to do the same thing with the contents of files. This would take a different framework than that which enables `humfs` but is still not difficult. It would be possible to load a UML filesystem into a database, be it SQL, Glimpse, or Google, and have that database imported into UML as a bootable filesystem. Queries to the database would be provided by a separate filesystem, as described earlier. In this way, UML users would have access to their files through any database the host administrator is willing to provide.

An alternate use of this is to load some portion of your data, such as your mail, into such a database-backed filesystem. These directories and files will remain accessible in the normal way, but the database interface to them will allow you to search the file contents more quickly than is possible with utilities such as `find` and `grep`. For example, loading your mail directory into a filesystem indexed by something like Glimpse would give you a very fast way to search your mail. It would still be a normal Linux filesystem, so mail clients and the like would still work on it, and the index would be kept up to date constantly since the filesystem sees all changes and feeds them into the index. This means that you could search for something soon after it is created (and find it) rather than waiting for the next indexing run, which would probably be in the wee hours, making the change visible in the index the following day.

HOST ACCESS TO UML FILESYSTEMS

To round out this discussion of UML filesystem options, we need to take another look at the standard `ubd` block device. Both `humfs` and `hostfs` allow easy access on the host to the UML's file since both mount host directory hierarchies into UML. With `hostfs`, these files can be manipulated directly.

With `humfs`, some knowledge of the directory layout is necessary. Changing the contents of a file is done in the expected way, while changing metadata—permissions, ownerships, and file type in the case of devices, sockets, and named pipes—requires that the contents of the metadata file be changed, rather than simply using the usual tools

such as chmod and chown. In the case of a database representation of the metadata, this would require a database update.

A ubd device allows even less convenient access to the UML's files, as a filesystem image is a rather opaque storage medium. However, loop-mounting the image on the host provides hostfs-like access to the files. This works as follows:

```
host# mount uml-root-fs host-mount-point -o loop
```

After this, the UML filesystem is available as a normal directory hierarchy under host-mount-point. However, the UML should not be running at this point, since there is no guarantee that the filesystem is consistent. There may be data cached inside the UML that hasn't been flushed out to the filesystem image and that is needed in order for the filesystem to be consistent. Second, any sort of mount requires root privileges. So, while a loopback-mount makes a ubd device look like a hostfs directory, it is necessary to be root on the host and, normally, for the UML to not be running. In the next section, we'll look at a way around this last restriction and describe a method for getting a consistent backup from a running UML instance.

This consistency problem is also present with hostfs and humfs. By default, they cache changes to their files inside the UML page cache, writing them out later. If you change a hostfs or humfs file, you probably won't see the change on the host immediately. When hostfs is used as a file transfer mechanism between the UML instance and the host, this can be a problem. It can be solved by mounting the filesystem synchronously, so that all changes are written immediately to the host. This is most easily done by adding sync to the options field in the UML /etc/fstab file:

```
none            /host               hostfs  sync  0 0
```

If the filesystem is already mounted, it can be remounted to be synchronous without disturbing anything that might already be using it:

```
mount -o remount,sync /host
```

Doing this will decrease the performance of the filesystem, as the amount of I/O that it does will be greatly increased.

hostfs is more likely to be used as a file transfer mechanism between the UML instance and the host since the humfs directory structure doesn't lend itself as well to being used in this way. A host

directory can also be shared with `hostfs` between multiple UML instances without problems because the filesystem consistency is maintained by the host. Delays in seeing file updates will happen with a `hostfs` mount shared by multiple UML instances just as they happen when the mount is shared by the host and UML instance. To avoid this, the `hostfs` directories have to be mounted synchronously by all of the UML instances.

The `hostfs` directory does not have to be mounted synchronously on the host—changes made by the host are immediately visible.

MAKING BACKUPS

The final point of comparison between `ubd` devices, `hostfs`, and `humfs` is how to back them up on the host. `hostfs` should normally be used only for access to host files that don't form a UML filesystem, so the question of specifically backing them up shouldn't arise. However, if a directory on the host is expected to be mounted as a `hostfs` mount, backing it up on the host can be done normally, using any backup utility desired. The consistency of the hierarchy is guaranteed by the host since it's a normal host filesystem. Any changes that are still cached inside the UML will obviously not be captured by a backup, but this won't affect the consistency of a backup.

`humfs` is a bit more difficult. Since file metadata is stored separately from the file, a straightforward backup on the host could possibly be inconsistent if the filesystem is active within the UML. For example, when a `humfs` file is deleted, both the data file and the metadata file (in the case of the `shadow_fs` metadata format) must be deleted. If the backup is taken between these two deletions, it will be inconsistent, as it will show a partially deleted file. The obvious way around this problem is to ensure that the `humfs` filesystem isn't mounted at the time of the backup, either by shutting down the UML or by having it unmount the filesystem. This last option might be difficult if the `humfs` filesystem is the UML's root.

However, there is a neat trick to get around this problem: a facility within Linux called Magic SysRq. On a physical system, this involves using the SysRq key in combination with some other key in order to get the kernel to do one of a set of operations. This is normally used in emergencies, to exercise some degree of control over the machine when nothing else works. One of the functions provided by the facility is to

flush all filesystem changes to stable storage. On a physical machine, this would normally be done prior to crashing it by turning off the power or hitting the reset button. Flushing out dirty data ensures that the filesystems will be in good shape when the system is rebooted.

In addition to this, UML's `mconsole` facility provides the ability to stop the virtual machine, so that it only listens to `mconsole` requests, and later continue it.

The trick involves these three operations:

```
host% uml_mconsole umid stop
OK
host% uml_mconsole umid sysrq s
OK
host% uml_mconsole umid go
OK
```

Here, we stop the UML, force it to sync all data to disk (`sysrq s`), and restart it.

When this is being done as part of a backup procedure, the actual backup would take place between the `sysrq s` command and continuing the UML.

Finally, backing up `ubd` filesystem images involves the same considerations as `humfs` filesystems. Without taking care, you may back up an inconsistent image, and booting a UML on it may not work. However, in lieu of shutting down the UML, the `mconsole` trick I just described for a `humfs` filesystem will work just as well for a `ubd` image. If the `ubd` filesystem uses a COW layer, this can be extremely fast. In this case, only the COW file needs to be copied, and if it is largely empty, and the backup tool is aware of sparse files, a multigigabyte COW file can be copied to a safe place in a few seconds.

EXTENDING FILESYSTEMS

Sometimes you might set up a filesystem for a UML instance that subsequently turns out to be too small. For the different types of filesystems we have covered in this chapter, there are different options.

By default, the space available in a `hostfs` mount is the same as in the host filesystem in which the data resides. Increasing this requires either deleting files to increase the amount of free space or increasing the size of the filesystem somehow. If the filesystem resides on a logical volume, a free disk partition can be added to the corresponding volume group. Otherwise, you will need to move the `hostfs`

data to a different partition or repartition the disk to increase the size of the existing partition.

Another option is to control the space consumption on `hostfs` mounts by using quotas on the host. By running different UML instances as different UIDs and assigning disk quotas to those UIDs, you can control the disk consumption independently of the space that's actually available on the host filesystem. In this case, increasing the space available to a UML instance on a `hostfs` mount is a matter of adjusting its disk quota on the host.

As we saw earlier, you can change the size of a `humfs` mount by changing the value on the `total` line in the `superblock` file.

The situation with a `ubd` block device is more complicated. Increasing the size of the host file is simple:

```
host% dd if=/dev/zero of=root_fs bs=1024 \
    seek=$[ 2 * 1024 * 1024 ] count=1
```

This increases the size of the `root_fs` file to 2GB. A more complicated problem is making that extra space available within the UML filesystem. Some but not all filesystems support being resized without making a backup and recreating the filesystem from scratch. Fewer support being resized without unmounting the filesystem. One that does is `ext2` (and `ext3` since it has a nearly identical on-disk format). By default, `ext2online` resizes the filesystem to fill the disk that it resides on, which is what you almost always want:

```
UML# ext2online /dev/ubda
```

You can also specify the mount point rather than the block device, which may be more intuitive and less error prone:

```
UML# ext2online /
```

With other filesystems, you may have to unmount the filesystem before resizing it to fill the device. If the filesystem in question is the UML instance's root filesystem, you will likely need to halt the instance and resize the filesystem on the host.

For filesystems that don't support resizing at all, you have to copy the data to someplace else and recreate the filesystem from scratch using `mkfs`. Then you can copy your data back into it. Again, if this is the root filesystem of the UML instance, you will need to shut it down and then recreate the filesystem on the host.

WHEN TO USE WHAT

Now that you have learned about these three mechanisms for providing filesystem data to a UML, the question remains: Under what circumstances should you use each of them? The answer is fairly easy for `hostfs`—normally, it should be used only for access to host files that belong to the user owning the UML or to files that are available read-only. In the first case, the user should be logged in to the UML as root, and there should be no other UML users accessing the `hostfs` mount. In the second, the read-only restriction avoids all of the permission and ownership issues with `hostfs`.

`humfs` hierarchies and `ubd` images can be used to provide general-purpose filesystems, including root filesystems. `humfs` provides easier access to the UML files, although some care is needed when changing those files in order to ensure that the file metadata is updated properly.

There are also some potential efficiency advantages with both `humfs` and `ubd` devices. An issue with host memory consumption is that both the host and UML will generally cache file data separately. As a result, the host's memory will contain multiple copies of UML file data, one in the host's page cache and one for each UML that has read the data.

`ubd` devices can avoid this double caching by using `O_DIRECT` I/O on 2.6 hosts. `O_DIRECT` avoids the use of the host page cache, so the only copies of the data will be in the UMLs that have read it. In order to truly minimize host memory consumption, this should be used only for data that's private to the UML, such as a private filesystem image or a COW file. For a COW file, the memory savings obtained by avoiding the double caching are probably outweighed by the duplicate caching of the backing file data in the UMLs that are sharing it.

For shared data, `humfs` avoids the double caching by mapping the data from the host. The data is cached on the host, but mapping it provides the UML with the same page of memory that's in the host page cache. Taking advantage of this would require a form of COW for `humfs`, which currently doesn't exist. A file-level form of COW is possible and may exist by the time you read this. With this, a `humfs` equivalent of a backing file, in the form of a read-only host directory hierarchy, would be mapped into the UMLs that share it. They would all share the same memory, so there would be only one copy of it in the host's memory.

In short, both `ubd` devices and `humfs` directories have a place in a well-run UML installation. The use of one or the other should be driven by the importance of convenient host access to the UML filesystem, the ease and speed of making backups of the data, and avoidance of excessive host memory consumption.

CHAPTER 7

UML Networking in Depth

MANUALLY SETTING UP NETWORKING

TUN/TAP with Routing

In earlier chapters we briefly looked at how to put a UML on the network. Now we will go into this area in some depth. The most involved part is setting up the host when the UML will be given access to the physical network. The host is responsible for transmitting packets between the network and the UML, so correct setup is essential for a working UML network.

There are two different methods for configuring the host to allow a UML access to the outside world: routing packets to and from the UML and bridging the host side of the virtual interface to the physical Ethernet device. First we will use the former method, which is more complicated. We will start with a completely unconfigured host, on which a UML will fail to get access to the network, and, step by step, we'll debug it until the UML is a fully functional network node. This will provide a good understanding of exactly what needs to be done to the host and will allow you to adapt it to your own needs. The step-by-step

debugging will also show you how to debug connectivity problems that you encounter on your own.

Later in this chapter, we will cover the second method, bridging. It is simpler but has disadvantages and pitfalls of its own.

Configuring a TUN/TAP Device

We are going to use the same host mechanism, TUN/TAP, as before. Since we are doing this entirely by hand, we need to provide a TUN/TAP device for the UML to attach to. This is done with the `tunctl` utility:

```
host% tunctl
Failed to open '/dev/net/tun' : Permission denied
```

This is the first of many roadblocks we will encounter and overcome. In this case, we can't manipulate TUN/TAP devices as a normal user because the permissions on the control device are too restrictive:

```
host% ls -l /dev/net/tun
crw-------  1 root root 10, 200 Jul 30 07:36 /dev/net/tun
```

I am going to do something that's a bit risky from a security standpoint—change the permissions to allow any user to create TUN/TAP devices:

```
host# chmod 666 /dev/net/tun
```

When the TUN/TAP control device is open like this, any user can create an interface and use it to inject arbitrary packets into the host networking system. This sounds nasty, but the actual practicality of an attack is doubtful. When you can construct packets and get the host to route them, you can do things like fake name server, DHCP, or Web responses to client requests. You could also take over an existing connection by faking packets from one of the parties. However, faking a server response requires knowing there was a request from a client and what its contents were. This is difficult because you have set yourself up to create packets, not receive them. Receiving packets still requires help from root.

Faking a server response without knowing whether there was an appropriate request requires guessing and spraying responses out to the network, hoping that some host has just sent a matching request and will be faked out by the response. If successful, such an attack could persuade a DHCP client to use a name server of your choice. With a maliciously configured name server, this would allow the attacker to see essentially all of the client's subsequent network traffic since nearly all transactions start with a name lookup.

Another possibility is to fake a name server response. If successful, this would allow the attacker to intercept the resulting connection, with the possibility of seeing sensitive data if the intercepted connection is to a bank Web site or something similar.

However, opening up `/dev/net/tun` as I have just done would require that such an attack be done blind, without being able to see any incoming packets. So, attacks on clients must be done randomly, which would require very high amounts of traffic for even a remote chance of success. Attacks on existing connections must similarly be done blind, with the added complication that the attack must correctly guess a random TCP sequence number.

So, normally, a successful attack would be remote. However, you should take this possibility seriously. The permissions on `/dev/net/tun` are a layer of protection against this sort of attack, and removing it increases the possibility of being attacked using an unrelated vulnerability. For example, if there was an exploit that allowed an attacker to sniff the network, the arguments I just made about how unlikely a successful attack would be go right out the window. Attacks would no longer be blind, and the attacker could see DHCP and name requests and try to respond to them through a TUN/TAP device, with good chances of success. In this case, the `/dev/net/tun` permissions would have likely stopped the attacker.

So, before opening up `/dev/net/tun`, consider whether you have untrusted, and possibly malicious, users on the host and whether you think there is any possibility of holes that would allow outsiders to gain shell access to the host. If that is remotely possible, you may consider a better option, which is used by Debian—create a `uml-users` group and make `/dev/net/tun` accessible only to members of that group. This reduces the number of accounts that could possibly be used to attack your network. It doesn't eliminate the risk, as one of those users could be malicious, or an outsider could gain access to one of those accounts.

However you have decided to set up `/dev/net/tun`, you should have read and write access to it, either as a normal user or as a member of a `uml-users` group. Once this is done, you can try the `tunctl` command again and it will succeed:

```
host% tunctl
Set 'tap0' persistent and owned by uid 500
```

This created a new TUN/TAP device and made it usable by the `tunctl` user.

For scripting purposes, a -b option makes tunctl output only the new device name:

```
host% tunctl -b
tap1
```

This eliminates the need to parse the relatively verbose output from the first form of the command.

There are also -u and -t options, which allow you to specify, respectively, which user the new TUN/TAP device will belong to and which TUN/TAP device that will be:

```
host# tunctl -u jdike -t jeffs-uml
Set 'jeffs-uml' persistent and owned by uid 500
```

This demonstrates a highly useful feature: the ability to give arbitrary names to the devices. Suitably chosen, these can serve as partial documentation of your UML network setup. We will use this jeffs-uml device from now on.

For cleanliness, we should shut down all of the TUN/TAP devices created by our playing with tunctl with commands such as the following:

```
host% tunctl -d tap0
Set 'tap0' nonpersistent
```

ifconfig -a will show you all the network interfaces on the system, so you should probably shut down all of the TUN/TAP devices except for the last one you made and any others created for some other specific reason.

The first thing to do is to enable the device:

```
host# ifconfig jeffs-uml 192.168.0.254 up
host# ifconfig jeffs-uml
jeffs-uml Link encap:Ethernet  HWaddr 2A:B1:37:41:72:D5
          inet addr:192.168.0.254  Bcast:192.168.0.255  \
Mask:255.255.255.0
          inet6 addr: fe80::28b1:37ff:fe41:72d5/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:5 overruns:0 carrier:0
          collisions:0 txqueuelen:500
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
```

As usual, choose an IP address that's suitable for your network. If IP addresses are scarce, you can reuse one that's already in use by the host, such as the one assigned to its eth0.

Basic Connectivity

Let's add a new interface to a UML instance. If you have an instance already running, you can plug a new network interface into it by using `uml_mconsole`:

```
uml_mconsole debian config eth0=tuntap,jeffs-uml
OK
```

If you are booting a new UML instance, you can do the same thing on the command line by adding `eth0=tuntap,jeffs-uml`.

This differs from the syntax we saw earlier. Before, we specified an IP address and no device name. Here, we specify the device name but not an IP address. When no device name is given, that signals the driver to invoke the `uml_net` helper to configure the host. When a name is given, the driver uses it and assumes that it has already been configured appropriately.

Now that the instance has an Ethernet device, we can configure it and bring it up:

```
UML# ifconfig eth0 192.168.0.253 up
```

Let's try pinging the host:

```
UML# ping 192.168.0.254
PING 192.168.0.254 (192.168.0.254): 56 data bytes

--- 192.168.0.254 ping statistics ---
28 packets transmitted, 0 packets received, 100% packet loss
```

Nothing but silence. The usual way to start debugging problems like this is to sniff the interface using `tcpdump` or a similar tool. With the ping running again, we see this:

```
host# tcpdump -i jeffs-uml -l -n
tcpdump: verbose output suppressed, use -v or -vv for full \
    protocol decode
listening on jeffs-uml, link-type EN10MB (Ethernet), capture \
    size 96 bytes
18:12:34.115634 IP 192.168.0.253 > 192.168.0.254: icmp 64: echo \
    request seq 0
18:12:35.132054 IP 192.168.0.253 > 192.168.0.254: icmp 64: echo \
    request seq 256
```

Ping requests are coming out, but no replies are getting back to it. This is a routing problem—we have not yet set any routes to the TUN/TAP

device, so the host doesn't know where to send the ping replies. This is easily fixed:

```
host# route add -host 192.168.0.253 dev jeffs-uml
```

Now, pinging from the UML instance works:

```
UML# ping 192.168.0.254
PING 192.168.0.254 (192.168.0.254): 56 data bytes
64 bytes from 192.168.0.254: icmp_seq=0 ttl=64 time=0.7 ms
64 bytes from 192.168.0.254: icmp_seq=1 ttl=64 time=0.1 ms
64 bytes from 192.168.0.254: icmp_seq=2 ttl=64 time=0.1 ms

--- 192.168.0.254 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.1/0.3/0.7 ms
```

It's always a good idea to check connectivity in both directions, in case there is a problem in one direction but not the other. So, check whether the host can ping the instance:

```
host% ping 192.168.0.253
PING 192.168.0.253 (192.168.0.253) 56(84) bytes of data.
64 bytes from 192.168.0.253: icmp_seq=0 ttl=64 time=0.169 ms
64 bytes from 192.168.0.253: icmp_seq=1 ttl=64 time=0.077 ms

--- 192.168.0.253 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 0.077/0.123/0.169/0.046 ms, pipe 2
```

So far, so good. The next step is to ping a host on the local network by its IP address:

```
UML# ping 192.168.0.3
PING 192.168.0.3 (192.168.0.3): 56 data bytes

--- 192.168.0.3 ping statistics ---
7 packets transmitted, 0 packets received, 100% packet loss
```

No joy. Using `tcpdump` to check what's happening shows this:

```
host# tcpdump -i jeffs-uml -l -n
tcpdump: verbose output suppressed, use -v or -vv for full \
    protocol decode
listening on jeffs-uml, link-type EN10MB (Ethernet), capture \
    size 96 bytes
18:20:29.522769 arp who-has 192.168.0.3 tell 192.168.0.253
18:20:30.524576 arp who-has 192.168.0.3 tell 192.168.0.253
18:20:31.522430 arp who-has 192.168.0.3 tell 192.168.0.253
```

The UML instance is trying to figure out the Ethernet MAC address of the target. To this end, it's broadcasting an arp request on its `eth0` interface and hoping for a response. It's not getting one because the target machine can't hear the request. arp requests, like other Ethernet broadcast protocols, are limited to the Ethernet segment on which they originate, and the UML `eth0` to host TUN/TAP connection is effectively an isolated Ethernet strand with only two hosts on it. So, the arp requests never reach the physical Ethernet where the other machine could hear it and respond.

This can be fixed by using a mechanism known as proxy arp and enabling packet forwarding. First, turn on forwarding:

```
host# echo 1 > /proc/sys/net/ipv4/ip_forward
```

Then enable proxy arp on the host for the TUN/TAP device:

```
host# echo 1 > /proc/sys/net/ipv4/conf/jeffs-uml/proxy_arp
```

This will cause the host to arp to the UML instance on behalf of the rest of the network, making the host's arp database available to the instance. Retrying the ping and watching `tcpdump` shows this:

```
host# tcpdump -i jeffs-uml -l -n tcpdump: verbose output \
    suppressed, use -v or -vv for full protocol decode
listening on jeffs-uml, link-type EN10MB (Ethernet), capture \
    size 96 bytes
19:25:16.465574 arp who-has 192.168.0.3 tell 192.168.0.253
19:25:16.510440 arp reply 192.168.0.3 is-at ae:42:d1:20:37:e5
19:25:16.510648 IP 192.168.0.253 > 192.168.0.3: icmp 64: echo \
    request seq 0
19:25:17.448664 IP 192.168.0.253 > 192.168.0.3: icmp 64: echo \
    request seq 256
```

There is still no pinging, but the arp request did get a response. We can verify this by seeing what's in the UML arp cache.

```
UML# arp
Address                  HWtype  HWaddress           Flags Mask \
    Iface
192.168.0.3              ether   AE:42:D1:20:37:E5   C          \
    eth0
```

If you see nothing here, it's likely because too much time elapsed between running the ping and the arp, and the arp entry got flushed from the cache. In this case, rerun the ping, and run arp immediately afterward.

Since the instance is now getting arp service for the rest of the network, and ping requests are making it out through the TUN/TAP device, we need to follow those packets to see what's going wrong. On my host, the outside network device is `eth1`, so I'll watch that. On other machines, the outside network will likely be `eth0`. It's also a good idea to select only packets involving the UML, to eliminate the noise from other network activity:

```
host# tcpdump -i eth1 -l -n host 192.168.0.253
tcpdump: verbose output suppressed, use -v or -vv for full \
    protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size \
    96 bytes
19:36:14.459076 IP 192.168.0.253 > 192.168.0.3: icmp 64: echo \
    request seq 0
19:36:14.461960 arp who-has 192.168.0.253 tell 192.168.0.3
19:36:15.460608 arp who-has 192.168.0.253 tell 192.168.0.3
```

Here we see a ping request going out, which is fine. We also see an arp request from the other host for the MAC address of the UML instance. This is going unanswered, so this is the next problem.

We set up proxy arp in one direction, for the UML instance on behalf of the rest of the network. Now we need to set it up in the other direction, for the rest of the network on behalf of the instance, so that the host will respond to arp requests for the instance:

```
host# arp -Ds 192.168.0.253 eth1 pub
```

Retrying the ping gets some good results:

```
UML# ping 192.168.0.3
PING 192.168.0.3 (192.168.0.3): 56 data bytes
64 bytes from 192.168.0.3: icmp_seq=0 ttl=63 time=133.1 ms
64 bytes from 192.168.0.3: icmp_seq=1 ttl=63 time=4.0 ms
64 bytes from 192.168.0.3: icmp_seq=2 ttl=63 time=4.9 ms

--- 192.168.0.3 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 4.0/47.3/133.1 ms
```

To be thorough, let's make sure we have connectivity in the other direction and ping the UML instance from the other host:

```
192.168.0.3% ping 192.168.0.254
PING 192.168.0.254 (192.168.0.254) from 192.168.0.3 : 56(84) \
    bytes of data.
64 bytes from 192.168.0.254: icmp_seq=1 ttl=64 time=6.48 ms
64 bytes from 192.168.0.254: icmp_seq=2 ttl=64 time=2.76 ms
```

```
64 bytes from 192.168.0.254: icmp_seq=3 ttl=64 time=2.75 ms

--- 192.168.0.254 ping statistics ---
3 packets transmitted, 3 received, 0% loss, time 2003ms
rtt min/avg/max/mdev = 2.758/4.000/6.483/1.756 ms
```

We now have basic network connectivity between the UML instance and the rest of the local network. Here's a summary of the steps we took.

1. Create the TUN/TAP device for the UML instance to use to communicate with the host.
2. Configure it.
3. Set a route to it.
4. Enable packet forwarding on the host.
5. Enable proxy arp in both directions between the UML instance and the rest of the network.

Thoughts on Security

At this point, the machinations of the `uml_net` helper should make sense. To recap, let's add another interface to the instance and let `uml_net` set it up for us:

```
host% uml_mconsole debian config eth1=tuntap,,,192.168.0.252
OK
```

Configuring the new device in the instance shows us this:

```
UML# ifconfig eth1 192.168.0.251 up
* modprobe tun
* ifconfig tap0 192.168.0.252 netmask 255.255.255.255 up
* bash -c echo 1 > /proc/sys/net/ipv4/ip_forward
* route add -host 192.168.0.251 dev tap0
* bash -c echo 1 > /proc/sys/net/ipv4/conf/tap0/proxy_arp
* arp -Ds 192.168.0.251 jeffs-uml pub
* arp -Ds 192.168.0.251 eth1 pub
```

Here we can see the helper doing just about everything we just finished doing by hand. The one thing that's missing is actually creating the TUN/TAP device. `uml_net` does that itself, without invoking an outside utility, so that doesn't show up in the list of commands it runs on our behalf.

Aside from knowing how to configure the host in order to support a networked UML instance, this is also important for understanding

the security implications of what we have done and for customizing this setup for a particular environment.

What `uml_net` does is not secure against a nasty root user inside the instance. Consider what would happen if the UML user decided to configure the UML `eth0` with the same IP address as your local name server. `uml_net` would set up proxy arp to direct name requests to the UML instance. The real name server would still be there getting requests, but some requests would be redirected to the UML instance. With a name server in the UML instance providing bogus responses, this could easily be a real security problem. For this reason, `uml_net` should not be used in a serious UML establishment. Its purpose is to make UML networking easy to set up for the casual UML user. For any more serious uses of UML, the host should be configured according to the local needs, security and otherwise.

What we just did by hand isn't that bad because we set the route to the instance and proxy arp according to the IP address we expected it to use. If root inside our UML instance decides to use a different IP address, such as that of our local name server, it will see no traffic. The host will only arp on behalf of the IP we expect it to use, and the route is only good for that IP. All other traffic will go elsewhere.

A nasty root user can still send out packets purporting to be from other hosts, but since it can't receive any responses to them, it would have to make blind attacks. As I discussed earlier, this is unlikely to enable any successful attacks on its own, but it does remove a layer of protection that might prove useful if another exploit on the host allows the attacker to see the local network traffic.

So, it is probably advisable to filter out any unexpected network traffic at the `iptables` level. First, let's see that the UML instance can send out packets that pretend to be from some other host. As usual for this discussion, these will be pings, but they could just as easily be any other protocol.

```
UML# ifconfig eth0 192.168.0.100 up
UML# ping 192.168.0.3
PING 192.168.0.3 (192.168.0.3): 56 data bytes

--- 192.168.0.3 ping statistics ---
4 packets transmitted, 0 packets received, 100% packet loss
```

Here I am pretending to be `192.168.0.100`, which we will consider to be an important host on the local network. Watching the `jeffs-uml` device on the host shows this:

```
host# tcpdump -i jeffs-uml -l -n
tcpdump: verbose output suppressed, use -v or -vv for full \
    protocol decode
listening on jeffs-uml, link-type EN10MB (Ethernet), capture \
    size 96 bytes
20:20:34.978090 arp who-has 192.168.0.3 tell 192.168.0.100
20:20:35.506878 arp reply 192.168.0.3 is-at ae:42:d1:20:37:e5
20:20:35.508062 IP 192.168.0.100 > 192.168.0.3: icmp 64: echo \
    request seq 0
```

We can see those faked packets reaching the host. Looking at the host's interface to the rest of the network, we can see they are reaching the local network:

```
tcpdump -i eth1 -l -n
tcpdump: verbose output suppressed, use -v or -vv for full \
    protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size \
    96 bytes
20:23:30.741482 IP 192.168.0.100 > 192.168.0.3: icmp 64: echo \
    request seq 0
20:23:30.744305 arp who-has 192.168.0.100 tell 192.168.0.3
```

Notice that arp request. It will be answered correctly, so the ping responses will go to the actual host that legitimately owns `192.168.0.100`, which is not expecting them. That host will discard them, so they will cause no harm except for some wasted network bandwidth and CPU cycles. However, it would be preferable for those packets not to reach the network or the host in the first place. This can be done as follows:

```
host# iptables -A FORWARD -i jeffs-uml -s \! 192.168.0.253 -j \
    DROP
Warning: wierd character in interface `jeffs-uml' (No aliases, \
    :, ! or *).
```

`iptables` is apparently complaining about the dash in the interface name, but it does create the rule, as we can see here:

```
host# iptables -L
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
DROP       all  -- !192.168.0.253        anywhere

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

So, we have just told `iptables` to discard any packet it sees that:

- Is supposed to be forwarded
- Enters the host through the `jeffs-uml` interface
- Has a source address other than 192.168.0.253

After creating this firewall rule, you should be able to rerun the previous ping and `tcpdump` will show that those packets are not reaching the outside network.

At this point, we have a reasonably secure setup. As originally configured, the UML instance couldn't see any traffic not intended for it. With the new firewall rule, the rest of the network will see only traffic from the instance that originates from the IP address assigned to it. A possible enhancement to this is to log any attempts to use an unauthorized IP address so that the host administrator is aware of any such attempts and can take any necessary action.

You could also block any packets from coming in to the UML instance with an incorrect IP address. This shouldn't happen because the proxy arp we have set up shouldn't attract any packets for IP addresses that don't belong somehow to the host, and any such packets that do reach the host won't be routed to the UML instance. However, an explicit rule to prevent this might be a good addition to a layered security model. In the event of a malfunction or compromise of this configuration, such a rule could end up being the one thing standing in the way of a UML instance seeing traffic that it shouldn't. This rule would look like this:

```
host# iptables -A FORWARD -o jeffs-uml -d \! 192.168.0.253 -j \
    DROP
Warning: wierd character in interface `jeffs-uml' (No aliases, \
    :, ! or *).
```

Access to the Outside Network

We still have a bit of work to do, as we have demonstrated access only to the local network, using IP addresses rather than more convenient host names. So, we need to provide the UML instance with a name service. For a single instance, the easiest thing to do is copy it from the host:

```
host# cat > /etc/resolv.conf
; generated by /sbin/dhclient-script
search user-mode-linux.org
nameserver 192.168.0.3
```

I cut the contents of the host's `/etc/resolv.conf` and pasted them into the UML. You should do the same on your own machine, as my `resolv.conf` will almost certainly not work for you.

We also need a default route, which hasn't been necessary for the limited testing we've done so far but is needed for almost anything else:

```
UML# route add default gw 192.168.0.254
```

I normally use the IP address of the host end of the TUN/TAP device as the default gateway.

If you still have the unauthorized IP address assigned to your instance's `eth0`, reassign the original address:

```
ifconfig eth0 192.168.0.253
```

Now we should have name service:

```
UML# host 192.168.0.3
Name: laptop.user-mode-linux.org
Address: 192.168.0.3
```

That's a local name—let's check for a remote one:

```
UML# host www.user-mode-linux.org
www.user-mode-linux.org A       66.59.111.166
```

Now let's try pinging it, to see if we have network access to the outside world:

```
UML# ping www.user-mode-linux.org
PING www.user-mode-linux.org (66.59.111.166): 56 data bytes
64 bytes from 66.59.111.166: icmp_seq=0 ttl=52 time=487.2 ms
64 bytes from 66.59.111.166: icmp_seq=1 ttl=52 time=37.8 ms
64 bytes from 66.59.111.166: icmp_seq=2 ttl=52 time=36.0 ms
64 bytes from 66.59.111.166: icmp_seq=3 ttl=52 time=73.0 ms

--- www.user-mode-linux.org ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 36.0/158.5/487.2 ms
```

Copying `/etc/resolv.conf` from the host and setting the default route by hand works but is not the right thing to do. The real way to do these is with DHCP. The reason this won't work here is the same reason that ARP didn't work—the UML is on a different Ethernet strand than the rest of the network, and DHCP, being an Ethernet broadcast protocol, doesn't cross Ethernet broadcast domain boundaries.

DHCP through a TUN/TAP Device

Some tools work around the DHCP problem by forwarding DHCP requests from one Ethernet domain to another and relaying whatever replies come back. One such tool is `dhcp-fwd`. It needs to be installed on the host and configured. It has a fairly scary-looking default config file. You need to specify the interface from which client requests will come and the interface from which server responses will come.

In the default config file, these are `eth2` and `eth1`, respectively.

On my machine, the client interface is `jeffs-uml` and the server interface is `eth1`. So, a global replace of `eth2` with `jeffs-uml`, and leaving `eth1` alone, is sufficient to get a working `dhcp-fwd`.

Let's get a clean start by unplugging the UML `eth0` and plugging it back in. First we need to bring the interface down:

```
UML# ifconfig eth0 down
```

Then, on the host, remove the device:

```
host% uml_mconsole debian remove eth0
OK
```

Now, let's plug it back in:

```
host% uml_mconsole debian config eth0=tuntap,,fe:fd:c0:a8:00:fd,\
    192.168.0.254
OK
```

Notice that we have a new parameter to this command. We are specifying a hardware MAC address for the interface. We never did this before because the UML network driver automatically generates one when it is assigned an IP address for the first time. It is important that these be unique. Physical Ethernet cards have a unique MAC burned into their hardware or firmware. It's tougher for a virtual interface to get a unique identity. It's also important for its IP address to be unique, and I have taken advantage of this in order to generate a unique MAC address for a UML's Ethernet device.

When the administrator provides an IP address, which is very likely to be unique on the local network, to a UML Ethernet device, the driver uses that as part of the MAC address it assigns to the device. The first two bytes of the MAC will be `0xFE` and `0xFD`, which is a private Ethernet range. The next four bytes are the IP address. If the IP address is unique on the network, the MAC will be, too.

When configuring the interface with DHCP, the MAC is needed before the DHCP server can assign the IP. Thus, we need to assign the

MAC on the command line or when plugging the device into a running UML instance.

There is another case where you may need to supply a MAC on the UML command line, which I will discuss in greater detail later in this chapter. That is when the distribution you are using brings the interface up before giving it an IP address. In this case, the driver can't supply the MAC after the fact, when the interface is already up, so it must be provided ahead of time, on the command line.

Now, assuming the `dhcp-fwd` service has been started on the host, `dhclient` will work inside UML:

```
UML# dhclient eth0
Internet Software Consortium DHCP Client 2.0pl5
Copyright 1995, 1996, 1997, 1998, 1999 The Internet Software \
    Consortium.
All rights reserved.

Please contribute if you find this software useful.
For info, please visit http://www.isc.org/dhcp-contrib.html

Listening on LPF/eth0/fe:fd:c0:a8:00:fd
Sending on   LPF/eth0/fe:fd:c0:a8:00:fd
Sending on   Socket/fallback/fallback-net
DHCPREQUEST on eth0 to 255.255.255.255 port 67
DHCPACK from 192.168.0.254
bound to 192.168.0.9 -- renewal in 21600 seconds.
```

Final Testing

At this point, we have full access to the outside network. There is still one thing that could go wrong. Ping packets are relatively small; in some situations small packets will be unmolested but large packets, contained in full-size Ethernet frames, will be lost. To check this, we can copy in a large file:

```
UML# wget http://www.kernel.org/pub/linux/kernel/v2.6/\
    linux-2.6.12.3.tar.bz2
--01:35:56--  http://www.kernel.org/pub/linux/kernel/v2.6/\
    linux-2.6.12.3.tar.bz2           => `linux-2.6.12.3.tar.bz2'
Resolving www.kernel.org... 204.152.191.37, 204.152.191.5
Connecting to www.kernel.org[204.152.191.37]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 37,500,159 [application/x-bzip2]

100%[====================================>] 37,500,159    \
     87.25K/s    ETA 00:00

01:43:04 (85.92 KB/s) - `linux-2.6.12.3.tar.bz2' saved \
    [37500159/37500159]
```

Copying in a full Linux kernel tarball is a pretty good test, and in this case, it's fine. If this does nothing for you, it's likely that there's a problem with large packets. If so, you need to lower the Maximal Transfer Unit (MTU) of the UML's `eth0`:

```
UML# ifconfig eth0 mtu 1400
```

You can determine the exact value by experiment. Lower it until large transfers start working.

The cases where I've seen this involved a PPPoE connection to the outside world. PPPoE usually means a DSL connection, and I've seen UML connectivity problems when the host was my DSL gateway. Lowering the MTU to 1400 made the network fully functional. In fact, the MTU for a PPPoE connection is 1492, so lowering it to 1400 was overkill.

Bridging

As mentioned at the start of this chapter, there are two ways to configure a host to give a UML access to the outside world. We just explored one of them. The alternative, bridging, doesn't require the host to route packets to and from the UML, and so doesn't require new routes to be created or proxy arp to be configured. With bridging, the TUN/TAP device used by the UML instance is combined with the host's physical Ethernet device into a sort of virtual switch. The bridge interface forwards Ethernet frames from one interface to another based on their destination MAC addresses. This effectively merges the broadcast domains associated with the bridged interfaces. Since this caused DHCP and arp to not work when we were doing IP forwarding, bridging provides a neat solution to these problems.

If you currently have an active UML network, you should shut it down before continuing:

```
UML# ifconfig eth0 down
```

Then, on the host, remove the device:

```
host% uml_mconsole debian remove eth0
OK
```

Bring down and remove the TUN/TAP interface, which will delete the route and one side of the proxy arp, and delete the other side of the proxy arp:

```
host# ifconfig jeffs-uml down
host% tunctl -d jeffs-uml
Set 'jeffs-uml' nonpersistent
host# arp -i jeffs-uml -d 192.168.0.253 pub
```

Now, with everything cleaned up, we can start from scratch:

```
host% tunctl -u jdike -t jeffs-uml
```

Let's start setting up bridging. The idea is that a new interface will provide the host with network access to the outside world. The two interfaces we are currently using, `eth0` and `jeffs-uml`, will be added to this new interface. The bridge device will forward frames from one interface to the other as needed, so that both `eth0` and `jeffs-uml` will see traffic that's intended for them (or that needs to be sent to the local network, in the case of `eth0`).

The first step is to create the device using the `brctl` utility, which is in the `bridge-utilities` package of your favorite distribution:

```
host# brctl addbr uml-bridge
```

In the spirit of giving interfaces meaningful names, I've called this one `uml-bridge`.

Now we want to add the two existing interfaces to it. For the physical interface, choose a wired Ethernet—for some reason, wireless interfaces don't seem to work in bridges. The virtual interface will be the `jeffs-uml` TUN/TAP interface.

We need to do some configuration to make it usable:

```
host# ifconfig jeffs-uml 0.0.0.0 up
```

These interfaces can't have their own IP addresses, so we have to clear the one on `eth0`. This is a step you want to think about carefully. If you are logged in to the host remotely, this will likely kill your session and any network access you have to it. If the host has two network interfaces, and you know that your session and all other network activity you care about is traveling over the other, then it should be safe to remove the IP address from this one:

```
host# ifconfig eth0 0.0.0.0
```

We can now add the two interfaces to the bridge:

```
host# brctl addif uml-bridge jeffs-uml
host# brctl addif uml-bridge eth0
```

And then we can look at our work:

```
host# brctl show
bridge name     bridge id               STP enabled     \
    interfaces
uml-bridge              8000.0012f04be1fa       no      \
    eth0
                                                        \
    jeffs-uml
```

At this point, the bridge configuration is done and we need to bring it up as a new network interface:

```
host# dhclient uml-bridge
Internet Systems Consortium DHCP Client V3.0.2
Copyright 2004 Internet Systems Consortium.
All rights reserved.
For info, please visit http://www.isc.org/products/DHCP

/sbin/dhclient-script: configuration for uml-bridge not found. \
    Continuing with defaults.
Listening on LPF/uml-bridge/00:12:f0:4b:e1:fa
Sending on   LPF/uml-bridge/00:12:f0:4b:e1:fa
Sending on   Socket/fallback
DHCPDISCOVER on uml-bridge to 255.255.255.255 port 67 interval 4
DHCPOFFER from 192.168.0.10
DHCPREQUEST on uml-bridge to 255.255.255.255 port 67
DHCPACK from 192.168.0.10
/sbin/dhclient-script: configuration for uml-bridge not found. \
    Continuing with defaults.
bound to 192.168.0.2 -- renewal in 20237 seconds.
```

The bridge is functioning, but for any local connectivity to the UML instance, we'll need to set a route to it:

```
host# route add -host 192.168.0.253 dev uml-bridge
```

Now we can plug the interface into the UML instance and configure it there:

```
host% uml_mconsole debian config eth0=tuntap,jeffs-uml,\
    fe:fd:c0:a8:00:fd
OK

UML# ifconfig eth0 192.168.0.253 up
```

Note that we plugged the `jeffs-uml` TUN/TAP interface into the UML instance. The bridge is merely a container for the other two interfaces, which can actually send and receive frames.

Also note that we assigned the MAC address ourselves rather than letting the UML driver do it. A MAC is necessary in order to make a DHCP request for an IP address, while the driver requires the IP address before it can construct the MAC. In order to break this circular requirement, we need to assign the MAC that the interface will get.

Now we can see some benefit from the extra setup that the bridge requires. DHCP within the UML instance now works:

```
UML# dhclient eth0
Internet Systems Consortium DHCP Client V3.0.2-RedHat
Copyright 2004 Internet Systems Consortium.
All rights reserved.
For info, please visit http://www.isc.org/products/DHCP

Listening on LPF/eth0/fe:fd:c0:a8:00:fd
Sending on   LPF/eth0/fe:fd:c0:a8:00:fd
Sending on   Socket/fallback
DHCPDISCOVER on eth0 to 255.255.255.255 port 67 interval 5
DHCPOFFER from 192.168.0.10
DHCPREQUEST on eth0 to 255.255.255.255 port 67
DHCPACK from 192.168.0.10
bound to 192.168.0.253 -- renewal in 16392 seconds.
```

This requires no messing around with `arp` or `dhcp-fwd`. Binding the TUN/TAP interface and the host's Ethernet interface makes each see broadcast frames from the other. So, DHCP and arp requests sent from the TUN/TAP device are also sent through the `eth0` device. Similarly, arp requests from the local network are forwarded to the TUN/TAP interface (and thus the UML instance's `eth0` interface), which can respond on behalf of the UML instance.

The bridge also forwards nonbroadcast frames, based on their MAC addresses. So, DHCP and arp replies will be forwarded as necessary between the two interfaces and thus between the UML instance and the local network. This makes the DHCP forwarding and the proxy arp that we did earlier completely unnecessary.

The main downside to bridging is the need to remove the IP address from the physical Ethernet interface before adding it to the bridge. This is a rather pucker-inducing step when the host is accessible only remotely over that one interface. Many people will use IP forwarding and proxy arp instead of bridging rather than risk taking their remote server off the net. Others have written scripts that set up the bridge, taking the server's Ethernet interface offline and bringing the bridge interface online.

Bridging and Security

Bridging provides access to the outside network in a different way than we got with routing and proxy arp. However, the security concerns are the same—we need to prevent a malicious root user from making the UML instance pretend to be an important server. Before, we filtered traffic going through the TUN/TAP device with `iptables`. This was appropriate for a situation that involved IP-level routing and forwarding, but it won't work here because the forwarding is done at the Ethernet level.

There is an analogous framework for doing Ethernet filtering and an analogous tool for configuring it: `ebtables`, with the "eb" standing for "Ethernet Bridging."

First, in order to demonstrate that we can do nasty things to our network, let's change our Ethernet MAC to one we will presume belongs to our name server or DHCP server. Then let's verify that we still have network access:

```
UML# ifconfig eth0 hw ether fe:fd:ba:ad:ba:ad
# ping -c 2 192.168.0.10
PING 192.168.0.10 (192.168.0.10) 56(84) bytes of data.
64 bytes from 192.168.0.10: icmp_seq=0 ttl=64 time=3.75 ms
64 bytes from 192.168.0.10: icmp_seq=1 ttl=64 time=1.85 ms

--- 192.168.0.10 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1018ms
rtt min/avg/max/mdev = 1.850/2.803/3.756/0.953 ms, pipe 2
```

We do, so we need to fix things so that the UML instance has network access only with the MAC we assigned to it.

Precisely, we want any Ethernet frame leaving the `jeffs-uml` interface on its way to the bridge that doesn't have a source MAC of `fe:fd:c0:a8:00:fd` to be dropped. Similarly, we want any frame being forwarded from the bridge to the `jeffs-uml` interface without that destination MAC to be dropped.

The `ebtables` syntax is very similar to `iptables`, and the following commands do what we want:

```
host# ebtables -A INPUT --in-interface jeffs-uml \
    --source \! FE:FD:C0:A8:00:FD -j DROP
host# ebtables -A OUTPUT --out-interface jeffs-uml \
    --destination \! FE:FD:C0:A8:00:FD -j DROP
host# ebtables -A FORWARD --out-interface jeffs-uml \
    --destination \! FE:FD:C0:A8:00:FD -j DROP
host# ebtables -A FORWARD --in-interface jeffs-uml \
    --source \! FE:FD:C0:A8:00:FD -j DROP
```

There is a slight subtlety here—my first reading of the `ebtables` man page suggested that using the `FORWARD` chain would be sufficient since that covers frames being forwarded by the bridge from one interface to another. This works for external traffic but not for traffic to the host itself. These frames aren't forwarded, so we could spoof our identity to the host if the `ebtables` configuration used only the `FORWARD` chain. To close this hole, I also use the `INPUT` and `OUTPUT` chains to drop packets intended for the host as well as those that are forwarded.

At this point the `ebtables` configuration should look like this:

```
ebtables -L
Bridge table: filter

Bridge chain: INPUT, entries: 1, policy: ACCEPT
-s ! fe:fd:c0:a8:0:fd -i jeffs-uml -j DROP

Bridge chain: FORWARD, entries: 2, policy: ACCEPT
-d ! fe:fd:c0:a8:0:fd -o jeffs-uml -j DROP
-s ! fe:fd:c0:a8:0:fd -i jeffs-uml -j DROP

Bridge chain: OUTPUT, entries: 1, policy: ACCEPT
-d ! fe:fd:c0:a8:0:fd -o jeffs-uml -j DROP
```

We can check our work by trying to ping an outside host again:

```
host# ping -c 2 192.168.0.10
PING 192.168.0.10 (192.168.0.10) 56(84) bytes of data.
From 192.168.0.253 icmp_seq=0 Destination Host Unreachable
From 192.168.0.253 icmp_seq=1 Destination Host Unreachable

--- 192.168.0.10 ping statistics ---
2 packets transmitted, 0 received, +2 errors, 100% packet \
  loss, time 1018ms, pipe 3
```

We should also check that we haven't made things too secure by accidentally dropping all packets. Let's reset our MAC to the approved value and see that we have connectivity:

```
UML# ifconfig eth0 hw ether FE:FD:C0:A8:00:FD
UML# ping -c 2 192.168.0.10
PING 192.168.0.10 (192.168.0.10) 56(84) bytes of data.
64 bytes from 192.168.0.10: icmp_seq=0 ttl=64 time=40.4 ms
64 bytes from 192.168.0.10: icmp_seq=1 ttl=64 time=3.93 ms

--- 192.168.0.10 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1036ms
rtt min/avg/max/mdev = 3.931/22.190/40.449/18.259 ms, pipe 2
```

At this point, the UML instance can communicate with other hosts using only the MAC that we assigned to it. We should also be concerned with whether it can do harm by spoofing its IP.

```
UML# ifconfig eth0 192.168.0.100
UML# ifconfig eth0 hw ether FE:FD:C0:A8:00:FD
UML# ping -c 2 192.168.0.10
PING 192.168.0.10 (192.168.0.10) 56(84) bytes of data.
64 bytes from 192.168.0.10: icmp_seq=0 ttl=64 time=3.57 ms
64 bytes from 192.168.0.10: icmp_seq=1 ttl=64 time=1.73 ms

--- 192.168.0.10 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1017ms
rtt min/avg/max/mdev = 1.735/2.655/3.576/0.921 ms, pipe 2
```

It can, so we need to apply some IP filtering. Because the `jeffs-uml` interface is part of a bridge, we need to use the `physdev` module of `iptables`:

```
host# iptables -A FORWARD -m physdev --physdev-in jeffs-uml \
     -s \! 192.168.0.253 -j DROP
Warning: wierd character in interface `jeffs-uml' (No aliases, \
     :, ! or *).
host# iptables -A FORWARD -m physdev --physdev-out jeffs-uml \
     -d \! 192.168.0.253 -j DROP
Warning: wierd character in interface `jeffs-uml' (No aliases, \
     :, ! or *).
host# iptables -A INPUT -m physdev --physdev-in jeffs-uml \
     -s \! 192.168.0.253 -j DROP
Warning: wierd character in interface `jeffs-uml' (No aliases, \
     :, ! or *).
host# iptables -A OUTPUT -m physdev --physdev-out jeffs-uml \
     -d \! 192.168.0.253 -j DROP
Warning: wierd character in interface `jeffs-uml' (No aliases, \
     :, ! or *).
```

These take care of packets intended for both this host and other systems. Earlier, I didn't include a rule to prevent packets with incorrect destination IP addresses from reaching a UML instance because the proxy arp and routing provided pretty good protection against that. I'm including the equivalent rule here because we don't have the same protection—the bridging exposes the UML instances much more to the local network.

THE UML NETWORKING TRANSPORTS

Now that we've had an in-depth look at using TUN/TAP devices on the host to get a UML instance on the network, it's time to look at the other

mechanisms that can be used. There are a total of six, probably two of which are by far the most commonly used. However, there are situations in which you would choose to use one of the other four, albeit very rare situations for some of them.

In order to classify them, we can first divide them between transports that can be used to connect a UML to the host and those that can be used only to connect UML instances to each other. In the first group are TUN/TAP, Ethertap, SLIP, and Slirp. In the second are the virtual switch and multicast. Blurring this distinction somewhat is that `uml_switch` has an option to attach itself to a host TUN/TAP device, thereby providing access to the host. The final transport, `pcap`, is fundamentally different from the others and doesn't really belong in either group. It does connect to the host, but it can only receive packets, not transmit them. `pcap` allows you to use a UML instance as a preconfigured packet sniffer.

Access to the Host Network

TUN/TAP and Ethertap

Among the transports that can provide access to the host network, TUN/TAP is very much the preferred option. Ethertap is an older interface that does the same thing, only worse. Ethertap was the standard for this on Linux version 2.2, and early in 2.4. At that point, TUN/TAP entered the Linux kernel in its current form. It supplanted Ethertap because it lacked various problems that made Ethertap hard to work with.

These problems are pretty well hidden by the `uml_net` helper and the UML Ethertap driver, but they do affect performance and possibly security. These effects are caused by the fact that there needs to be a root helper to create the Ethertap device and to handle every packet going through the device. It's impossible for the helper to open a file descriptor to the Ethertap interface and pass it to UML, as is the case with TUN/TAP. So, UML sends and receives packets over a pipe to the helper, which communicates with the interface. This extra step hurts latency and throughput compared to TUN/TAP. Having a root helper running continuously may also be a security issue, as it would be a continuous target for any attacks.

The one advantage that Ethertap has over TUN/TAP is that it's available on Linux kernels that predate early version 2.4. So, if you have a host running such a kernel, and it can't be updated, you have to use Ethertap for your UML networking.

SLIP

The SLIP transport exists because it was the first networking mechanism for UML. Ethertap was available on the first host on which I developed UML, but SLIP was the first mechanism I learned about. There is essentially no reason to use it now. The only one I can think of is that maybe some UML hosts don't have either TUN/TAP or Ethertap available, and this can't be changed. Then SLIP would be the mechanism of choice, even though it's a poor choice.

The following issues are among its disadvantages.

- ☞ It can carry only IP traffic. Important non-IP protocols such as DHCP and ARP, and other lesser-known protocols from the likes of Apple and Novell, can't be carried over it.
- ☞ The encapsulation required by the SLIP protocol is a performance drag.
- ☞ It can't carry Ethernet frames, so it can't talk directly to an Ethernet network. All packets must be routed through the host, which will convert them into Ethernet frames.

Slirp

Slirp is interesting but little used. The Slirp networking emulator provides network access without needing any root privileges or help whatsoever. It is unique in this regard, as all of the other transports require some sort of root assistance.

However, it has a number of disadvantages.

- ☞ It is slow. Slirp contains a network stack of its own that is used to parse the packets coming from the UML network stack. Slirp opens a normal socket connection to the target and sends the packet payload to it. When receiving packets, the process is reversed. The data coming from the remote side is assembled into a network packet that is immediately disassembled by the UML network stack.
- ☞ It can't receive connections on well-known ports. Since it receives connections by attaching to host ports, as an unprivileged process, it can only attach to ports greater than 1024. Since it doesn't act as a full network node, it can't have its own ports that the host can route packets to.
- ☞ The disadvantages of SLIP also apply, since Slirp provides an emulated SLIP connection.

Nevertheless, in some situations, Slirp is the only mechanism for providing a UML instance access to the outside network. I've seen cases where people are running UML instances on hosts on which they have no privileges. In one case, the "host" was a vserver instance on which the user had "root" privileges, but the vserver was so limited that Slirp was the only way to get the UML instance on the network. Cases like these are rare, but when they do happen, Slirp is invaluable, despite its limitations.

Isolated Networks

There are two purely virtual transports, which can connect a UML only to other UML instances: `uml_switch` and multicast.

`uml_switch`

`uml_switch` is a process that implements a virtual switch. UML instances connect to it and communicate with it over a UNIX domain socket on the host. It can act as a switch, which is its normal operation, or as a hub, which is sometimes useful when you want to sniff the traffic between two UML instances from a third. It also has the ability to connect to a preconfigured TUN/TAP device, allowing the UML instances attached to it to communicate with the host and outside network.

Multicast

Multicast is the second purely virtual network transport for UML. As its name suggests, it uses a multicast network on the host in order to transmit Ethernet frames from one UML instance to another. The UML instances all join the same multicast network, so that a packet sent from any instance is seen by all of the others. This is somewhat less efficient than the virtual switch because it behaves like a hub—all packets are received by all nodes attached to it. So, the UML instances will have to process and drop any packets that aren't intended for it, unnecessarily consuming host CPU time.

`pcap`

The last transport is unlike the others, in that it doesn't provide two-way network traffic. A UML interface based on `pcap` is read-only—it

receives packets but doesn't transmit them. This allows UML to act as a preconfigured network sniffer. A variety of network sniffing and traffic analysis tools are available, and they can be complicated to configure. This transport makes it possible to install a set of network analysis tools in a UML root filesystem, configure them, and distribute the filesystem.

Users can then boot UML on this filesystem and specify the `pcap` interface on the command line or with `uml_mconsole`. The traffic analysis will then work, with no further configuration needed.

As the name suggests, this transport is based on the `pcap` library, which underlies `tcpdump` and other tools. Use of this may require some familiarity with `libpcap` or `tcpdump`, especially if you want to filter packets before the tools inside UML see them. In this case, you will need to provide a filter expression to select the desired packets. Anyone who has done anything similar with `tcpdump` will know how to write an appropriate expression. For those who have not used `tcpdump`, the man page contains a good reference to the expression language.

How to Choose the Right Transport

Now that we've seen all of the UML network transports, we can make decisions about when to use each one. The advantages and disadvantages discussed earlier should make this pretty clear, but it's useful to summarize them.

If you need to give the UML instances access to the outside network, TUN/TAP is preferred. This has been standard in Linux kernels since early version 2.4, so virtually all Linux machines that might host UML instances should be sufficiently new to have TUN/TAP support. If you have one that is not, upgrading would probably be a better idea than falling back to Ethertap.

Once you've decided to use TUN/TAP, the next decision is whether to give each UML its own TUN/TAP device or to connect them with `uml_switch` and have it forward packets to the host through its own TUN/TAP interface. Using the switch instead of individual TUN/TAP devices has a number of trade-offs.

- ☞ The switch is a single point of control, meaning that bandwidth tracking and management as well as filtering can be done at a single interface, and it is a single point of failure.
- ☞ The switch is more efficient than individual TUN/TAP devices for traffic between the UML instances because the packets experience

only Ethernet routing by the switch rather than IP routing by the host. However, for external traffic, there's one more process handling the packets, so that will introduce more latency.

☞ The switch may be less of a security worry. If you are concerned about making `/dev/net/tun` world accessible (or even group accessible by a `uml-users` group), you may be happier having it owned by a user whose only purpose is to run `uml_switch`. In this way, faked packets can be injected into the host only by an attacker who has managed to penetrate that one account.

Against this, there is the UNIX socket that `uml_switch` uses to set up connections with UML instances. This needs to be writable by any users who are allowed to connect UML instances to the switch. A rogue process could possibly connect to it and inject packets to the switch, for forwarding to the UML instances or the outside network.

This would seem to be a wash, where we are replacing a security concern about `/dev/net/tun` with the same concern about the UNIX socket used by the switch. However, access to `/dev/net/tun` allows the creation of new interfaces, which aren't subject to whatever filtering is applied to "authorized" TUN/TAP interfaces. Any packets injected through the UNIX socket that go to the outside network will need to pass through the filters on the TUN/TAP interface used by the switch. On balance, I would have to call this a slight security gain.

SLIP and Slirp are useful only in very limited circumstances. Again, I would recommend fixing the host so that TUN/TAP can be used before using either SLIP or Slirp. If you must get a UML with network access, and you have absolutely no way to get root assistance, you may need to use Slirp.

For an isolated network, the choice is between `uml_switch` and multicast. Multicast is trivial to set up, as we will see in the next section. However, the switch isn't that difficult either. If you want a quick-and-dirty isolated network, multicast is likely the better choice. However, multicast is less efficient because of the hub behavior I mentioned earlier.

Configuring the Transports

We need to take care of one loose end. The usage of the transports varies somewhat because of their differing configuration needs. In most

cases, these differences are confined to the configuration string provided to UML on the command line or to `uml_mconsole`. In the case of `uml_switch`, we also need to look at the invocation of the switch.

Despite the differences, there are some commonalities. The parameters to the device are separated by commas. Many parameters are optional; to exclude one, just specify it as an empty string. Trailing commas can be omitted. For example, a TUN/TAP interface that the `uml_net` helper will set up can look like this:

```
eth0=tuntap,,fd:fe:1:2:3:4,192.168.0.1
```

Leaving out the Ethernet MAC would make it look like this:

```
eth0=tuntap,,,192.168.0.1
```

Omitted parameters will be provided with default values. In the case above, the omitted MAC will be initialized as described below. The omitted TUN/TAP interface name will be determined by the `uml_net` helper when it configures the interface.

The transports that create an Ethernet device inside UML can take an Ethernet MAC in the device specification. If not specified, it will be assigned a MAC when it is first assigned an IP address. The MAC will be derived from the IP—the first two bytes are `0xfd` and `0xfe`, and the last four are the IP address. This makes the MAC as unique as the IP address. Normally, the MAC can be left out. However, when you want the UML instance to be able to use DHCP, you must specify a MAC because the device will not operate without one and it must have a MAC in order for the DHCP server to provide an IP address. When it is acceptable for the UML interface to not work until it is assigned an IP address, you can let the driver assign the MAC.

However, if the interface is already up before it is assigned an IP address, the driver cannot change the MAC address on its own. Some distributions enable interfaces like this. In this case, the MAC will end up as `fd:fe:00:00:00:00`. If you are running several UML instances, it is likely that these MACs will conflict, causing mysterious network failures. The easiest way to fix this problem is to provide the MAC on the command line. You can also take the interface down and bring it back up by hand. When you bring it back up, you should specify the IP address on the `ifconfig` command line. This will ensure that the driver knows the IP address when the interface is enabled, so it can be assigned a reasonable MAC.

Whenever there is a network interface on the host that the transport communicates through, such as a TUN/TAP or Ethertap device,

the IP address of that interface, the host-side IP, can be included. As we saw earlier in the chapter, when an IP address is specified in the device configuration, the driver will run the `uml_net` helper in order to set up the device on the host. When it is omitted, a preconfigured host device should be included in the configuration string.

As we have already seen, the configuration syntax for a device is identical whether it is being configured on the UML command line or being hot-plugged with an MConsole client.

TUN/TAP

The TUN/TAP configuration string comes in two forms, depending on whether you are assigning the UML interface a preconfigured host interface or whether you want the `uml_net` helper to create and configure the host interface.

In the first case, you specify

- `tuntap`
- The host interface name
- Optionally, the MAC of the UML interface

For example:

```
eth0=tuntap,my-uml-tap,fe:fd:1:2:3:4
```

or

```
eth0=tuntap,my-uml-tap
```

In the second case, you specify

- `tuntap`
- An empty parameter, in place of the host interface name
- Optionally, the MAC of the UML interface
- The IP address of the host interface to be configured

For example:

```
eth0=tuntap,,fe:fd:1:2:3:4,192.168.0.1
```

or

```
eth0=tuntap,,,192.168.0.1
```

The three commas mean that parameters two and three (the host interface name and Ethernet MAC) are empty and will be assigned values by the driver.

Ethertap

The Ethertap configuration string is nearly identical, except that the device type is `ethertap` and that you must specify a host interface name. When the host interface doesn't exist and you provide an IP address, `uml_net` will configure that device. This example tells the driver to use a preconfigured Ethertap interface:

```
eth0=ethertap,tap0
```

This results in the `uml_net` helper creating and configuring a new Ethertap interface:

```
eth0=ethertap,tap0,,192.168.0.1
```

SLIP

The SLIP configuration is comparatively simple—only the IP address of the host SLIP device needs to be specified. It must be there since `uml_net` will always run in order to configure the SLIP interface. There is no possibility of specifying a MAC since the UML interface will not be an Ethernet device. This means that DHCP and other Ethernet protocols, such as ARP, can't be used with SLIP.

```
eth0=slip,192.168.0.1
```

Slirp

The Slirp configuration requires

- ☞ `slirp`
- ☞ Optionally, the MAC of the UML interface
- ☞ The command line of the Slirp executable

If you decide to try this, you should probably first configure and run Slirp without UML. Once you can run it by hand, you can put the Slirp command line in the configuration string and it will work as it did before.

Adding the Slirp command line requires that it be transformed somewhat in order to not confuse the driver's parser. First, the command and its arguments should be separated by commas rather than

spaces. Second, any spaces embedded in an argument should be changed to underscores. However, in the normal case Slirp takes no arguments, and only the path to the Slirp executable needs to be specified.

If some arguments need to be provided, Slirp will read options from your `~/.sliprc`. Putting the requisite information there will simplify the UML command line. It is also possible to pass the name of a wrapper script that will invoke `slirp` with the correct arguments.

Multicast

Multicast is the simplest transport to configure, if you want the defaults:

```
eth0=mcast
```

The full configuration contains

- `mcast`
- Optionally, the MAC of the UML interface
- Optionally, the address of the multicast network
- Optionally, the port to bind to in order to send and receive multicast packets
- Optionally, the time to live (TTL) for transmitted packets

Specifying the MAC is the same with `mcast` as with all the other transports.

The address determines which multicast group the UML instance will join. You can have multiple, simultaneous, `mcast`-based virtual networks by assigning the interfaces to different multicast groups. All IP addresses within the range `224.0.0.0` to `239.255.255.255` are multicast addresses. If a value isn't specified, `239.192.168.1` will be used.

The TTL determines how far the packets will propagate.

- 0: The packet will not leave the host.
- 1: The packet will not leave the local network and will not cross a router.
- Less than 32: The packet will not leave the local organization.
- Less than 64: The packet will not leave the region.
- Less than 128: The packet will not leave the continent.
- All other values: The packet is unrestricted.

Obviously, the terms "local organization," "region," and "continent" are not well defined in terms of networking hardware, even if they are well-defined geographically, which they often aren't. It is up to the router administrators to decide whether or not their equipment is on the border of one of these areas and configure it appropriately. Once configured, the routers will drop any multicast packets that have insufficient TTLs to cross the border.

The default TTL is 1, so the packet can leave the host but not the local Ethernet.

The port should be specified if there are multiple UML instances on different multicast networks on the host so that instances on different networks are attached to different ports. The default port is 1102.

However, not all hosts provide multicast support. The `CONFIG_IP_MULTICAST` and `CONFIG_IP_MROUTE` (under "IP: Multicast router" in the kernel configuration) must be enabled. Without these, you'd see:

```
mcast_open: IP_ADD_MEMBERSHIP failed, error = 19
There appears not to be a multicast-capable network \
   interface on the host.
eth0 should be configured in order to use the multicast \
   transport.
```

uml_switch

The `daemon` transport differs from all the others in requiring a process to be started before the network will work. The process is `uml_switch`, which implements a virtual switch, as its name suggests. The simplest invocation is this:

```
host% uml_switch
uml_switch attached to unix socket '/tmp/uml.ctl'
```

The corresponding UML device configuration would be:

```
eth0=daemon
```

The defaults of both `uml_switch` and the UML driver are such that they will interoperate with each other. So, if you want a single switch on the host, the configurations above will work.

If you want multiple switches on the host, then all but one of them, and the UML instances that will connect to them, need to be configured differently. The switch and the UML instances communicate with datagrams over UNIX domain sockets. The default socket is `/tmp/uml.ctl`, as the message from the switch indicates.

A different socket can be specified with:

```
host% uml_switch -unix /tmp/uml-2.ctl
```

In order to attach to this switch, the same socket must be provided to the UML network driver:

```
eth0=daemon,,unix,/tmp/uml-2.ctl
```

`unix` specifies the type of socket to use, and the following argument specifies the socket address. At this writing, only UNIX domain sockets are supported, but this is intended to extend to allowing communication over IP sockets as well. In this case, the socket address would consist of an IP address or host name and a port number.

Some distributions (notably Debian) change the default location of the pipe used by `uml_switch` (to `/var/run/uml-utilities/uml_switch.ctl2` in Debian's case). If you use the defaults as described above and there is no connection between the UML instance and the `uml_switch` process, you need to figure out where the `uml_switch` socket is and configure the UML interface to use it.

As I mentioned earlier, `uml_switch` normally acts as a switch, so that it remembers what Ethernet MACs it has seen on what ports and transmits packets only to the port that the UML instance with the destination MAC is attached to. This saves the switch from having to forward all packets to all its instances, and it also saves the UML instances from having to receive and parse them and discard all packets not addressed to them.

`uml_switch` can be configured as a hub by using the `-hub` switch. In this case, all instances attached to it will see all packets on the network. This is sometimes useful when you want to sniff traffic between two UML instances from a third.

Normally, the switch provides an isolated virtual network, with no access to the host network. There is an option to have it connect to a preconfigured TUN/TAP device, in which case, that device will be another port on the switch, and packets will be forwarded to the host through it as appropriate. The command line would look like this:

```
uml_switch -tap switch-tap
```

`switch-tap` must be a TUN/TAP device that has already been created and configured as described in the TUN/TAP section earlier. Either bridging or routing, IP packet forwarding, and proxy arp should already be configured for this device.

The full UML device configuration contains

- daemon
- Optionally, the MAC of the UML interface
- Optionally, the socket type, which currently must be unix
- Optionally, the socket that the switch has attached to

pcap

The oddball transport, pcap, has a configuration unlike any of the others. The configuration comprises

- pcap
- The host interface to sniff
- A filter expression that determines which packets the UML interface will emit
- Up to two options from the set promisc, nopromisc, optimize, and nooptimize

The host interface may be the special string any. This will cause all host interfaces to be opened and sniffed.

The filter expression is a pcap filter program that specifies which packets should be selected.

The promisc flag determines whether libpcap will explicitly set the interface as promiscuous. The default is 1, so promisc has no effect, except for documentation. Even if nopromisc is specified, the pcap library may make the interface promiscuous for some other reason, such as being required to sniff the network.

The optimize and nooptimize flags control whether libpcap optimizes the filter expression.

Here is an example of configuring a pcap interface to emit only TCP packets to the UML interface:

```
eth0=pcap,eth0,tcp
```

This configures a second interface that would emit only non-TCP packets:

```
eth0=pcap,eth0,\!tcp
```

AN EXTENDED EXAMPLE

Now that we've covered most of what there is to know about setting up UML networking, I am going to show off some of how it works. This extended example involves multiple UML instances. To simplify their launching, I will assign them unique filesystems by giving them different COW files with the same backing file and by giving each a different `umid`. So, the command line of the first one will have this:

```
ubda=cow1,../..debian30 umid=debian1
```

and the second will have this:

```
ubda=cow2,../..debian30 umid=debian2
```

You'll probably want to do something similar as you follow along. I will be hot-plugging all network interfaces, so those won't be on the command lines.

A Multicast Network

To start, I'll run two UML instances like this. We'll begin the networking with the simplest virtual network—a default multicast network. So, let's plug a multicast device into both:

```
host% uml_mconsole debian1 config eth0=mcast
OK
host% uml_mconsole debian2 config eth0=mcast
OK
```

The kernel log of each instance shows something like this:

```
Configured mcast device: 239.192.168.1:1102-1
Netdevice 0 : mcast backend multicast address: \
    239.192.168.1:1102, TTL:1
```

You can see this by running `dmesg`, and it may also appear on the main console, depending on the distribution you are running.

Now, let's bring up both UML instances. I'm using the `192.168.1.0/24` network to keep the virtual network separate from my physical network since I intend to hook this network up to the host later. So, the first one is `192.168.1.1`:

```
UML1# ifconfig eth0 192.168.1.1 up
```

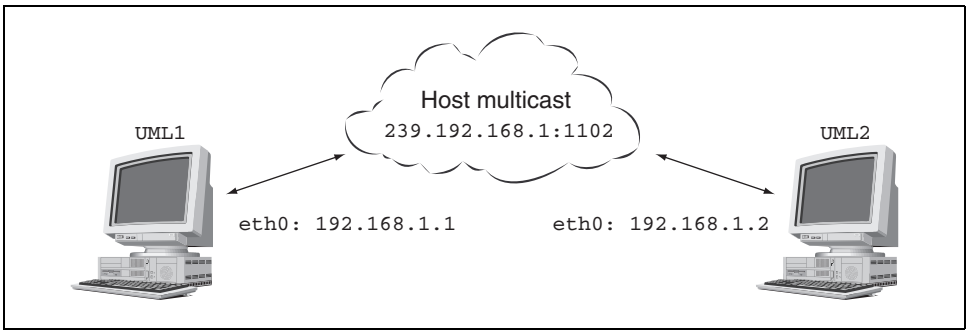


Figure 7.1 A single multicast network

and the second is `192.168.1.2`:

```
UML2# ifconfig eth0 192.168.1.2 up
```

Figure 7.1 shows what we have set up so far—two UML instances on the `192.168.1.0` network connected by the host's multicast network.

Now, check connectivity in one direction:

```
UML1# ping 192.168.1.2
PING 192.168.1.2 (192.168.1.2): 56 data bytes
64 bytes from 192.168.1.2: icmp_seq=0 ttl=64 time=0.4 ms
64 bytes from 192.168.1.2: icmp_seq=1 ttl=64 time=0.3 ms
64 bytes from 192.168.1.2: icmp_seq=2 ttl=64 time=0.3 ms

--- 192.168.1.2 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.3/0.3/0.4 ms
```

Pinging in the other direction will show something similar.

A Second Multicast Network

Now, let's set up a second, partially overlapping multicast network. This will demonstrate the use of nondefault multicast parameters. It will also make us set up some routing in order to get the two UMLs that aren't on the same network to talk to each other.

This calls for launching a third UML instance, which will get a third COW file and `umid`, with this on its command line:

```
ubda=cow3,../..debian30 umid=debian3
```

Let's put the second and third instances on the new multicast network:

```
host% uml_mconsole debian2 config eth1=mcast,,239.192.168.2,1103
OK
host% uml_mconsole debian3 config eth0=mcast,,239.192.168.2,1103
OK
```

The second instance's `eth1` and the third instance's `eth0` are now on this new network, which is defined by being on the next multicast IP and the next port. Now, we configure them on a different subnet:

```
UML2# ifconfig eth1 192.168.2.2 up
```

and

```
UML3# ifconfig eth0 192.168.2.1 up
```

Figure 7.2 shows our network so far.

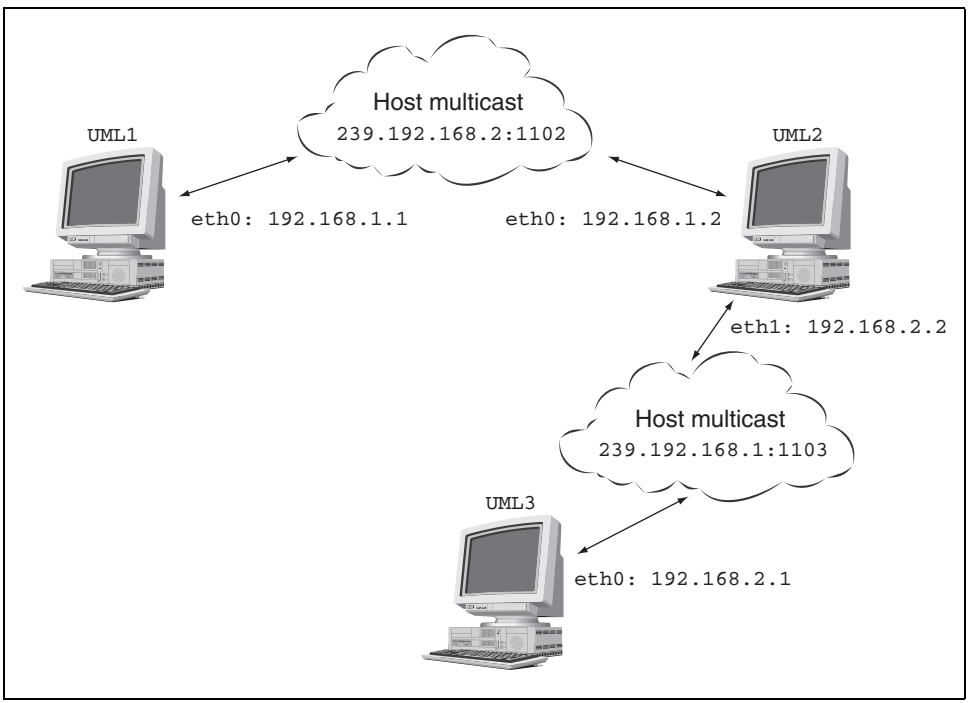


Figure 7.2 Two multicast networks

Testing connectivity here shows us what we expect:

```
UML3# ping 192.168.2.2
PING 192.168.2.2 (192.168.2.2): 56 data bytes
64 bytes from 192.168.2.2: icmp_seq=0 ttl=64 time=25.7 ms
64 bytes from 192.168.2.2: icmp_seq=1 ttl=64 time=0.4 ms

--- 192.168.2.2 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.4/13.0/25.7 ms
```

Now, let's ping the first UML from the third:

```
UML3# ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
ping: sendto: Network is unreachable
ping: wrote 192.168.1.1 64 chars, ret=-1
ping: sendto: Network is unreachable
ping: wrote 192.168.1.1 64 chars, ret=-1

--- 192.168.1.1 ping statistics ---
2 packets transmitted, 0 packets received, 100% packet loss
```

The third UML has no idea how to reach that other network. So, we need to do some routing:

```
UML3# route add -net 192.168.1.0/24 gw 192.168.2.2
```

Retrying the ping gives us different behavior—dead silence:

```
ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes

--- 192.168.1.1 ping statistics ---
4 packets transmitted, 0 packets received, 100% packet loss
```

Let's watch `tcpdump` on the second UML instance to learn what traffic it sees:

```
UML2# tcpdump -i eth1 -l -n
device eth1 entered promiscuous mode
tcpdump: listening on eth1
02:06:28.795435 192.168.2.1 > 192.168.1.1: icmp: echo \
    request (DF)
02:06:29.820703 192.168.2.1 > 192.168.1.1: icmp: echo \
    request (DF)
02:06:30.848753 192.168.2.1 > 192.168.1.1: icmp: echo \
    request (DF)
```

This is fine; ping requests are reaching the gateway between the two networks. Next, the pings should be sent out through `eth1` to the target UML instance:

```
# tcpdump -i eth0 -l -n
device eth0 entered promiscuous mode
tcpdump: listening on eth0

0 packets received by filter
0 packets dropped by kernel
device eth0 left promiscuous mode
```

They're not. This is a big clue to something we saw on the host—generally, Linux systems aren't set up as gateways and need to be told to forward packets when they can:

```
UML2# echo 1 > /proc/sys/net/ipv4/ip_forward
```

Let's look at the gateway instance's `eth0` again while the ping is running:

```
UML2# tcpdump -i eth0 -l -n
device eth0 entered promiscuous mode
tcpdump: listening on eth0
02:09:45.388353 192.168.2.1 > 192.168.1.1: icmp: echo \
    request (DF)
02:09:45.389009 192.168.2.1 > 192.168.1.1: icmp: echo \
    request (DF)
02:09:46.415998 192.168.2.1 > 192.168.1.1: icmp: echo \
    request (DF)
02:09:46.416025 192.168.2.1 > 192.168.1.1: icmp: echo \
    request (DF)
02:09:47.432823 192.168.2.1 > 192.168.1.1: icmp: echo \
    request (DF)
02:09:47.432854 192.168.2.1 > 192.168.1.1: icmp: echo \
    request (DF)

6 packets received by filter
0 packets dropped by kernel
device eth0 left promiscuous mode
```

Now pings are going out the gateway's `eth0`. We should look at the target's `eth0`:

```
UML1# tcpdump -i eth0 -l -n
device eth0 entered promiscuous mode
tcpdump: listening on eth0
02:12:36.599365 192.168.2.1 > 192.168.1.1: icmp: echo \
    request (DF)
```

```
02:12:37.631098 192.168.2.1 > 192.168.1.1: icmp: echo \
    request (DF)

2 packets received by filter
0 packets dropped by kernel
device eth0 left promiscuous mode
```

Nothing but requests here. There should be replies, but there aren't. This is the same problem we saw on the pinging UML—it doesn't know how to reply to the other network. A new route will fix this:

```
UML1# route add -net 192.168.2.0/24 gw 192.168.1.2
```

If you left the ping running, you'll see it immediately start getting replies at this point.

Now, we have three UML instances on two virtual networks, with one UML acting as a gateway between the two and with routing set up so that all three instances can communicate with each other. I'm running ping only to test connectivity, but it is fun to `ssh` between them and to fetch Web pages from one to another.

Adding a `uml_switch` Network

Let's bring the virtual switch into the action, and with it, the host. First, we'll set up a TUN/TAP device for the switch to communicate with the host:

```
host% tunctl -t switch
Set 'switch' persistent and owned by uid 500
host# ifconfig switch 192.168.3.1 up
```

Now let's run the switch using a nondefault socket:

```
host% uml_switch -unix /tmp/switch.sock -tap switch
uml_switch attached to unix socket '/tmp/switch.sock' \
    tap device 'switch'
New connection
 Addr: 86:e5:03:6f:7e:49 New port 5
```

It fakes a new connection to itself when it attaches to the TUN/TAP device. You'll see the same sorts of messages when we plug interfaces into the UML instances. I'll attach `UML1` and `UML3` to the switch:

```
host% uml_mconsole debian1 config eth1=daemon,,unix,\
    /tmp/switch.sock
```

```
OK
host% uml_mconsole debian3 config eth1=daemon,,unix,\
    /tmp/switch.sock
OK
```

You'll see a message like this in each instance:

```
Netdevice 1 : daemon backend (uml_switch version 3) - \
    unix:/tmp/switch.sock
```

Let's bring these up on the `192.168.3.0/24` network:

```
UML1# ifconfig eth1 192.168.3.2 up
UML3# ifconfig eth1 192.168.3.3 up
```

These are getting `192.168.3.2` and `192.168.3.3` because `192.168.3.1` was assigned to the TUN/TAP device.

Figure 7.3 shows our growing network.

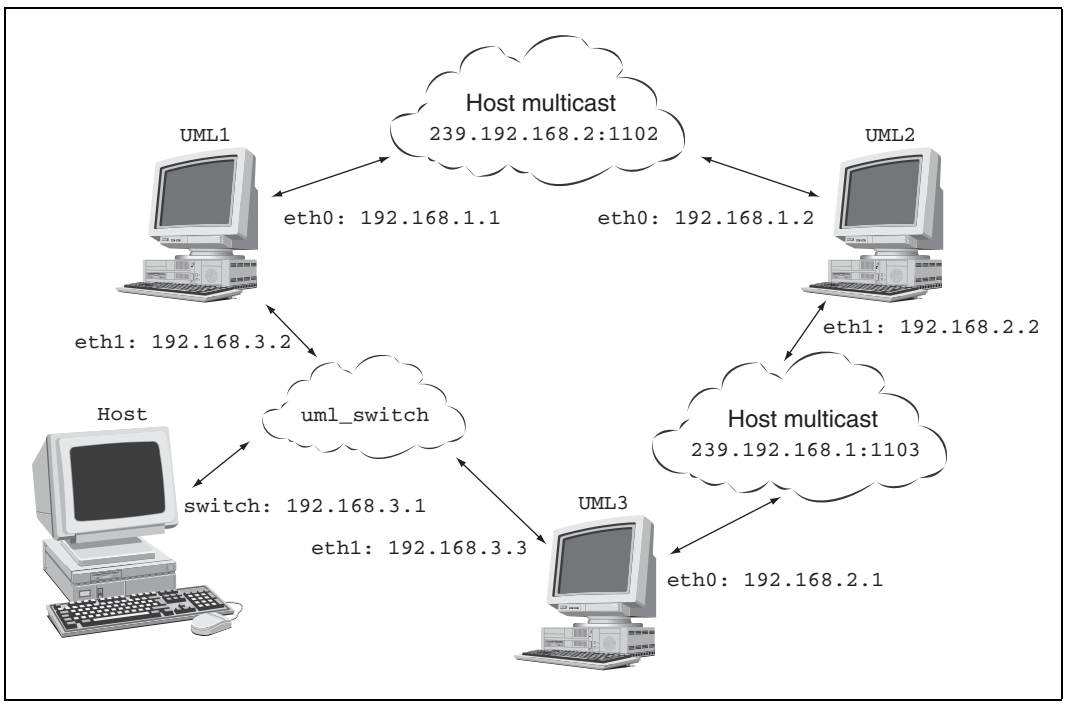


Figure 7.3 Three networks

As usual, let's check connectivity, this time through the switch:

```
UML3# ping 192.168.3.2
PING 192.168.3.1 (192.168.3.1): 56 data bytes
64 bytes from 192.168.3.1: icmp_seq=0 ttl=64 time=26.8 ms
64 bytes from 192.168.3.1: icmp_seq=1 ttl=64 time=0.2 ms
64 bytes from 192.168.3.1: icmp_seq=2 ttl=64 time=0.2 ms
64 bytes from 192.168.3.1: icmp_seq=3 ttl=64 time=0.2 ms

--- 192.168.3.1 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.2/6.8/26.8 ms
```

You'll get something similar if you ping in the other direction.

Some chatter from the switch occurs as you configure the devices and run ping:

```
New connection
New connection
 Addr: fe:fd:c0:a8:03:02 New port 7
 Addr: fe:fd:c0:a8:03:01 New port 6
 Addr: d2:a1:c9:78:bd:d7 New port 5
```

The `New connection` message is printed whenever a new device is attached to the switch, whether it's a UML instance or a host TUN/TAP interface. This is the equivalent of plugging something new into a physical switch. The `New connection` message is more or less equivalent to the link light on that port.

Messages such as `Addr: fe:fd:c0:a8:03:02 New port 7` are printed whenever the switch sees a new Ethernet MAC on a port. The address is self-explanatory. The port is the file descriptor over which the switch is communicating with the other device. Physical switches have a fixed number of ports, but this virtual switch is limited only by the number of file descriptors it can have open.

These messages will be repeated periodically as the switch does garbage collection and throws out MACs that haven't been seen recently. When the connection later wakes up, as the UML refreshes its arp cache or something similar, the switch will remember the MAC again and print another message to that effect.

At this point, we should have access to the host from the first and third UML instances through the TUN/TAP device attached to the switch:

```
UML3# ping 192.168.0.2
PING 192.168.0.2 (192.168.0.2): 56 data bytes
ping: sendto: Network is unreachable
```

```
ping: wrote 192.168.0.2 64 chars, ret=-1

--- 192.168.0.2 ping statistics ---
1 packets transmitted, 0 packets received, 100% packet loss
```

Well, not quite, but we've seen this message before, and we know what to do about it:

```
UML1# route add -net 192.168.0.0/24 gw 192.168.3.1
UML3# route add -net 192.168.0.0/24 gw 192.168.3.1
```

This is setting the gateway to the `192.168.3.0/24` network to be the `switch` TUN/TAP device. This ensures that packets to this network are addressed to the TUN/TAP device so that the switch routes them appropriately. Once they've reached the TUN/TAP device, they are on the host, and the host will deal with them as it sees fit.

At this point, the first and third UML instances have connectivity with the host:

```
UML3# ping 192.168.0.2
PING 192.168.0.2 (192.168.0.2): 56 data bytes
64 bytes from 192.168.0.2: icmp_seq=0 ttl=64 time=26.4 ms
64 bytes from 192.168.0.2: icmp_seq=1 ttl=64 time=0.2 ms
64 bytes from 192.168.0.2: icmp_seq=2 ttl=64 time=0.2 ms

--- 192.168.0.2 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.2/8.9/26.4 ms
```

The second UML instance has no access to the host because it is attached only to the two virtual networks. So, let's fix that by having it route packets through the third UML. We've done part of this already. We can finish it by enabling IP forwarding on the gateway and routing on the second UML:

```
UML3# echo 1 > /proc/sys/net/ipv4/ip_forward
UML2# route add -net 192.168.3.0/24 gw 192.168.2.1
UML2# route add -net 192.168.0.0/24 gw 192.168.2.1
```

Rather than adding two routes, it would also work to specify `192.168.2.1` as the default gateway for `UML2`.

The gateway is set to `192.168.2.1` since that's the IP address that the gateway UML has on the `192.168.2.0/24` network.

The ping doesn't work:

```
UML2# ping 192.168.0.2
PING 192.168.0.2 (192.168.0.2): 56 data bytes
```

```
--- 192.168.0.2 ping statistics ---
115 packets transmitted, 0 packets received, 100% packet loss
```

Now we have to go through the usual `tcpdump` exercise. Running `tcpdump` on the gateway's `eth0` tells us whether the requests are showing up:

```
UML3# tcpdump -i eth0 -l -n
device eth0 entered promiscuous mode
tcpdump: listening on eth0
16:37:19.634422 192.168.2.2 > 192.168.0.2: icmp: echo \
    request (DF)
16:37:20.654462 192.168.2.2 > 192.168.0.2: icmp: echo \
    request (DF)
16:37:21.683267 192.168.2.2 > 192.168.0.2: icmp: echo \
    request (DF)

3 packets received by filter
0 packets dropped by kernel
```

They are, so let's make sure they are being forwarded to `eth1` so they reach the switch:

```
UML3# tcpdump -i eth1 -l -n
device eth1 entered promiscuous mode
tcpdump: listening on eth1
16:37:24.738960 192.168.2.2 > 192.168.0.2: icmp: echo \
    request (DF)
16:37:25.768702 192.168.2.2 > 192.168.0.2: icmp: echo \
    request (DF)
16:37:26.697330 arp who-has 192.168.3.1 tell 192.168.3.3
16:37:26.697483 arp reply 192.168.3.1 is-at d2:a1:c9:78:bd:d7
16:37:26.787541 192.168.2.2 > 192.168.0.2: icmp: echo \
    request (DF)
16:37:27.818978 192.168.2.2 > 192.168.0.2: icmp: echo \
    request (DF)
16:37:28.839216 192.168.2.2 > 192.168.0.2: icmp: echo \
    request (DF)

7 packets received by filter
0 packets dropped by kernel
device eth1 left promiscuous mode
```

So far, so good. The next interface the packets should reach is the `switch` TUN/TAP interface, so let's go to the host and `tcpdump` that:

```
host# tcpdump -i switch -l -n
tcpdump: verbose output suppressed, use -v or -vv for full \
    protocol decode
listening on switch, link-type EN10MB (Ethernet), capture \
    size 96 bytes
```

```
12:44:31.851022 arp who-has 192.168.3.1 tell 192.168.3.3
12:44:32.208988 arp reply 192.168.3.1 is-at d2:a1:c9:78:bd:d7
12:44:32.209001 IP 192.168.2.2 > 192.168.0.2: icmp 64: echo \
    request seq 0
12:44:32.817880 IP 192.168.2.2 > 192.168.0.2: icmp 64: echo \
    request seq 256
12:44:33.846666 IP 192.168.2.2 > 192.168.0.2: icmp 64: echo \
    request seq 512
12:44:34.875457 IP 192.168.2.2 > 192.168.0.2: icmp 64: echo \
    request seq 768

6 packets captured
6 packets received by filter
0 packets dropped by kernel
```

Here's the problem—ping requests are reaching the host, but no replies are being sent back. The reason is that the host doesn't have a route back to the `192.168.2.0/24` network:

```
host% route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric \
    Ref    Use Iface
192.168.3.0     0.0.0.0         255.255.255.0   U     0      \
    0        0 switch
192.168.0.0     0.0.0.0         255.255.255.0   U     0      \
    0        0 eth0
169.254.0.0     0.0.0.0         255.255.0.0     U     0      \
    0        0 eth1
0.0.0.0         192.168.0.1     0.0.0.0         UG    0      \
    0        0 eth1
```

We didn't need to add a route for `192.168.3.0/24` because we got one automatically when we assigned the `192.168.3.1` address to the `switch` TUN/TAP device. We need to manually add a route for the `192.168.2.0/24` network because that's hidden behind the switch, and the host can't see it directly.

So, let's add one and see if this changes anything:

```
host# route add -net 192.168.2.0/24 gw 192.168.3.3

UML2# ping 192.168.0.2
PING 192.168.0.2 (192.168.0.2): 56 data bytes
64 bytes from 192.168.0.2: icmp_seq=0 ttl=63 time=0.5 ms
64 bytes from 192.168.0.2: icmp_seq=1 ttl=63 time=0.4 ms

--- 192.168.0.2 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.4/0.4/0.5 ms
```

For good measure, since this is the most complicated routing we have done so far, let's check pinging in the other direction:

```
ping 192.168.2.2
PING 192.168.2.2 (192.168.2.2) 56(84) bytes of data.
64 bytes from 192.168.2.2: icmp_seq=0 ttl=63 time=16.2 ms
64 bytes from 192.168.2.2: icmp_seq=1 ttl=63 time=0.369 ms

--- 192.168.2.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.369/8.295/16.221/7.926 ms, pipe 2
```

Summary of the Networking Example

We've grown a fairly complicated network during this example, so before shutting everything down, it's useful to recap what we've done.

We now have three UMLs and three two-node networks:

- `192.168.1.0/24` is a multicast network with `UML1` and `UML2`.
- `192.168.2.0/24` is a second multicast network with `UML2` and `UML3`.
- `192.168.3.0/24` is a `uml_switch` network connecting `UML1` and `UML3`, with access to the host through a TUN/TAP device.

`UML2` is acting as the gateway between the two multicast networks, `192.168.1.0/24` and `192.168.2.0/24`.

`UML3` is acting as the gateway between the `192.168.2.0/24` multicast network and the `uml_switch` network.

The gateway UMLs need to have IP forwarding enabled so they will forward packets that are not addressed to them.

The UMLs that are not directly attached to a network need a route to that network through the gateway UML. Finally, the host needs a route for any networks it is not directly attached to.

CHAPTER 8

Managing UML Instances from the Host

One of the major advantages of a virtual machine over a physical one is that it is far more manageable. It is possible to provide access to it when it has been mismanaged or misconfigured and to control it in ways that are otherwise impossible. We've seen some examples of this already, with hot-plugging of devices and querying their configurations with the `uml_mconsole` utility. This chapter covers the full suite of UML management tools.

THE MANAGEMENT CONSOLE

The UML Management Console (MConsole) support comes in two distinct pieces—a protocol and the clients that support the protocol. All we've seen so far is the default MConsole client, `uml_mconsole`. The protocol determines how `uml_mconsole` (and other clients) send requests to the MConsole driver in the UML kernel and get responses back.

We will talk more about the MConsole protocol later in this chapter. For now, it suffices to say that the protocol is dead simple, and it takes much less than a day of work to implement a basic client for it in any reasonable language such as C or a scripting language such as Perl.

MConsole clients can, and do, implement some functionality that has no counterpart in the MConsole protocol. These things are implemented locally, within the client, and will differ from client to client. The upcoming discussion refers to the `uml_mconsole` client. Later, we will talk about some other MConsole clients.

`uml_mconsole` can be used to perform a number of types of queries and control operations on a UML instance, such as:

- Reporting the version of the UML kernel
- Hot-plugging, hot-unplugging and reporting the configuration of the virtual hardware
- Doing any operation supported by the SysRq facility
- Reporting the contents of any file in the UML instance's /proc

MConsole Queries

Version

The most basic query is the `version` command, which returns the version of the kernel that the UML instance is running. The syntax is simple:

```
host% uml_mconsole debian version
OK Linux usermode 2.6.13-rc5 #29 Fri Aug 5 19:12:02 EDT 2005 \
    i686
```

This returns nearly the same output as `uname -a` would return inside the UML instance:

```
# uname -a
Linux usermode 2.6.13-rc5 #29 Fri Aug 5 19:12:02 EDT 2005 \
    i686 GNU/Linux
```

The output is composed of the following pieces:

- `Linux`—the kernel name, from `uname -s`.
- `usermode`—the node name, from `uname -n`.
- `2.6.13-rc5`—the kernel version, from `uname -r`.
- `#29 Fri Aug 5 19:12:02 EDT 2005 i686`—the kernel build information, from `uname -v`. The fact that `uname` calls this the kernel version is misleading because it's not obvious how that would differ from the kernel release. It is made up of the build number since the last `mrproper` clean of the UML kernel tree. The first

part, `#29`, indicates that this is the 29th build of this tree since it was last configured. The date and timestamp are when this UML kernel was built, and `i686` is the architecture of the build host.

You don't generally care about the version that a particular UML is running since, if you are a careful UML administrator, you should know that already. The real value of this query is that it serves as a sort of ping to the UML to check that it is alive, at least enough to respond to interrupts.

Hardware Configuration

We've seen this use of `uml_mconsole` already, when figuring out which host devices our UML consoles and serial lines had been attached to and when hot-plugging block and network devices. Even in those examples, we've seen only some of the available functionality. All of the drivers that have MConsole support, which is all of the commonly used ones, support the following operations:

- ☞ Hot-plugging
- ☞ Hot-unplugging
- ☞ Configuration request

The syntax for hot-plugging a device is:

```
config device=configuration
```

The syntax for hot-unplugging a device is:

```
remove device
```

The syntax for requesting the configuration of a device is:

```
config device
```

Unplugging a device will fail if the device is busy in some way that makes it hard or impossible to remove.

Table 8.1 summarizes the device names, syntax of the configuration data, and busyness criteria.

Halting and Rebooting a UML Instance

A UML instance can be halted or rebooted from the host using the `halt` or `reboot` commands, respectively. The kernel will run its shutdown

Table 8.1 Device Hot-Plugging, Hot-Unplugging, and Configuration

Device Type	Device Name	Configuration Syntax	Busy When
Console	conn or ssln	conn=fd:*n* conn=xterm conn=port:*n* conn=tty:*tty device* ssln=pts ssln=pty:*pty device* ssln=null ssln=none	A UML process has the console open
Network interface	ethn	ethn=tuntap,*tap device* ethn=tuntap,,*MAC*,*host IP address* ethn=ethertap,*tap device* ethn=ethertap,*tap device*,*MAC*,*host IP address* ethn=daemon,*MAC*,*unix*,*control socket* ethn=mcast,*MAC*,*host multicast IP*,*port*,*TTL* ethn=slip,*host IP address* ethn=slirp,*MAC*,*Slirp command line* ethn=pcap,*host interface*,*filter expression*,*flags*	The interface is up
Block device	ubd<*n*><*flags*>	ubd<*n*><*flags*>=*filename* ubd<*n*><*flags*>=*COW file*,*backing file*	The device is open in any way, including being mounted
Memory	mem	mem=+*memory increase* mem=-*memory decrease* mem=*memory*	Always—the amount of memory size can be decreased but can't be removed totally

procedure, which involves flushing unwritten data out to stable storage, shutting down some subsystems, and freeing host resources. However, this is a forced shutdown—the distribution's shutdown procedure will not run. So, services that are running inside the UML will not be shut down cleanly, and this may cause some problems with them on the next reboot. For example, `pid` files won't be removed, and these may prevent the initialization scripts from starting services by faking them into believing they are already running.

For a mechanism to shut down the guest more cleanly, use the MConsole `cad` command.

The `halt` and `reboot` commands are useful when the UML instance's userspace isn't responding reasonably and can't shut itself down. If the kernel is still responding to interrupts, these commands can ensure a clean kernel shutdown with the filesystems unmounted and clean.

Invoking the Ctrl-Alt-Del Handler

The distribution's Ctrl-Alt-Del handler can be invoked using the `cad` command. Unlike the `halt` and `reboot` commands, `cad` can be used to cleanly shut down the guest, including running the distribution's full shutdown procedure. This will cause all the services to be cleanly turned off, so there will be no problems as a result on the next boot.

The exact action taken by the UML instance in response to this command depends on the distribution. The `init` process is in charge of handling this, as the kernel passes the event on to it. An entry in `/etc/inittab` controls what `init` does. The most common action is to reboot, as shown in this entry:

```
# What to do when CTRL-ALT-DEL is pressed.
ca:12345:ctrlaltdel:/sbin/shutdown -t1 -a -r now
```

Before booting a UML instance on a filesystem image, it's best to decide on the preferred action for Ctrl-Alt-Del. If you want to halt the UML instance rather than reboot it, remove the `-r` from the `inittab` entry above.

Note that actually pressing the Ctrl, Alt, and Del keys on your keyboard into a UML session will not have the desired effect. If that has any effect at all, it will reboot the host since the keyboard belongs to it rather than the UML instance. Since UML doesn't have anything like a keyboard that can be made to treat a particular key combination

specially, it uses this rather more abstract method in order to obtain the same results.

Invoking the SysRq Handler

The SysRq key is another way to get the kernel to perform some action on your behalf. Generally, this is intended to debug a sick system or to shut it down somewhat cleanly when nothing else will work. Like Ctrl-Alt-Del, access to this is provided through the MConsole protocol, using the `sysrq` command to the `uml_console` client.

Use of this command requires that the UML kernel have `CONFIG_MAGIC_SYSRQ` enabled. Failure to do this will result in an error such as the following:

```
host% uml_mconsole debian sysrq p
ERR Sysrq not compiled in
```

The facility also must be turned on during boot. This is controlled by the `/proc/sys/kernel/sysrq` file (if it contains `1`, SysRq is enabled; `0` means that it is disabled) and by the `kernel.sysrq` `sysctl` parameter. Some distributions disable SysRq by default during boot. For example, Fedora Core 4 disables it with these lines in `/etc/sysctl.conf`:

```
# Controls the System Request debugging functionality of the kernel
kernel.sysrq = 0
```

You would need to change that `0` to `1` in order for the instance to support `sysrq` requests.

Any output from a `sysrq` command is returned to the MConsole client and sent to the UML kernel log and, depending on the distribution, the main console.

For example, invoking the `sysrq m` command, to dump the kernel's memory statistics, looks like this:

```
host% uml_mconsole debian sysrq m
OK SysRq : Show Memory
Mem-info:
DMA per-cpu:
cpu 0 hot: low 62, high 186, batch 31 used:174
cpu 0 cold: low 0, high 62, batch 31 used:19
Normal per-cpu: empty
HighMem per-cpu: empty
Free pages:      433128kB (0kB HighMem)
Active:1995 inactive:1157 dirty:2 writeback:0 unstable:0 \
    free:108282 slab:917 mapped:1399 pagetables:510
```

```
DMA free:433128kB min:2724kB low:3404kB high:4084kB \
    active:7980kB inactive:4628kB present:463768kB pages_scanned:0 \
    all_unreclaimable? no
lowmem_reserve[]: 0 0 0
Normal free:0kB min:0kB low:0kB high:0kB active:0kB inactive:0kB \
    present:0kB pages_scanned:0 all_unreclaimable? no
lowmem_reserve[]: 0 0 0
HighMem free:0kB min:128kB low:160kB high:192kB active:0kB \
    inactive:0kB present:0kB pages_scanned:0 all_unreclaimable? no
lowmem_reserve[]: 0 0 0
DMA: 2*4kB 2*8kB 5*16kB 2*32kB 1*64kB 0*128kB 1*256kB 1*512kB \
    0*1024kB 1*2048kB 105*4096kB = 433128kB
Normal: empty
HighMem: empty
Swap cache: add 0, delete 0, find 0/0, race 0+0
Free swap  = 0kB
Total swap = 0kB
Free swap:            0kB
115942 pages of RAM
0 pages of HIGHMEM
3201 reserved pages
5206 pages shared
0 pages swap cached
```

The output will also appear in the kernel log. This and the output of many of the other commands are dumps of internal kernel state and aren't meant to be analyzed by normal users. This information is useful when a UML instance is in sufficiently bad shape as to require internal kernel information for a diagnosis.

Table 8.2 summarizes the `sysrq` commands and what they do.

Table 8.2 `sysrq` Commands

Command	Function
`0–9`	Set the log level: `0` is the lowest, `9` is the highest. Any messages with a priority at least as high as this are logged.
`b`	Reboot—UML cleanup, but no kernel or userspace cleanup.
`e`	Terminate all tasks by sending a `SIGTERM`.
`f`	Simulate an out-of-memory condition, forcing a process to be killed to reclaim its memory.
`i`	Kill all tasks by sending a `SIGKILL`.
`m`	Show memory usage.

(*continues*)

Table 8.2 `sysrq` Commands (*continued*)

Command	Function
`n`	Make all real-time tasks become normal round-robin tasks.
`p`	Dump the registers and stack of the current task.
`s`	Sync dirty data to stable storage.
`t`	Show the state, stack trace, and registers for all tasks on the system.
`u`	Remount all filesystems read-only.

Stopping and Restarting a UML Instance

MConsole provides the ability to stop and continue a UML instance. When it is stopped in this manner, it is doing nothing but interpreting MConsole commands. Nothing else is happening. Processes aren't running and nothing else in the kernel is running, including device interrupts. This state will persist until the MConsole driver receives the command to continue.

The main use of this functionality is to perform an online backup by stopping the instance, having it write out all unwritten file data to disk, copying the now-clean filesystem to someplace safe, and continuing the UML instance.

The full procedure looks like this:

```
host% uml_mconsole debian stop
OK
host% uml_mconsole debian sysrq s
OK SysRq : Emergency Sync
host% cp --sparse=always cow save-cow
host% uml_mconsole debian go
OK
```

The `sysrq s` command performs the synchronization of unwritten data to disk, resulting in this output to the kernel log:

```
SysRq : Emergency Sync
Emergency Sync complete
```

In this example, I just copied the UML instance's COW file to a file in the same directory. Obviously, a rather more organized backup system would be advisable on a serious UML host. Such a system would

keep track of what UML instances had been backed up, when they were last backed up, and the location of the backups.

I used the `--sparse=always` switch to `cp` in order to preserve the sparseness of the COW file. This is important for speeding up the copy and for conserving disk space. Without it, all unoccupied blocks in the COW file will be filled with zeros on disk in the copy. This will result in those zero-filled blocks occupying host page cache for a while and will require that they all be written out to disk at some point. Keeping the copy sparse ensures that unoccupied blocks don't become instantiated, so they don't occupy memory before being written to disk, I/O bandwidth while being written, and disk space afterward.

The copy took just under three seconds on my laptop, making this a very quick way to get a backup of the UML instance's data, causing almost no downtime.

This works particularly well with COW files since backing up a full filesystem image would take noticeably longer and consume more bandwidth while writing the copy out to disk.

Logging to the UML Instance's Kernel Log

The `log` command enables arbitrary text to be inserted into the UML instance's kernel log. This was written in order to allow administrators of UML honeypots to overwrite the preexisting, UML-specific kernel log with a log that looks like it came from a physical machine. Since the purpose of a virtual honeypot is to pretend to be a physical machine, it is important that there be no easy ways for an intruder to discern that it is a virtual machine. Examining the kernel log is a fairly easy way to tell what sort of machine you're on because it contains a great deal of information about the hardware.

Since the kernel log has a fixed size, logging enough data will cause any previous data to be lost, and the kernel log will contain only what you logged with MConsole.

There are probably limited uses of this ability outside of honeypots, but it could be useful in a situation where events inside a UML instance need to be coordinated with events outside. If the kernel log of the UML instance is the official record of the procedure, the `log` MConsole command can be used to inject outside messages so that the kernel log contains all relevant information in chronological order.

The `uml_mconsole` client has a `log -f <file>` command that will log the contents of the given file to the UML instance's kernel log.

Examining the UML Instance's `/proc`

You can use the MConsole `proc` command to examine the contents of any file within the UML's `/proc`. This is useful for debugging a sick UML instance, as well as for gathering performance data from the host.

This output gets returned to the MConsole client, as seen here:

```
host% uml_mconsole debian proc meminfo
OK MemTotal:     450684 kB
MemFree:         434608 kB
Buffers:            724 kB
Cached:            8180 kB
SwapCached:           0 kB
Active:            7440 kB
Inactive:          3724 kB
HighTotal:            0 kB
HighFree:             0 kB
LowTotal:        450684 kB
LowFree:         434608 kB
SwapTotal:            0 kB
SwapFree:             0 kB
Dirty:                0 kB
Writeback:            0 kB
Mapped:            5632 kB
Slab:              3648 kB
CommitLimit:     225340 kB
Committed_AS:     10820 kB
PageTables:        2016 kB
VmallocTotal:   2526192 kB
VmallocUsed:         24 kB
VmallocChunk:   2526168 kB
```

This sort of thing would be useful in monitoring the memory consumption of the UML instances running on a host. Its intended purpose is to allow a daemon on the host to monitor memory pressure inside the UML instances and on the host, and to use memory hot-plug to shift memory between instances in order to optimize use of the host's physical memory. At this writing, this is a work in progress, as support in the host kernel is needed in order to make this work. A prototype of the host functionality has recently been implemented. However, it is unclear whether this interface will survive or when this ability will appear in the mainline kernel.

Currently, this command can be used only for `/proc` files that you know exist. In other words, it doesn't work on directories, meaning you can't use it to discover what processes exist inside the UML and get their statistics.

Forcing a Thread into Context

The MConsole `stack` command is a bit of a misnomer. While it does do what it suggests, its real purpose is somewhat different. Sending this command to a UML instance will cause it to dump the stack of the given process:

```
host% uml_mconsole debian stack 1
OK EIP: 0073:[<400ecdb2>] CPU: 0 Not tainted ESP: 007b:bf903da0 \
    EFLAGS: 00000246
    Not tainted
EAX: ffffffda EBX: 0000000b ECX: bf903de0 EDX: 00000000
ESI: 00000000 EDI: bf903dd8 EBP: bf903dd8 DS: 007b ES: 007b
15b07a20:  [<080721bd>] show_regs+0xd1/0xd8
15b07a40:  [<0805997d>] _switch_to+0x6d/0x9c
15b07a80:  [<081b3371>] schedule+0x2e5/0x574
15b07ae0:  [<081b3d37>] schedule_timeout+0x4f/0xbc
15b07b20:  [<080c1a85>] do_select+0x255/0x2e4
15b07ba0:  [<080c1d75>] sys_select+0x231/0x43c
15b07c20:  [<0805f591>] handle_syscall+0xa9/0xc8
15b07c80:  [<0805e65a>] userspace+0x1ae/0x2bc
15b07ce0:  [<0805f11e>] new_thread_handler+0xaa/0xbc
15b07d20:  [<00dde420>] 0xdde420
```

The process ID I gave was one internal to UML, that of the `init` process. If you don't know what processes are running on the system, you can get a list of them with `sysrq t`:

```
host% uml_mconsole debian2 sysrq t
```

The output looks in part like this:

```
apache        S 00000246     0   253    238                \
    252 (NOTLB)
14f03b10 00000001 bfffe0cc 0013a517 00000246 14f03b10 \
    000021a0 144d8000
       14e75860 1448c740 144db98c 144db8d4 0805e941 00000001 \
    12002000 00000000
       00000033 00000025 bfffe0cc 0013a517 00000246 bfacf178 \
    0000007b 0013a517 Call Trace:
144db990:  [<0805f039>] switch_to_skas+0x39/0x74
144db9c0:  [<08059955>] _switch_to+0x45/0x9c
144dba00:  [<081b38a1>] schedule+0x2e5/0x574
144dba60:  [<081b4289>] schedule_timeout+0x71/0xbc
144dba90:  [<08187cf9>] inet_csk_wait_for_connect+0xc5/0x10c
144dbad0:  [<08187de1>] inet_csk_accept+0xa1/0x150
144dbb00:  [<081a529a>] inet_accept+0x26/0xa4
144dbb30:  [<08165e10>] sys_accept+0x80/0x12c
144dbbf0:  [<081667dd>] sys_socketcall+0xbd/0x1c4
144dbc30:  [<0805f591>] handle_syscall+0xa9/0xc8
```

```
144dbc90:  [<0805e65a>] userspace+0x1ae/0x2bc
144dbcf0:  [<0805f1e0>] fork_handler+0x84/0x9c
144dbd20:  [<00826420>] 0x826420
```

As with `sysrq` output, this will also be recorded in the kernel message log.

This tells us we have an `apache` process whose process ID is 253. This also dumps the stack of every process on the system in exactly the same format as with the `stack` command.

So why have the `stack` command when `sysrq t` gives us the same information and more? The reason is that the real intent of this command is to temporarily wake up a particular thread within the UML instance so it will hit a breakpoint, letting you examine the thread with the debugger.

To do this, you must have the UML instance running under `gdb`, either from the start or by attaching to the instance later. You put a breakpoint on the `show_regs()` call in `_switch_to`, which is currently in `arch/um/kernel/process_kern.c`:

```
do {
        current->thread.saved_task = NULL ;
        CHOOSE_MODE_PROC(switch_to_tt, switch_to_skas, prev, \
    next);
        if(current->thread.saved_task)
                show_regs(&(current->thread.regs));
        next= current->thread.saved_task;
        prev= current;
} while(current->thread.saved_task);
```

This call is what actually dumps out the stack. But since you put a breakpoint there, `gdb` will stop before that actually happens. At this point, `gdb` is sitting at the breakpoint with the desired thread in context. You can now examine that thread in detail. Obviously this is not useful for the average UML user. However, it is immensely useful for someone doing kernel development with UML who is seeing processes hang. Most commonly, it's a deadlock of some sort, and figuring out exactly what threads are holding what locks, and why is essential to debugging it. Waking up a particular thread and making it hit a breakpoint is very helpful.

This sort of thing had been possible in `tt` mode for a long time, but not in `skas` mode until this functionality was implemented. In `tt` mode, every UML process or thread has a corresponding host process, and that process includes the UML kernel stack. This makes it possible to see the kernel stack for a process by attaching `gdb` to the proper host process.

In skas mode, this is not the case. The UML kernel runs entirely within a single process, using longjmp to switch between kernel stacks on context switches. gdb can't easily access the kernel stacks of processes that are not currently running. Temporarily waking up a thread of interest and making it hit a breakpoint is a simple way to fix this problem.

Sending an Interrupt to a UML Instance

The int command is implemented locally within uml_mconsole. It sends an interrupt signal (SIGINT) to the UML instance that uml_mconsole is communicating with. It operates by reading the instance's pid file and sending the signal to that process.

Normally, that instance will be running under gdb, in which case, the interrupt will cause the UML instance to stop running and return control to gdb. At that point, you can use gdb to examine the instance as you would any other process.

If the UML instance is not running under gdb, the signal will cause it to shut down.

Getting Help

Finally, there is a help command, which will display a synopsis of the available MConsole commands:

```
host% uml_mconsole debian help
OK Commands:
    version - Get kernel version
    help - Print this message
    halt - Halt UML
    reboot - Reboot UML
    config <dev>=<config> - Add a new device to UML;
        same syntax as command line
    config <dev> - Query the configuration of a device
    remove <dev> - Remove a device from UML
    sysrq <letter> - Performs the SysRq action controlled by \
        the letter
    cad - invoke the Ctrl-Alt-Del handler
    stop - pause the UML; it will do nothing until it receives \
        a 'go'
    go - continue the UML after a 'stop'
    log <string> - make UML enter <string> into the kernel \
        log
    proc <file> - returns the contents of the UML's \
        /proc/<file>
```

```
Additional local mconsole commands:
    quit - Quit mconsole
    switch <socket-name> - Switch control to the given \
        machine
    log -f <filename> - use contents of <filename> as \
        UML log messages
    mconsole-version - version of this mconsole program
```

The first section shows requests supported by the MConsole driver within the UML kernel. These are available to all clients, although perhaps in a different form. The second section lists the commands supported locally by this particular client, which may not be available in others.

There is no predetermined set of requests within the MConsole protocol. Requests are defined by the driver and can be added or removed without changing the protocol. This provides a degree of separation between the client and the UML kernel—the kernel can add more commands and existing clients will be able to use them.

This separation can be seen in the help message above. The first section was provided by the UML kernel and merely printed out by the `uml_mconsole` client. When a new request is added to the driver, it will be added to the kernel's help string, and it will automatically appear in the help text printed by the `uml_mconsole` client.

Running Commands within the UML Instance

An oft-requested MConsole feature is the ability to run an arbitrary command within a UML instance. I oppose this on the basis that there are perfectly good ways to run commands inside a UML instance, for example, by logging in and executing a command within a shell.

A design ethic in the Linux kernel community holds that only things that need to be done in the kernel should be done there. The existence of other ways to run commands within a UML is proof that this functionality doesn't need to be in the kernel. Thus, I have refused to implement this or to merge other people's implementations.

Nevertheless, a patch implementing this ability does exist, and it has a following in the UML community. With a suitably patched UML, it works like this:

```
host% uml_mconsole debian exec "ps uax > /tmp/x"
OK The command has been started successfully.
```

The command's output isn't returned back to the MConsole client because it would be complicated to start a process from a kernel thread, capture its output, and return it to the outside. Thus, if you

want the output, you need to save it someplace, as I did above by redirecting the output to `/tmp/x`, and then retrieve it.

This is convenient, but I would claim that, with a little foresight on behalf of the host administrator, essentially the same thing can be done in other ways.

The most straightforward way to do this is simply to log in to the UML and run the commands you need. Some people make a couple of common objections to this method.

- A login is hard because it's tough to parse the login and password prompts and respond to them robustly.
- A login modifies things such as network counters and `wtmp` and `utmp` entries, which some people would prefer to see unchanged.
- MConsole `exec` is harder for the UML user to disable, purposefully or not, than a login.

I have what I believe to be solid answers to these objections. First, with an `ssh` key in the appropriate place in the UML filesystem, parsing the login and password prompts is unnecessary because there aren't any.

Second, logging in over a UML console doesn't modify any network counters. Dedicating this console to the admin makes it possible to have a root shell permanently running on it, making even `ssh` unnecessary, and also not modifying the `wtmp` or `utmp` files because there's no login.

Third, I don't think any of the alternatives are any more robust against disabling or manipulation than MConsole `exec`. An `ssh` login can be disabled by the UML root user deleting the `ssh` key. The console with a permanent root shell can be disabled by editing the UML instance's `/etc/inittab`. But MConsole `exec` can be disabled by moving or replacing the commands that the host administrator will run.

The desire for something like MConsole `exec` is a legitimate one, and all of the current solutions have limitations. I believe that the long-term solution may be something like allowing a host process to migrate into the UML instance, allowing it to do its work inside that environment. In this case, the UML environment wouldn't be as opaque to the host as it is now. It would be possible to create a process on the host, guaranteeing that it is running the correct executable, and then move it into the UML instance. It would then see the UML filesystem, devices, processes, and so on and operate in that environment. However, it would retain ties to the host environment. For example, it would

retain the file descriptors opened on the host before the migration, and it would be able to accept input and send output through them. Something like this, which can be seen as a limited form of clustering, seems to me to suffice and has none of the limitations of the other solutions.

The `uml_mconsole` Client

We've already seen a great deal of the `uml_mconsole` client, as it has been used to illustrate all of the MConsole discussion to date. However, there are some aspects we haven't seen yet.

We have seen the format of the output of a successful request, such as this:

```
host% uml_mconsole debian version

OK Linux usermode 2.6.13-rc5 #29 Fri Aug 5 19:12:02 EDT 2005 \
    i686
```

It always starts with `OK` or `ERR` to simplify automating the determination of whether the request succeeded or failed. This is how a failure looks:

```
host% uml_mconsole debian remove ubda
ERR Device is currently open
```

Because the `/dev/ubda` device is currently mounted, the removal request fails and is reported with `ERR` followed by the human-readable error message.

An important part of `uml_mconsole` that we haven't seen is its internal command line. Every example I have used so far has had the command in the argument list. However, if you ran `uml_mconsole` with just one argument, a `umid` for a running UML instance, you would see something like this:

```
host% uml_mconsole debian
(debian)
```

At this point, you can run any MConsole command. The prompt tells you which UML instance your request will be sent to.

You can change which UML you are talking to by using the `switch` local command:

```
(debian) switch new-debian
Switched to 'new-debian'
(new-debian)
```

At this point, all requests will go to the `new-debian` UML instance.

Finally, there is a local command that will tell you what version of the client you are running:

```
(new-debian) mconsole-version
uml_mconsole client version 2
```

Whether you're using `uml_mconsole` in single-shot mode, with the command on the `uml_mconsole` command line, or you're using its internal command line, commands intended for the UML MConsole driver are generally passed through unchanged. A single-shot command is formed by concatenating the command-line argument vector into a single string with spaces between the arguments.

The one exception to this is for commands that take filenames as arguments. Currently, there is only one time this happens—when indicating the files that a block device will be attached to. These may be specified as relative paths, which can cause problems when the UML instance and `uml_mconsole` process don't have the same working directory. A path relative to the `uml_mconsole` process working directory will not be successfully opened by the UML instance from its working directory. To avoid this problem, `uml_mconsole` makes such paths absolute before passing the request to the UML instance.

The MConsole Protocol

The MConsole protocol, between the MConsole client and the MConsole driver in the UML kernel, is the glue that makes the whole thing work. I'm going to describe the protocol in enough detail that someone, sufficiently motivated, could implement a client. This won't take too long since the protocol is extremely simple.

As with any client-server protocol, the client forms a request, sends it to the server, and at some later point gets a reply from the server.

The request structure contains

- A magic number
- A version number
- The request
- The request length

In C, it looks like this:

```
#define MCONSOLE_MAGIC (0xcafebabe)
#define MCONSOLE_MAX_DATA (512)
#define MCONSOLE_VERSION 2

struct mconsole_request {
        u32 magic;
        u32 version;
        u32 len;
        char data[MCONSOLE_MAX_DATA];
};
```

The command goes into the data field as a string consisting of space-separated words—exactly what the `uml_mconsole` client reads from its command line. The length of the command, the index of the NULL-terminator, is put in the `len` field.

In Perl, forming a request looks like this:

```
my $MCONSOLE_MAGIC = 0xcafebabe;
my $MCONSOLE_MAX_DATA = 512;
my $MCONSOLE_VERSION = 2;
my $msg = pack("LiiA*", $MCONSOLE_MAGIC, $MCONSOLE_VERSION, \
    length($cmd),
               $cmd);
```

Once the request is formed, it must be sent to the server in the UML MConsole driver over a UNIX domain socket created by the driver. On boot, UML creates a subdirectory for instance-specific data, such as this socket. The subdirectory has the same name as the UML instance's `umid`, and its parent directory is the `umid` directory, which defaults to `~/.uml`. So, a UML instance with a `umid` of `debian` will have its MConsole socket created at `~/.uml/debian/mconsole`. The `umid` directory can be changed with the `umid=` switch on the UML command line.

The request is sent as a datagram to the MConsole socket, where it is received by the driver and handled. The response will come back over the same socket in a form very similar to the request:

```
struct mconsole_reply {
        u32 err;
        u32 more;
        u32 len;
        char data[MCONSOLE_MAX_DATA];
};
```

`err` is the error indicator—if it is zero, the request succeeded and data contains the reply. If it is nonzero, there was some sort of error, and the data contains the error message.

`more` indicates that the reply is too large to fit into a single reply, so more reply packets are coming. The final reply packet will have a `more` field of zero.

As with the request, the `len` field contains the length of the data in this packet.

In Perl, the response looks like this:

```
($err, $more, $len, $data) = unpack("iiiA*", $data);
```

where `$data` is the packet read from the socket.

The use of a UNIX domain socket, as opposed to a normal IP socket, is intentional. An IP socket would allow an MConsole client to control a UML instance on a different host. However, allowing this would require some sort of authentication mechanism built into the protocol, as this would enable anyone on the network to connect to a UML instance and start controlling it.

The use of a UNIX domain socket adds two layers of protection. First, it is accessible only on the UML instance's host, so any users must be logged in to the host. Second, UNIX domain sockets are protected by the normal Linux file permission system, so that access to it can be controlled by setting the permissions appropriately.

Rather than invent another authentication and authorization mechanism, the use of UNIX domain sockets forces the use of existing mechanisms. If remote access to the UML instance is required, executing the `uml_mconsole` command over `ssh` will use `ssh` authentication. Similarly, the file permissions on the socket make up the MConsole authorization mechanism.

The MConsole Perl Library

As is evident from the Perl snippets just shown, `uml_mconsole` is not the only MConsole client in existence. There is a Perl client that is really a library, not a standalone utility. Part of the UML test suite, it is used to reconfigure UML instances according to the needs of the tests.

In contrast to the `uml_mconsole` client, this library has a method for every MConsole request, rather than simply passing commands through to the server unchanged.

Requests Handled in Process and Interrupt Contexts

There is a subtlety in how MConsole requests are handled inside the driver that can affect whether a sick UML will respond to them. Some requests must be handled in a process context, rather than in the MConsole interrupt handler. Any request that could potentially sleep must be handled in a process context. This includes `config` and `remove`, `halt` and `reboot`, and `proc`. These all call Linux kernel functions, which for one reason or another might sleep and thus can't be called from an interrupt handler.

These requests are queued by the interrupt handler, and a special worker thread takes care of them at some later time. If the UML is sufficiently sick that it can't switch to the worker thread, such as if it is stuck handling interrupts, or the worker thread can't run, then these requests will never run, and the MConsole client will never get a reply.

In this case, another mechanism is needed to bring down the UML instance in a semicontrolled manner. For this, see the final section of this chapter, on controlling UML instances with signals from the host.

MConsole Notifications

So far, we have seen MConsole traffic initiated only by clients. However, sometimes the server in the UML kernel can initiate traffic. A notification mechanism in the MConsole protocol allows asynchronous events in the UML instance to cause a message to be sent to a client on the host. Messages can result from the following events.

- The UML instance has booted far enough that it can handle MConsole requests. This is to prevent races where a UML instance is booted and an MConsole client tries to send requests to it before it has set up its MConsole socket. This notification is sent once the socket is initialized and includes the location of the socket. When this notification is received, the MConsole driver is running and can receive requests.
- The UML instance is panicking. The panic message is included in the notification.
- The UML instance has hung. This one is unusual in not being generated by the UML kernel itself. Since the UML is not responding to anything, it is likely unable to diagnose its own hang and send this notification. Rather, the message is generated by an external

process on the host that is communicating with the UML `harddog` driver, which implements something like a hardware watchdog. If this process doesn't receive a message from the `harddog` driver every minute, and it has been told to generate a hang notification, it will construct the notification and send it. At that point, it is up to the client to decide what to do with the hung UML instance.

☞ A UML user has generated a notification. This is done by writing to the `/proc/mconsole` file in the UML instance. This file is created when the UML instance has been told on the command line to generate notifications.

The client that receives these notifications may be a different client than you would use to control the UML. In fact, the `uml_mconsole` client is incapable of receiving MConsole notifications. In order to generate notifications, a switch on the UML command line is needed to specify the UNIX socket to which the instance will send notifications. This argument on the command line specifies the file `/tmp/notify`, which must already exist, as the notification socket for this UML instance:

```
mconsole=notify:/tmp/notify
```

Using this small Perl script, we can see how notifications come back from the UML instance:

```
use UML::MConsole;
use Socket;
use strict;

my $sock = "/tmp/notify";

!defined(socket(SOCK, AF_UNIX, SOCK_DGRAM, 0)) and
    die "socket failed : $!\n";

!defined(bind(\*SOCK, sockaddr_un($sock))) and
    die "UML::new - bind failed : $!\n";

while(1){
    my ($type, $data) = UML::MConsole->read_notify(\*SOCK, undef);

    print "Notification type = \"$type\", data = \"$data\"\n";
}
```

By first running this script and then starting the UML instance with the switch given above, we can see notifications being generated.

The first one is the socket notification telling us that the MConsole request socket is ready:

```
Notification type = "socket", data = \
   "/home/jdike/.uml/debian/mconsole"
```

Once the instance has booted, we can log in and send messages to the host through the `/proc/mconsole` file:

```
UML# echo "here is a user notification" > /proc/mconsole
```

This results in the following output from the notification client:

```
Notification type = "user notification", \
data = "here is a user notification"
```

These notifications all have a role to play in an automated UML hosting environment. The socket notification tells when a UML instance is booted enough to be controllable with an MConsole client. When this message is received, the instance can be marked as being active and the control tools told of the location of the MConsole socket.

The panic and hang notifications are needed in order to know when the UML should be restarted, in the case of a panic, or forcibly killed and then restarted, in the case of a hang.

The user notifications have uses that are limited only by the imagination of the administrator. I implemented them for the benefit of workloads running inside a UML instance that need to send status messages to the host. In this scenario, whenever some milestone is reached or some significant event occurs, a user notification would be sent to the client on the host that is keeping track of the workload's progress.

You could also imagine having a tool such as a log watcher or intrusion detection system sending messages to the host through `/proc/mconsole` whenever an event of interest happens. A hosting provider could also use this ability to allow users to make requests from inside the UML instance.

Controlling a UML Instance with Signals

So far, I've described the civilized ways to control UML instances from the host. However, sometimes an instance isn't healthy enough to cooperate with these mechanisms. For these cases, some limited amount of control is available by sending the instance a signal.

To send a UML instance a signal, you first need to know which process ID to send it to. A UML instance is comprised of a number of threads, so the choice is not obvious. Also, when the host has a number of instances, there is a real chance of misreading the output of `ps` and hitting the wrong UML instance.

To solve this problem, a UML instance writes the process ID of its main thread into the `pid` file in its `umid` directory. This thread is the one responsible for handling the signals that can be used for this last-ditch control. Given a `umid`, sending a signal to the corresponding instance is done like this:

```
kill -TERM `cat ~/.uml/debian/pid`
```

When this main thread receives `SIGINT`, `SIGTERM`, or `SIGHUP`, it will run the UML-specific parts of the shutdown process. This will have the same effect as the MConsole `halt` or `sysrq b` requests. No userspace or kernel cleanup will happen. Only the host resources that have been allocated by UML will be released. The UML instance's filesystems will be dirty and need either an `fsck` or a journal replay.

CHAPTER 9

Host Setup for a Small UML Server

After having talked about UML almost exclusively so far, we will now talk about the host. This chapter and the next will cover setting up and running a secure, well-performing UML server. First we will talk about running a small UML server, where the UML instances will be controlled by fairly trusted people, such as the host administrator or others with logins on the host. Thus, we won't need the same level of security as on a large UML server with unknown, untrusted people inside the UML instances. We will have a basic level of security, where nothing can break out of a UML instance onto the host. We won't be particularly paranoid about whether network traffic from the UMLs is originating from the expected IP addresses or whether there is too much of it. Similarly, we will talk about getting good performance from the UML instances, but we won't try to squeeze every bit of UML hosting capacity from the host.

All of these things, which a large UML hosting provider cares about more than a casual in-house UML user does, will be discussed in the next chapter. There, we will cover tougher security measures, such as how to protect the host even if a user does somehow manage to break out of a UML instance and how to ensure that UML instances

are not spoofing IP addresses or sending out unreasonably large amounts of traffic. We will also discuss how to log resource usage, such as network traffic, in that chapter. But first, let's cover what more casual users want to know.

HOST KERNEL VERSION

Technically, UML will run on any x86 host kernel from a stable series (Linux kernel versions 2.2, 2.4, or 2.6) since 2.2.15. However, the 2.2 kernel is of historic interest only—if you have such a machine that you are going run UML instances on, you should upgrade. The 2.4 and 2.6 kernels make good hosts, but 2.6 is preferred. UML will run on any x86_64 (Opteron/AMD64 or Intel EM64T) host, which is a newer architecture and has had the necessary basic support since the beginning. However, x86_64 hosts are stable only on hosts running 2.6.12 or later. On S/390, a fairly new 2.6 host kernel is required because of bugs that were found and fixed during the UML port to that architecture.

UML makes use of the AIO and `O_DIRECT` facilities in the 2.6 kernels for better performance and lower memory consumption. AIO is kernel-level asynchronous I/O, where a number of I/O requests can be issued at once, and the process that issued them can receive notifications asynchronously when they finish. The kernel issues the notifications when the data is available, and the order in which that happens may not be related to the order in which they are issued.

The alternative, which is necessary on earlier kernels, is to either make normal read and write system calls, which are synchronous, and make the process sleep until the operation finishes, or to dedicate a thread (or multiple threads) to I/O operations. Doing I/O synchronously allows only one operation to be pending at any given time. Doing I/O asynchronously by having a separate thread do synchronous I/O at least allows the process to do other work while the operation is pending. On the other hand, only one operation can be pending for each such I/O thread, and the process must context-switch back and forth from these threads and communicate with them as operations are issued and completed. Having one thread for each pending I/O operation is hugely wasteful.

`glibc` has AIO support in all kernels, even those without AIO support, and it implements this using threads, potentially one thread per outstanding I/O request. UML, on such hosts, emulates AIO in a

similar way. It creates a single thread, allowing one I/O request to be pending at a time.

The AIO facility present in the 2.6 kernel series allows processes to do true AIO. UML uses this by having a separate thread handle all I/O requests, but now, this thread can have many operations pending at once. It issues operations to the host and waits for them to finish. As they finish, the thread interrupts the main UML kernel so that it can finish the operations and wake up anything that was waiting for them.

This allows better I/O performance because more parallel I/O is possible, which allows data to be available earlier than if only one I/O request can be pending.

`O_DIRECT` allows a process to ask that an I/O request be done directly to and from its own address space without being cached in the kernel, as shown in Figure 9.1. At first glance, the lack of caching would seem to hurt performance. If a page of data is read twice with `O_DIRECT` enabled, it will be read from disk twice, rather than the second request being satisfied from the kernel's page cache. Similarly, write requests will go straight to disk, and the request won't be considered finished until the data is on the disk.

However, `O_DIRECT` is intended for specialized applications that implement their own caching and use AIO. For an application like this, using `O_DIRECT` can improve performance and lower its total memory requirements, including memory allocated on its behalf inside the kernel. UML is such an application, and use of `O_DIRECT` actually makes it behave more like a native kernel.

A native kernel must wait for the disk when it writes data, and there is no caching level below it (except perhaps for the on-disk cache), so if it reads data, it must again wait for the disk. This is exactly the behavior imposed on a process when it uses `O_DIRECT` I/O.

The elimination of the caching of data at the host kernel level means that the only copy of the data is inside the UML instance that read it. So, this eliminates one copy of the data, reducing the memory consumption of the host. Eliminating this copy also improves I/O latency, making the data available earlier than if it was read into the host's page cache and then copied (or mapped) into the UML instance's address space.

For these reasons, for x86 hosts, a 2.6 host kernel is preferable to 2.4. As I pointed out earlier, running UML on x86_64 or S/390 hosts requires a 2.6 host because of host bugs that were fixed fairly recently.

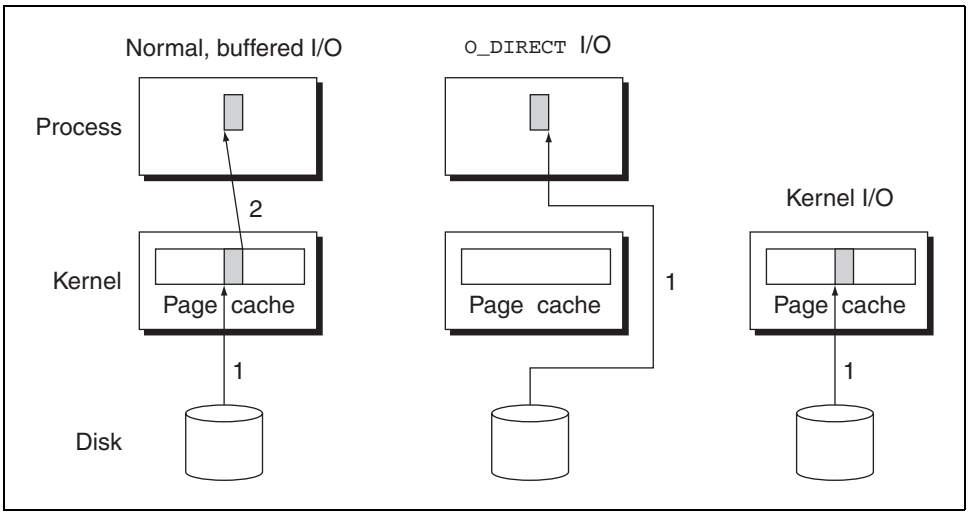


Figure 9.1 `O_DIRECT` I/O compared to buffered I/O. When a process does a buffered read, the data is first read from disk and stored in the kernel's page cache. Then it is copied into the address space of the process that initiated the read. Buffering it in the page cache provides faster access to the data if it is needed again. However, the data is copied and stored twice. When a process performs an `O_DIRECT` read, the data is read directly from the disk into the process address space. This eliminates the extra copy operation and the extra memory consumption caused by a buffered read. However, if another process needs the data, it must be read from disk rather than simply copied from the kernel's page cache. The figure also shows a read done by the kernel for its own purposes, to compare it to the `O_DIRECT` read. In both cases, the data is read directly from disk and stored only once. When the process doing the `O_DIRECT` read is UML reading data into its own page cache, the two cases are identical.

UML Execution Modes

Traditionally, UML has had two modes of operation, one for unmodified hosts and one for hosts that have been patched with what is known as the `skas` patch. The first mode is called `tt` mode, or "tracing thread" mode, after the single master thread that controls the operation of the rest of the UML instance. The second is called `skas` mode, or "separate kernel address space" mode. This requires a patch applied to the host kernel. UML running in this mode is more secure and performs better than in `tt` mode.

Recently, a third mode has been added that provides the same security as skas, plus some of the performance benefits, on unmodified hosts. The current skas host patch is the third version, so it's called skas3. This new mode is called skas0 since it requires no host changes. The intent is for this to completely replace tt mode since it is generally superior, but tt mode still has some advantages. Once this is no longer the case, the support for tt mode will be removed. Even so, I will describe tt mode here since it is not clear when support for it will be removed, and you may need an older release of UML that doesn't have skas0 support.

As the term skas suggests, the main difference between tt mode and the two skas modes is how UML lays out address spaces on the host. Figure 9.2 shows a process address space in each mode. In tt mode, the entire UML kernel resides within each of its process's address spaces. In contrast, in skas3 mode, the UML kernel resides entirely in a different host address space. skas0 mode is in between, as it requires that a small amount of UML kernel code and data be in its process address spaces.

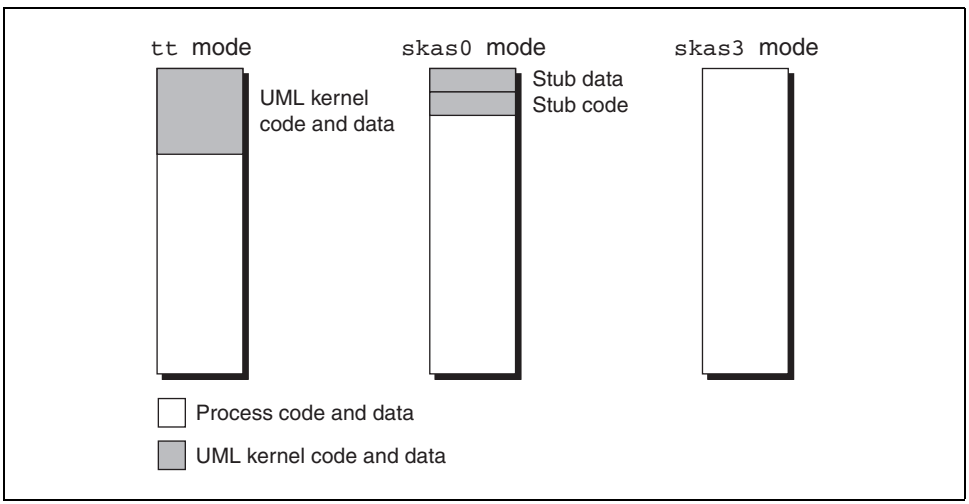


Figure 9.2 The three UML execution modes differ in how they lay out their process address spaces. tt mode maps the entire UML kernel into the upper .5GB of the process address space. skas0 mode leaves the UML kernel outside the process address space, except for two pages of memory mapped at the very top of the process address space. These are used to receive SIGSEGV signals and pass the resulting page fault information back to the UML kernel, and to modify the process address space. These two pages are unnecessary in skas3 mode, which allows its processes to use the entire address space.

The relationship between UML processes and the corresponding host processes for each mode follows from this. Figure 9.3 shows these relationships.

`tt` mode really only exists on x86 hosts. The x86_64 and S/390 ports were made after `skas0` mode was implemented, and they both use that rather than `tt` mode. Because of this, in the following discussion about `tt` mode, I will talk exclusively about x86. Also, the discussion about address space sizes and constraints on UML physical memory sizes are confined to x86, since this issue affects only 32-bit hosts.

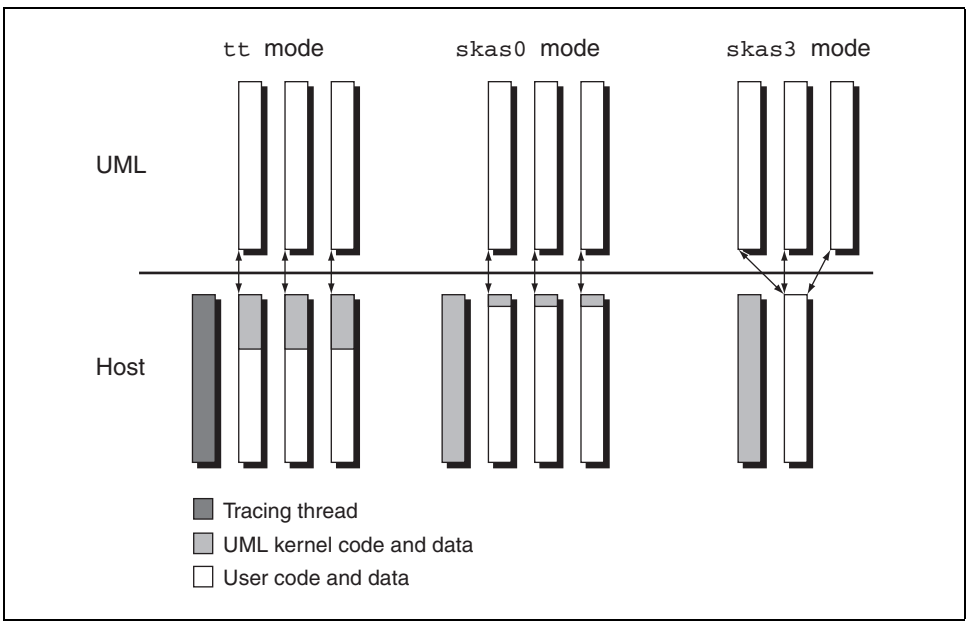


Figure 9.3 Comparison of the three UML execution modes. `tt` mode has a separate host thread (the tracing thread), which controls the execution of the other threads. Processes and threads within the UML instance have corresponding threads on the host. Each such host process has the UML kernel mapped into the top of its address space. In `skas3` mode, there is no separate tracing thread—this role is performed by the kernel thread. There is a single process on the host in which all UML processes run. `skas0` mode is a hybrid of `tt` mode and `skas3` mode. Like `skas3` mode, there is no tracing thread and there is a separate kernel thread in which the UML kernel lives. Like `tt` mode, each UML process has a corresponding host process.

`tt` Mode

In `tt` mode, a single tracing thread controls the rest of the threads in the UML instance by deciding when to intercept their system calls and have them executed within UML. When a UML process is running, it intercepts its system calls, and when the UML kernel is running, it doesn't. This is the tracing that gives this thread its name.

The tracing thread has one host process per UML process under its control. This is necessary because UML needs a separate host address space for each UML process address space, and creating a host process is the only way to get a new host address space. This is wasteful since all of the other host kernel data associated with the process, such as the kernel stack and task structure, are unnecessary from the point of view of UML. On a uniprocessor UML instance, there can be only one of these host processes running at any given time, so all of the idle execution contexts represented by the other host processes are wasted. This problem is fixed in `skas3` mode, as described in the next section.

The UML kernel is placed in the upper .5GB of each process address space. This is the source of the insecurity of `tt` mode—the UML kernel, including its data, is present and writable in the address spaces of its processes. Thus, a process that knew enough about the internals of UML could change the appropriate data inside UML and escape onto the host by tricking the tracing thread into not intercepting its system calls.

It is possible to protect the kernel's memory from its processes by write-protecting it when exiting the kernel and write-enabling it when entering the kernel. This has been implemented but never used because it imposes a huge performance cost. This protection has other problems as well, including complicating the code and making Symmetric Multi-Processing (SMP) impossible. So, it has probably never been used except in testing.

The fact that UML occupies a portion of the process address space is also a problem. The loss to UML processes of the upper .5GB of address space is inconvenient to some processes, and confining UML to that small address space limits the size of its physical memory. Since normal physical memory must be mapped into the kernel address space, the maximum physical memory size of a UML is less than .5GB. In practice, the limit is around 480MB.

You can use Highmem support to get around this. Highmem support in Linux exists because of the need to support more than 4GB of

physical memory in 32-bit x86 machines, which can access only 4GB of memory in a single 32-bit address space. In practice, since the x86 kernel has 1GB of address space (by default, it occupies the upper 1GB of its process's address spaces), it needs Highmem support to access more than 1GB of physical memory.

The memory pages above the lower 1GB can be easily used for process memory, but if the kernel is to use a Highmem page for its own memory, it must temporarily map it into its address space, manipulate the data in it, and then unmap it. This imposes a noticeable performance cost.

UML has a similar problem with Highmem memory, and, in `tt` mode, it starts at around .5GB of physical memory, rather than 1GB. To access memory outside this region, it must also map it into its address space, but this mapping is more expensive for UML than it is for the host. So, UML suffers a greater performance penalty with a large physical memory than the host does.

`skas3` Mode

The problems with `tt` mode motivated the development of the `skas3` host patch. These problems were driven by host limitations (or so we thought until someone figured out a way around them), so the `skas3` patch added mechanisms to the host that allowed UML to avoid them.

`skas3` gets its name from using the third version of the "separate kernel address space" host patch. As its rather unimaginative name suggests, the `skas3` patch allows the UML kernel to be in a separate host address space from its processes. This protects it from nosy processes because those processes can't form a UML kernel address to write. The UML kernel is completely inaccessible to its processes.

`skas3` also improved UML performance. Removing the UML kernel from its processes made new process creation faster, shrunk some pieces of data in the host kernel, and may speed context switching. In combination, these effects produced a very noticeable performance improvement over `tt` mode.

To allow the UML kernel to exist in a separate address space from its processes, a small number of new facilities were needed in the host:

- Creation, manipulation, and destruction of host address spaces that are not associated with a process

- ☞ Extraction of page fault information, such as the faulting address, access type, and processor flags, after a process receives a `SIGSEGV`
- ☞ Manipulation of the Local Descriptor Table (LDT) entries of another process

The address space manipulation is enabled through a new file in `/proc` called `/proc/mm`. Opening it creates a new, empty host address space and returns a file descriptor that refers to that address space. When the file descriptor is closed, and there are no users of the address space, the address space is freed.

A number of operations were formerly impossible to perform on an outside address space. Changing mappings is the most obvious. To handle a page fault in `tt` mode, it is sufficient to call `mmap` since the kernel is inside the process address space. When the process is outside it, we need something else. We can have the address space file descriptor support these operations through writing specially formatted structures to it. Mapping, unmapping, and changing permissions on pages are done this way, as is changing LDT entries associated with the address space.

Now that we can create host address spaces without creating new host processes, the resource consumption associated with `tt` mode goes away. Instead of one host process per UML process, there is now one host process per virtual processor. The UML kernel is in one host process that does system call interception on another, which, on a uniprocessor UML, runs all UML processes. It does so by switching between address spaces as required, under the control of the UML kernel invoking another `ptrace` extension, `PTRACE_SWITCH_MM`. This extension makes the `ptrace`d process switch from one host address space to another.

With the UML kernel in its own address space, it is no longer constrained to the 1GB of address space of `tt` mode. This enables it to have a much larger physical memory without needing to resort to Highmem. In principal, the entire 3GB address space on x86 is available for use as UML physical memory. In practice, the limit is somewhat lower, but, at around 2.5GB, still much greater than the 480MB limit imposed by `tt` mode.

In order to achieve this higher limit, the UML kernel must be configured with `CONFIG_MODE_TT` disabled. With both `CONFIG_MODE_TT` and `CONFIG_MODE_SKAS` enabled, the resulting UML kernel must be able to run in both modes, depending on its command line and the host capabilities it detects when it boots. A dual-mode UML instance will be compiled to load into the upper .5GB of its address space, as required

for tt mode, and will be subject to the 480MB physical memory limit. Disabling CONFIG_MODE_TT causes the UML binary to be compiled so it loads lower in its address space, where more normal processes load. In this case, the physical memory limit increases to around 2.5GB.

This is fortunate since Highmem is slower in skas3 mode than in tt mode, unlike almost all other operations. This is because a skas3 mode UML instance needs to map Highmem pages into its address space much more frequently than a tt mode UML instance does. When a UML process makes a system call, it is often the case that one of the arguments is a pointer, and the data referenced by that pointer must be copied into the UML kernel address space. In tt mode, that data is normally available to simply copy since the UML kernel is in the UML process address space. In skas3 mode, that isn't the case. Now, the UML kernel must work out from the process pointer it was given where in its own physical memory that data lies. In the case of Highmem memory, that data is not in its physical memory, and the appropriate page must be mapped into its address space before it can access the data.

Finally, it is necessary to extract page fault information from another process. Page faults happen when a process tries to execute code or access data that either has not been read yet from disk or has been swapped out. Within UML, process page faults manifest themselves as SIGSEGV signals being delivered to the process. Again, in tt mode, this is easy because the UML kernel itself receives the SIGSEGV signal, and all the page fault information is on its stack when it enters the signal handler. In skas3 mode, this is not possible because the UML kernel never receives the SIGSEGV. Rather, the UML kernel receives a notification from the host that its process received a SIGSEGV, and it cancels the signal so that it is never actually delivered to the process. So, the skas3 patch adds a ptrace option, PTRACE_FAULTINFO, to read this information from another process.

Together, these host changes make up the skas3 patch. UML needed to be modified in order to use them, of course. Once this was done, and the security and performance benefits became apparent, skas3 became the standard for serious UML installations.

skas0 Mode

More recently, an Italian college student, Paolo Giarrusso, who had been doing good work on UML, thought that it might be possible to implement something like skas3 on hosts without the skas3 patch.

His basic idea was to insert just enough code into the address space of each UML process to perform the address space updates and information retrieval for which `skas3` requires a host patch. As I implemented it over the following weekend, this inserted code takes the form of two pages mapped by the UML kernel at the top of each process address space. One of these pages is for a `SIGSEGV` signal frame and is mapped with write permission, and the other contains UML code and is mapped read-only.

The code page contains a function that invokes `mmap`, `munmap`, and `mprotect` as requested by the UML kernel. The page also contains the `SIGSEGV` signal handler. The function is invoked whenever address space changes are needed in a UML process and is the equivalent of requesting an address space change through a `/proc/mm` file descriptor. The signal handler implements the equivalent of `PTRACE_FAULTINFO` by receiving the `SIGSEGV` signal, reading all of the fault information from its stack, and putting it in a convenient form where the UML kernel can read it.

Without changes in the host kernel, we have no way to create new host address spaces without creating new host processes. So, `skas0` mode resembles `tt` mode in having one host process for each UML process.

This is the only similarity between `skas0` mode and `tt` mode. In `skas0` mode, the UML kernel runs in a separate host process and has a separate host address space from its processes. All of the `skas3` benefits to security and performance flow from this property. The fact that the UML kernel is controlling many more processes than in `skas3` mode means that we have the same wasted kernel memory that `tt` mode has. This makes `skas0` mode somewhat less efficient than `skas3` mode but still a large improvement over `tt` mode.

To Patch or Not to Patch?

With respect to how you want to run UML, at this writing, the basic choice is between `skas0` mode and `skas3` mode. The decision is controlled by whether you are willing to patch the host kernel in order to get better performance than is possible by using `skas0` mode.

We have a number of performance-improving patches in the works, some or all of which may be merged into the mainline kernel by the time this book reaches your bookshelf. You will be able to tell what, if any, patches are missing from your host kernel by looking at the early boot messages. Here is an example:

```
Checking that ptrace can change system call numbers...OK
Checking syscall emulation patch for ptrace...missing
Checking PROT_EXEC mmap in /tmp...OK
Checking if syscall restart handling in host can be \
   skipped...OK
Checking for the skas3 patch in the host:
  - /proc/mm...not found
  - PTRACE_FAULTINFO...not found
  - PTRACE_LDT...not found
UML running in SKAS0 mode
Adding 16801792 bytes to physical memory to account for \
   exec-shield gap
```

The message about the `syscall` emulation patch is talking about a `ptrace` extension that cuts in half the number of `ptrace` calls needed to intercept and nullify a host system call. This is separate from the `skas3` patch and is used in all UML execution modes. At this writing, this patch is in the mainline kernel, so a UML instance running on a host with 2.6.14 or later will benefit from this.

A few lines later, you can see the instance checking for the individual pieces of the `skas3` patch—`/proc/mm`, `PTRACE_FAULTINFO`, and `PTRACE_LDT`. Two of these, the two `ptrace` extensions, are likely to be merged into the mainline kernel separately, so there will likely be a set of host kernels for which UML finds some of these features but not all. In this case, it will use whatever host capabilities are present and use fallback code for those that are missing. `/proc/mm` will never be in the mainline kernel, so we are thinking about alternatives that will be acceptable to Linus.

For a smallish UML installation, a stock unmodified host kernel will likely provide good UML performance. So, in this case, it is probably not necessary to patch and rebuild a new kernel for the host.

Note that `tt` mode was not recommended in any situation. However, sometimes you may need to run an old version of UML in which `skas0` is not available. In this case, it may be a good idea to patch the host with the `skas3` patch. If UML running under `tt` mode is too slow or too resource intensive, or you need the security that comes with `skas3` mode, then patching with the `skas3` patch is the best course.

Vanderpool and Pacifica

Yet another option, which at this writing is not yet available but will be relatively soon, is to take advantage of the hardware virtualization support that Intel and AMD are incorporating into their upcoming pro-

cessors. These extensions are called Vanderpool and Pacifica, respectively. UML is currently being modified in order to take advantage of this support.

Vanderpool and Pacifica are similar, and compatible, in roughly the same way that AMD's Opteron and Intel's EM64T architectures are similar. There are some differences in the instructions, but they are relatively minor, and software written for one will generally run unmodified on the other. UML is currently getting Vanderpool Technology support, with the work being done by a pair of Intel engineers in Russia, but the result will likely run, perhaps with some tweaks, on an AMD processor with Pacifica support.

This support will likely bring UML performance close to native performance. The hardware support is sufficient to eliminate some of the largest performance bottlenecks that UML faces on current hardware. The main bottleneck is the context switching that `ptrace` requires to intercept and nullify system calls on the host. The hardware virtualization support will enable this to be eliminated, allowing UML to receive process system calls directly, without having to go through the host kernel. A number of other things will be done more efficiently than is currently possible, such as modifying process address spaces and creating new tasks.

In order to use this hardware virtualization support, you will need a host new enough to have the support in its processor. You will also need a version of UML that has the required support. Given these two requirements are met, UML will likely perform noticeably better than it does without that support.

MANAGING LONG-LIVED UML INSTANCES

It is common to want a UML instance to outlive the login session in which it is started. As with other processes, it is possible to background a UML instance and have it survive the end of the login session. The problem with this is the main console. It is natural to have it attached to the standard input and standard output of the UML instance's main process. But this means that the UML instance must be the foreground process. It can be backgrounded (with some difficulty because it sets the terminal raw, so Ctrl-Z and Ctrl-C don't send `SIGTSTP` and `SIGINT`, respectively, to the process), and once it is, and you log out, the main console is lost.

To avoid this, you can use a very handy tool called `screen`. Upon running it with no arguments, you get a new shell. At this point, you can run your UML instance as you normally do. When you decide to log out, you can detach the `screen` session, and it will, in effect, background the UML instance in a way that lets you reattach to it later.

Run `screen -r` and the session, with the UML instance and main console intact, will return. So, in the simplest case, here is the procedure for maintaining a long-lived UML instance.

1. Run `screen`.
2. Start the UML instance inside the resulting `screen` session.
3. Detach the screen session with Ctrl-A Ctrl-D.
4. Log out.
5. Later, log back in and run `screen -r`.
6. Detach, reattach, and repeat as often as necessary.

With a number of UML instances running on the host, the same procedure will work. The problem is knowing which screen session belongs to the UML instance you want to reattach to. The result of running `screen -r` may be something like this:

```
There are several suitable screens on:
       28348.pts-1.tp-w        (Detached)
       28368.pts-1.tp-w        (Detached)
       28448.pts-1.tp-w        (Detached)
       28408.pts-1.tp-w        (Detached)
       28308.pts-1.tp-w        (Detached)
       28488.pts-1.tp-w        (Detached)
       28530.pts-1.tp-w        (Detached)
       28328.pts-1.tp-w        (Detached)
       28428.pts-1.tp-w        (Detached)
       28550.pts-1.tp-w        (Detached)
       28468.pts-1.tp-w        (Detached)
       28288.pts-1.tp-w        (Detached)
       28510.pts-1.tp-w        (Detached)
       28388.pts-1.tp-w        (Detached)
Type "screen [-d] -r [pid.]tty.host" to resume one of them.
```

This is not helpful in figuring out which one you want to resume. To simplify this, `screen` has the ability to attach names to screen sessions. The `-S` switch will assign a meaningful name to the session and this name is what you will use to resume it. So,

```
host% screen -S joes-uml
```

will start a screen session named `joes-uml`. You can assign a name to each session you start. Then when you want to resume a particular one, run `screen -r` and you'll see something like this:

```
There are several suitable screens on:
       28868.work-uml          (Detached)
       28826.spare3-uml        (Detached)
       28910.simulator-uml     (Detached)
       28890.devel-uml         (Detached)
       28804.spare2-uml        (Detached)
       28784.spare1-uml        (Detached)
       28848.dmz-uml           (Detached)
       28764.janes-uml         (Detached)
       28742.named-uml         (Detached)
       28700.joes-uml          (Detached)
Type "screen [-d] -r [pid.]tty.host" to resume one of them.
```

It is now easy to pick out the one you want:

```
host% screen -r joes-uml
```

With good enough names, it may not even be necessary to look at the list in order to remember which one you want.

Finally, you may wish to start a set of UML instances during the host boot sequence. There is no terminal for the new UML instances to use as their main consoles, unless, of course, you provide them one. `screen` is useful here as well. The `-d -m` switch will start the `screen` session detached. Now you're not available to start the UML instances by hand, so `screen` will need to do this automatically. This can be accomplished, along with the other tricks we've seen, with something like this:

```
host% screen -d -m -S boot-uml ./linux con0=fd:0,fd:1 \
   con1=none con=pts ssl=pts umid=debian mem=450M \
    ubda=../../debian30 devfs=nomount mconsole=notify:/tmp/notify
```

This starts the screen session detached, runs the UML command that follows the screen switches, and names the screen session `boot-uml`. `screen -r` shows it like this:

```
       16799.boot-uml          (Detached)
```

Now, once the host has booted, and the UML instances with it, you can log in to the host and attach to whatever UML instance you wish.

NETWORKING

I've covered networking in sufficient detail earlier in the book that I don't need to belabor it here. However, I will repeat a few important points.

- ☞ Given that you control the host, the only two networking mechanisms you should consider for allowing access to the host network are TUN/TAP and `uml_switch`.
- ☞ Both bridging TUN/TAP devices with the host Ethernet and routing to unbridged TUN/TAP devices are appropriate models. They have differing setup and security requirements, which should drive the decision between the two.
- ☞ Make use of the ability to give descriptive names to your TUN/TAP devices to document your UML configuration.

UML PHYSICAL MEMORY

UML uses a temporary file as its physical memory. It does this rather than use anonymous memory since it needs file-backed memory so pages of memory can be mapped in multiple locations in multiple address spaces. This is impossible with anonymous memory. By default, UML creates the file in `/tmp` and removes it so it can't be accessed by any other process. If you look at the open file descriptors of a UML instance, you will see something like this:

```
lrwx------  1 jdike jdike 64 Aug 14 13:15 3 -> \
    /tmp/vm_file-lQkcul (deleted)
```

Because the file has been deleted and UML is holding a reference to it by keeping it open, the file is occupying space in `/tmp` but isn't visible to `ls`. If you ran `df`, you would see that some space has disappeared, but there are no visible files consuming that space.

Thus, the first requirement on the host with respect to UML physical memory is that the filesystem on which it will create its physical memory files must be large enough to hold all of those files. For example, if you decide not to change the default and to use `/tmp`, the filesystem on which `/tmp` lives must have enough free space to hold all of the physical memory files for all of the UML instances on the host. These files will be the same size as the physical memory assigned to the UML

instances. So, to decide how big your /tmp needs to be, you must add the physical memory sizes of all UML instances that will put their physical memory files in /tmp.

The UML instances will not occupy all of this space immediately. Rather, it will be consumed as they allocate and use pages of their own physical memory. Thus, the space on the host used by a UML instance will grow asymptotically to its physical memory size.

For performance reasons, it is a very good idea to create the UML physical memory files on a tmpfs filesystem. UML instances that have their memory files on a disk-based filesystem are noticeably slower. The filesystem should be sized as described above. In /etc/fstab, the entry for a 512MB tmpfs mount on /tmp would look like this:

```
none                    /tmp                    tmpfs    \
   size=512M       0 0
```

The equivalent command for doing this mount is:

```
host# mount none /tmp -t tmpfs -o size=512M
```

This is sufficient for one or two reasonably-sized UML instances. For a larger number, a much larger size, obtained by adding the physical memory sizes, would be needed.

You may wish to give each UML instance a separate tmpfs mount or to group them into several mounts, providing a degree of isolation between the UMLs or the groups. This could be useful if one instance somehow outgrew its physical memory and started occupying more space than it should.

This shouldn't happen, and I know of no cases where it has, but it is a conceivable failure that would affect the stability of the other UML instances sharing a tmpfs filesystem. If the filesystem gets filled, the host will stop allocating memory for new pages of memory within it. Since this is caused by the UML instances changing hitherto unmodified memory, if the tmpfs filesystem is full, those memory references will start failing. The UML instances will start receiving SIGBUS signals and very likely crash when some of those references occur inside the UML kernel.

Creating multiple tmpfs filesystems, up to one per UML instance, reduces the vulnerability of a UML instance to another overallocating space. With one UML instance per filesystem, if a UML instance somehow exceeded its physical memory size, that instance would be the only one affected.

Finally, a point I mentioned earlier bears repeating here. Giving a UML instance so much physical memory that it needs to use some of it as Highmem will hurt its performance. If you need to have physical memory sizes greater than the 480MB limit of `tt` mode, you should disable `CONFIG_MODE_TT`.

HOST MEMORY CONSUMPTION

Host memory is often the bottleneck constraining the number of UML instances that can be run while maintaining good performance. You can do two principle things to reduce the amount of host memory consumed by the UML instances. Both ideas involve cutting down on the caching of multiple copies of the same data.

- ☞ Run 2.6 on the host. As described earlier, this will cause the UML instances to use the `O_DIRECT` capability introduced in 2.6. Data read by the UML instances will be read directly into their page caches and won't occupy memory in the host page cache.
- ☞ Use COW files wherever possible. This will cause data from the backing files to be shared between the UML instances using them. Instead of having one copy in the host page cache for each UML instance, there will be only one total. There will still be one copy in every UML instance sharing that page.

  An enhancement that is not fully implemented at this writing is to have the `humfs` filesystem map, rather than copy, pages from its files into its page cache. This would reduce the number of copies of shared file pages from one per UML instance to one total since all the UML instances would be sharing the host's copy. This would require the UML host administrator to create `humfs` root filesystems and boot the UML instances on them.

  The UML block driver can't use `mmap` because the filesystems using it would lose control over when their file data and metadata are written to disk. This control is essential in their guarantees of file consistency and recovery in the case of a crash. Control would be lost because modifications to mapped data can be written to disk at any time by the host. Preventing this problem was one of the motivations for writing `humfs`. With the filesystem doing the mapping itself, rather than the underlying block device, it retains that control.

umid DIRECTORIES

By default, the unique machine id, or umid, directory for a UML instance is .uml/<umid> in the home directory of the user running the instance. This directory contains the mconsole socket and the pid file for the instance. If you decide to provide each instance with its own tmpfs mount, as described earlier, this would be a natural place to create it.

For management purposes, you may want to move the umid directories to a different location. For example, you might want to have each UML instance owned by a different user on the host but to have all of their umid directories in a single location. To do this, there is a UML switch that specifies the umid directory:

```
uml_dir=<umid path>
```

For instance, putting these switches on the command line for a UML would create its umid directory at /var/run/uml/debian:

```
umid=debian uml_dir=/var/run/uml
```

OVERALL RECOMMENDATIONS

This chapter boils down to a small number of recommendations for managing a modest UML server.

- ☞ Use a recent 2.6 kernel on the host. This will have performance enhancements for UML on all architectures and necessary bug fixes on x86_64 and S/390. It will give you the AIO and O_DIRECT capabilities, which UML will take advantage of.
- ☞ Make sure CONFIG_MODE_TT is disabled. It is disabled in the default configuration of UML, so you likely won't have to do anything except verify this. Having CONFIG_MODE_TT disabled will give you more flexibility in the amount of physical memory you can provide to your UML instances.
- ☞ Consider applying the skas3 patch to the host. This will provide somewhat better performance than skas0.
- ☞ Mount a tmpfs filesystem on /tmp, or wherever you have the UML instances create their physical memory files, and make sure it is large enough to hold all of those files.

- ☞ `screen` is an essential tool for managing long-lived UML instances. Become familiar with it.
- ☞ Be careful about managing the host's physical memory. If the sum of the UML instances' physical memory sizes greatly exceeds the host's physical memory, performance will suffer as the host swaps out the UML instances. Look into techniques for reducing memory consumption such as COWing your `ubd` filesystem images or booting from `humfs` directories.
- ☞ It may simplify the management of your instances to centralize their `umid` directories.

CHAPTER 10

Large UML Server Management

In the previous chapter, we talked about setting up a smallish UML server where the UML users would be local users who have accounts on the host and where it is not a goal to run as many UML instances on the host as possible. Now, we will take a look at running a large server where the UML users are untrusted, we want the largest possible number of instances running with acceptable performance, and we are willing to tune the host in order to accomplish this.

Security is going to be a major theme. The presence of untrusted users who do not have accounts on the host and who should stay confined to their UML instances requires a layered approach to security. The first layer is UML itself. There are no known exploits to break out of a UML jail, but it is prudent to take that possibility into account and ensure that if such an exploit did exist, the host would not be harmed.

We are also going to be careful about network security, taking steps to minimize the possibility of someone using a UML instance to launch attacks or otherwise engage in hostile network behavior.

Security is also an issue when providing users access to their console devices. These are normally attached to devices on the host, making it necessary to have access to the host in order to get console access.

Instead, we will look at a way to provide this access by using a dedicated UML instance for it, avoiding the need to provide direct access to the host.

Finally, I will describe some enhancements on both the host and UML that will improve performance, resource consumption, and manageability of UML instances in the future.

SECURITY

UML Configuration

When you are concerned about preventing people from breaking out of a UML instance, the first thing to look at is the configuration of UML itself. Like the host, UML has two protection levels, user mode and kernel mode. In user mode, system calls are intercepted by the UML kernel, nullified, and executed in the context of UML. This is the basis for UML jailing. The system calls and their arguments are interpreted within the context of UML.

For example, when a process executes a read from its file descriptor zero, the file that is written is taken from the first entry of the process's file table within the UML kernel rather than the first entry of the file table in the host kernel. That would be the standard input of UML itself rather than that of the UML process. Similarly, when a process opens a file, it is the filesystem code of the UML, rather than the host, that does the filename lookup. This ensures that a UML process has no access to files on the host. The same is true for all other resources, for the same reason.

When the UML kernel itself is running, system call tracing is disabled, and the kernel does have access to host resources. This is the critical difference between user mode and kernel mode in UML. Since the UML kernel can execute system calls on the host, all code in the UML kernel must remain trusted. If a user were able to insert arbitrary code into the kernel, that user could break out. It would simply be a matter of executing a shell on the host from within the UML kernel.

There is a well-known mechanism in Linux for doing exactly this: kernel modules. Of course, they are intended for something entirely different—dynamically extending the kernel's functionality by adding drivers, filesystems, network protocols, and the like. But extending the kernel's functionality, in the context of UML, can also be interpreted as allowing the UML user to execute arbitrary commands on the host.

Since we can't prevent this and also allow legitimate kernel modules to be installed, in a secure UML configuration, modules need to be disabled completely.

It turns out that modules aren't the only mechanism by which a user could inject code into the UML kernel. An entry in `/dev`, `/dev/mem`, provides access to the system's physical memory. Since the kernel and its data are in that memory, with the ability to write to this file, a nasty UML user could manually inject the equivalent of a module into the kernel and change data structures in order to activate it so that the kernel will execute the code.

This may sound hard to actually carry out successfully, but it is a skill that rootkit writers have perfected. In certain circles, it is very well known how to inject code into a Linux kernel and activate it, even in the absence of module support, and there are reliable tools for doing this automatically.

The obvious way to prevent someone from writing to a file is to set the permissions on the file in order to make that impossible. However, since the UML user could very likely be root, file permissions are useless. The root user is not in any way restricted by them.

Another mechanism is effective against the root user: capabilities. These are a set of permissions associated with a process rather than a file. They have two important properties.

1. They are inherited by a process from its parent.
2. They can be dropped, and once dropped, can't be regained by the process or its children.

Together, these properties imply that if the kernel or `init`, which is ultimately the parent of every other process on the system, drop a capability, then that capability is gone forever for that system. No process can ever regain it.

It turns out that there is a capability that controls access to `/dev/mem`, and that is `CAP_SYS_RAW`. If this is dropped by the kernel before running `init`, no process on the system, including any process run by the root user, will be able to modify the UML instance's physical memory through `/dev/mem`. Removing `CAP_SYS_RAW` from the initial set of capabilities (the bounding set) will irreversibly remove it from the entire system, and nothing will be able to write to kernel memory.

A second issue is access to host filesystems. If the UML kernel has `CONFIG_EXTERNFS` or `CONFIG_HOSTFS` enabled, a UML user will be able to mount directories on the host as filesystems within the UML

instance. For a secure UML environment, this is usually undesirable. The easiest way to prevent this is to disable CONFIG_EXTERNFS and CONFIG_HOSTFS in the UML kernel.

If you do want to allow some access to host files, it can be done securely, but it requires some care because it opens up some more avenues of attacks. There are no known holes here, but allowing any extra access to the host from the UML instance will provide more possibilities for malicious users or software to attack the host.

First of all, it's a good idea to run the UML instance inside a jail (we talk about setting up a good jail later in this chapter) and, inside that, provide the directory that you wish to allow the instance to access. Second, you can use a UML switch to force all hostfs mounts to be within a specified host directory. For example, the following option will restrict all hostfs mounts to be within the directory /uml-jails/jeffs-uml:

```
hostfs=/uml-jails/jeffs-uml
```

This is done by prepending that directory name to every host directory the UML attempts to mount. So, if the UML user tries to mount the host's /home like this:

```
UML# mount none /mnt -t hostfs -o /home
```

the UML instance will really attempt to mount /uml-jails/jeffs-uml/home. If there really is a directory named /uml-jails/jeffs-uml/home, that mount will succeed, and if not, it will fail. But in no case will the UML instance attempt to mount the host's /home.

If you wish to provide each UML instance with some host directory that will be private to the instance, simply copying that directory into the instance's jail is the easiest way to make it available.

If you wish to provide the same host directory to a number of UML instances, you can make it available within each jail directory with a bind mount. Bind mounts are new with 2.6, so you'll need a 2.6 host in order to use them. This facility allows you to make an existing directory available from multiple places within the filesystem. For example, here is how to make /tmp available as ~/tmp:

```
host% mkdir ~/tmp
host# mount --bind /tmp ~/tmp
host% ls /tmp
gconfd-jdike    orbit-jdike     ssh-Xqcrac2878
keyring-4gMKe0  ssh-QWYGts4184  ssh-vlklKu4277
```

```
mapping-jdike   ssh-VMnkLn4309  xses-jdike.oBNeep
host% ls ~/tmp
gconfd-jdike    orbit-jdike     ssh-Xqcrac2878
keyring-4gMKe0  ssh-QWYGts4184  ssh-vlklKu4277
mapping-jdike   ssh-VMnkLn4309  xses-jdike.oBNeep
```

Now the same directory is available through the paths `/tmp` and `~/tmp`. It's exactly the same directory—creating a new file through one path will result in it being visible through the other.

To use this technique to provide a common `hostfs` directory to a set of UML instances, you would do something like this for each instance:

```
host# mount --bind /umls/common-data /umls/uml1-jail/data
```

Following this with the `hostfs` jailing switch would add another layer of protection:

```
hostfs=/umls/uml1-jail/data
```

As I mentioned before, this does add another possible avenue of attacks on the host from the UML instances. However, the risk of a UML instance gaining access to information outside the directories explicitly provided to it is minimal when the instances are jailed and the `hostfs` mounts themselves are jailed.

Generally, such data would be read-only and would be provided to the UML instances as a reference, such as a package repository. This being the case, all files and subdirectories should be write-protected against the UML instances. You can accomplish this by having these files and subdirectories owned by a user that does not own any of the UML instances and having everything be read-only for the owner, group, and world.

In the spirit of having multiple layers of protection, an additional `hostfs` option, `append`, restricts the file modifications that can be performed through a `hostfs` mount.

```
hostfs=/uml-jail,append
```

When you add `append` to the `hostfs` switch as shown, the following restrictions come into force.

- ☞ All file opens are done with `O_APPEND`. This means that all file writes will append to the file rather than overwriting data that's already there.

- Files can't be shrunk, as with `truncate`.
- Files can't be removed.

The purpose of the `append` switch is to prevent data from being destroyed through the `hostfs` mount. It does not prevent writing of new data, so if you want that restriction, you must still write-protect the `hostfs` directories and everything in them.

If you do wish to provide the UML instances with the ability to write to their `hostfs` mounts, you are providing a new avenue of attack to a malicious UML user. This potentially enables a denial-of-service attack on the host's disk space rather than its data. By filling the `hostfs` directories with data and filling up the filesystem on which it lives, an instance could make that host disk space unusable by the other UML instances. This possible problem can be handled with disk quotas on the host if each UML instance is owned by a different host user.

Even so, `humfs` is probably a better option in this case. When writing files on the host, the permission and ownership problems I mentioned earlier rear their heads. `hostfs` files will be owned by the host user that is running the UML instance, rather than the UML user that created them, leading to a situation where a UML user can create a file but subsequently can't modify it. `humfs` handles this correctly, and it has a built-in size limit that can be used to control the consumption of host disk space.

JAILING UML INSTANCES

The centerpiece of any layered security design for a large UML server is the jail that the UML instances are confined to. Even though UML confines its users, it is prudent to assume that, at some point, someone will find a hole through which they can escape onto the host. No such holes are known, but it's impossible to prove that they don't exist and, if one did exist, that it couldn't be exploited.

This jail will make use of the Linux `chroot` system call, which confines a process to a specific directory. You can see the effect of using the `chroot` command, which is a wrapper around the system call, to confine a shell to `/sbin`.

```
host# chroot /sbin ./sash
Stand-alone shell (version 3.7)
> -pwd
/
```

```
> -ls
.
..
MAKEDEV
```

Notice how the current directory is `/`, but its contents are those of `/sbin`. `chroot` arranges that the directory specified as the jail becomes the new process's root directory. The process can do anything it normally has permissions to do within that directory tree but can't access anything outside it. This fact forced the choice of `sash` as the shell to run within the `chroot`. Most other shells are dynamically loaded, so they need libraries from `/lib` and `/usr/lib` in order to run. When jailed, they can no longer access those libraries—even if the libraries are within the jail, they will be in the wrong location for the dynamic loader to find, even when the dynamic loader itself can be found.

So, for demo purposes, the easiest way to show how `chroot` works is by running a statically linked shell within its own directory. More serious purposes require constructing a complete but minimal environment, which we will do now. This environment must contain everything that the jailed process will need, but nothing else.

We will construct a jail that is surprisingly empty. This provides as few tools as possible to an attacker who somehow manages to break out of a UML instance. He or she will want to continue the attack in order to subvert the host. In order to do this, the attacker will need to break out of the `chroot` environment. If there is a vulnerability (and I am aware of no current holes in `chroot`), the attacker will need tools in order to exploit it. Making the `chroot` environment as empty as it can be will go some way toward denying him or her these tools.

First we must decide what a UML instance needs in order to simply boot. Looking at the shared libraries that UML links against and a typical UML command line gives us a start:

```
host% ldd linux
        linux-gate.so.1 =>  (0x003ca000)
        libutil.so.1 => /lib/libutil.so.1 (0x00c87000)
        libc.so.6 => /lib/libc.so.6 (0x0020a000)
        /lib/ld-linux.so.2 (0x001ec000)
host% ./linux con0=fd:0,fd:1 con1=none con=pts ssl=pts \
     umid=debian mem=450M ubda=../../debian30 devfs=nomount
```

With this UML binary, those libraries would need to be present within `./lib` and within the jail in order to even launch it. After launching, the command line makes a number of other requirements in order for UML to boot:

- ☞ The root filesystem, `debian30`, needs to be present in the jail, and not two directory levels higher, as I have it here.
- ☞ `con=pts` and `ssl=pts` require that `./dev/pts` exist within the jail.
- ☞ The UML instance will try to create the `umid` directory for the `pid` file and `mconsole` socket in the user's home directory within the jail.

This would be far from being an empty directory, and it would contain files such as libraries and device nodes that an attacker might find useful. Fortunately, these requirements can be reduced in some fairly simple ways.

First, to eliminate the requirement for libraries, we can make the UML executable statically, rather than dynamically, linked. If `CONFIG_MODE_TT` is enabled, UML is linked statically. However, for a serious server, it is highly recommended that the UML instances use either `skas0`, if the server is running an unmodified kernel, or `skas3`, if the `skas3` patch can be applied to the host kernel. With `CONFIG_MODE_TT` disabled, UML will link dynamically. However desirable this is in general, it complicates setting up a jail. So, a configuration option for UML, `CONFIG_STATIC_LINK`, forces the UML build to produce a statically linked executable, even when `CONFIG_MODE_TT` is disabled.

Enabling `CONFIG_STATIC_LINK` results in a larger UML binary, which is slightly less efficient for the host because the UML instances are no longer sharing library code that they would share if they were linked dynamically. This is unavoidable—even if you copied the necessary libraries into the jail, each UML instance would have its own copy of them, so there would still be no sharing. There is a clever way to have the libraries be present in each jail but still shared with each other—use the `mount --bind` capability described earlier to mount the necessary libraries into the jails.

However, this is too clever—it opens up a possible security hole. If an attacker were somehow able to break out of a UML instance, gain access to the jail contents, and modify the libraries, those libraries would be modified for the entire system. So, if the attacker could add code to `libc`, at some point that code would be executed by a root-owned process, and the host would be subverted. So, for security reasons, we need no shared code between the UML instance and anything else on the system. Once we have made that decision, there is no further cost to statically linking the UML binary.

The next issue is the /dev/pts requirements imposed by the console and serial line arguments. These are easy to dispose of by changing those configurations to ones that require no files in the jail. We have a variety of possibilities—null, port, fd, and none all fill the bill. null and none effectively make the consoles and serial lines unusable. port and fd make them usable from the host outside the jail. For an fd configuration, you would have to open the necessary file descriptors and pass them to the UML instance on its command line.

Finally, there is the umid directory. We can't eliminate it without losing the ability to control the instance, but we can specify that it be put someplace other than the user's home directory within the jail. By creating a ./tmp directory within the jail and using the uml_dir switch to UML, we can arrange for the pid file and mconsole socket to be put there.

At this point, the jail contents look like this:

```
host% ls -Rl
.:
total 1033664
-rw-rw-r--  1 jdike jdike 1074790400 Aug 18 17:46 debian30
-rwxrwxr-x  1 jdike jdike   20860108 Aug 18 17:39 linux
drwxrwxr-x  2 jdike jdike       4096 Aug 18 17:46 tmp

./tmp:
total 0
```

As a quick test of whether UML can boot in this environment and of its new command-line switches, we can do the following as root in the jail directory:

```
host# chroot . ./linux con0=fd:0,fd:1 con1=none con=port:9000 \
    ssl=port:9000 umid=debian mem=450M ubda=debian30 \
    devfs=nomount uml_dir=tmp
```

It does boot, printing out a couple messages we haven't seen before:

```
/proc/cpuinfo not available - skipping CPU capability checks
No pseudo-terminals available - skipping pty SIGIO check
```

Because /proc and /dev are not available inside the jail, UML couldn't perform some of its normal checks of the host's capabilities. These are harmless, as the /proc/cpuinfo checks come into play only on old processors, and the pseudo-terminal test is necessary only when attaching consoles to host pseudo-terminals, which we are not doing.

Running UML in this way is useful as a test, but we ran UML as root, which is very much not recommended. Running UML as a normal, nonprivileged user is one of the layers of protection the host has, and running UML as root throws that away. Root privileges are needed in order to enter the `chroot` environment, so we need a way to drop them before running UML.

It is tempting to try something like this:

```
host# chroot jail su 1000 ./linux ...
```

However, this won't work because the `su` binary must be present inside the jail, which is undesirable. So, we need something like the following small C program, which does the `chroot`, changes its `uid` to one we provide on the command line, and executes the remainder of its command line:

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    int uid;
    char *dir, **command, *end;
        if(argc < 3){
        fprintf(stderr, "Usage - do-chroot dir uid \
            command-line...\n");
        exit(1);
    }

    dir = argv[1];
    uid = strtoul(argv[2], &end, 10);
    if(*end != '\0'){
        fprintf(stderr, "the uid \"%s\" isn't a number\n", \
            argv[2]);
        exit(1);
    }
    command = &argv[3];

    if(chdir(dir) < 0){
        perror("chroot");
        exit(1);
    }

    if(chroot(".") < 0){
        perror("chroot");
        exit(1);
    }
```

```
    if(setuid(uid) < 0){
        perror("setuid");
        exit(1);
    }

    execv(command[0], command);
    perror("execv");
    exit(1);
}
```

This is run as follows:

```
host# do-chroot jail 1000 ./linux con0=fd:0,fd:1 con1=none \
   con=port:9000 ssl=port:9000 umid=debian mem=450M \
   ubda=debian30 devfs=nomount uml_dir=tmp
```

Since I am specifying a nonexistent `uid`, everything in the jail should be owned by that user in order to prevent permission problems:

```
host# chown -R 1000.1000 jail
```

Now UML runs as we would like. It is owned by a nonexistent user, so it has even fewer permissions on the host than something run by a normal user.

We saw the contents of the jail directory as we have it set up. With the UML instance running, there are a couple more things in it:

```
host% ls -Rl
.:
total 1033664
-rw-rw-r--  1 1000 1000 1074790400 Aug 18 19:12 debian30
-rwxrwxr-x  1 1000 1000   20860108 Aug 18 17:39 linux
drwxrwxr-x  3 1000 1000       4096 Aug 18 19:12 tmp

./tmp:
total 4
drwxr-xr-x  2 1000 root 4096 Aug 18 19:12 debian

./tmp/debian:
total 4
srwxr-xr-x  1 1000 root 0 Aug 18 19:12 mconsole
-rw-r--r--  1 1000 root 5 Aug 18 19:12 pid
```

We also have the `mconsole` socket and the `pid` file in the `tmp` directory.

This is reasonably minimal, but we can do better. Some files are opened and never closed. In these cases, we can remove the files after we know that the UML instance has opened them. The instance will be able to access the file through the open file descriptor and won't need the file to actually exist within its jail.

Chief among these are the UML binary and the filesystem. We can remove them after we are sure that UML is running and has opened its filesystem. It is tempting to remove them immediately after executing UML, but that is somewhat prone to failure because the removals might run before the UML instance has run or before it has opened its filesystem.

To avoid this, we can use the MConsole `notify` mechanism we saw in Chapter 8. We'll use a slightly modified version of the Perl script used in that chapter to read notifications from a UML instance:

```
use UML::MConsole;
use Socket;
use strict;

@ARGV < 2 and die "Usage : running.pl notify-socket uid";

my $sock = $ARGV[0];
my $uid = $ARGV[1];

!defined(socket(SOCK, AF_UNIX, SOCK_DGRAM, 0)) and
    die "socket failed : $!\n";

!defined(bind(\*SOCK, sockaddr_un($sock))) and
    die "UML::new - bind failed : $!\n";

chown $uid, $uid, $sock || die "chown failed - $!";

my ($type, $data) = UML::MConsole->read_notify(\*SOCK, undef);
$type ne "socket" and
    die "Expected socket notification, got \"$sock\" " .
        "notification with data \"$data\"";
exit 0;
```

Running this as root like this:

```
host# perl running.pl tmp/notify 1000
```

and adding the following:

```
mconsole=notify:tmp/notify
```

to the UML command line will cause the `running.pl` script to exit when the instance announces that it has booted sufficiently to respond to MConsole requests.

At this point, the UML instance is clearly running and has the root filesystem open, so the UML binary and the filesystem can be safely removed. Under `tmp`, there is the MConsole socket and `pid` file.

The `pid` file is for management convenience, so it can be read and removed. The MConsole socket can be moved outside the jail, where it's inaccessible to anyone who somehow manages to break out of the UML instance, but where an MConsole client can access it.

The only thing that can't be removed is the `notify` socket, which has to stay where it is so that the UML instance can send notifications to it. If that socket is removed, you lose an element of control since you can't find out if the instance has crashed. If this is OK, you can remove the socket, and the UML instance will run in a completely empty jail.

One thing we haven't done here is to provide the UML instance with a swap device. Like the root filesystem, the swap device file needs to be in the jail. If it's removed, it can possibly be lost by the UML instance. If `swapoff` is run inside the instance, the block driver will close the swap device file. When this happens, the instance will lose the only handle it had to the file. If `swapon` is subsequently run, the block driver will attempt to open the file, and fail, since you removed it. This is not a problem for the root filesystem since, once mounted, it is never unmounted until the instance is shut down.

One side effect of removing the UML binary is that reboot will stop working. Rebooting is implemented by `exec`-ing the binary to get a clean start for the new instance. If the binary has been removed, `exec` will fail. However, this is probably not a big problem since a reboot is no different from a shutdown followed by a restart.

You need to be careful with the root filesystem. If you simply copy it into the jail, boot the UML instance, and remove the filesystem file, the instance will have access to the filesystem as long as it keeps the file open. When it shuts down, it will close the file, and it will be removed, along with whatever changes were made to it. To prevent this, you should keep the filesystem out of the jail and make a hard link to it from inside the jail. Now there will remain one reference to the file—the original name for it—and it will not be removed when the instance closes it.

PROVIDING CONSOLE ACCESS SECURELY

If you're running a large UML server where you need to be concerned about the behavior of outsiders, you're likely going to need a way to provide console access to the UML instances in a secure way. The obvious way to do this is to attach the UML consoles to some host device

and provide some sort of login on the host, where the login shell is a script that connects the user to the appropriate UML console. That's relatively simple, but it does have the disadvantage of providing users with unjailed access to the host. This sort of script often turns out to have security holes in it. Some kind of command interpreter inside might, through a programming mistake, allow a user to execute some arbitrary command on the host.

There is a way to provide console access that doesn't require any new tools to be written and doesn't give the UML user any unjailed access to the host.

The idea is to run a separate UML instance that serves as a console server for the other UML instances on the host. The other instances have their consoles attached to terminals within this console server. Each UML administrator has a normal, unprivileged account on this UML and has access to his or her consoles through these pseudo-terminals, which have been appropriately protected so as to allow access only to the administrator for the instance to which they connect.

I described this mechanism in Chapter 4, as a virtual serial line running between two UML instances. This is merely an application of it, with a bit of extra infrastructure. I will go through the process of setting this up for one UML by hand. If you run a large UML host, this procedure will need to be automated and included in your UML provisioning process.

First, we need to boot two UML instances and establish a virtual serial line connection between them. We start by finding a console in the user's instance that is attached to a host pseudo-terminal. Since I do so with all spare consoles and serial lines, this is easy:

```
host% uml_mconsole jeff config con2
OK pts:/dev/pts/11
```

I attach the slave end of this pseudo-terminal to an unused console in the console server instance:

```
host% uml_mconsole console-server config con6=tty:/dev/pts/11
OK
```

Now I need to create a normal user account for myself in the console server:

```
console-server# adduser jeff
console-server# passwd jeff
```

```
Changing password for user jeff.
New UNIX password:
Retype new UNIX password:
passwd: all authentication tokens updated successfully.
```

Since `tty6` in the console server is attached to my instance, I need to own the device file:

```
# chown jeff.jeff /dev/tty6
```

This allows me to access my instance, and it prevents other unprivileged users from accessing it.

Everything is now set up, and I should be able to log in over the network to the console server as `jeff` and from there attach to my UML instance over this virtual serial line:

```
[jdike@tp]$ ssh jeff@console-server
jeff@console-server's password:
[jeff@console-server ~]$ screen /dev/tty6
```

In the `screen` session, I now have the login prompt from my own UML instance and can log in to it:

```
Debian GNU/Linux testing/unstable jeff tty2
jeff login: root
Password:
Last login: Fri Jan 20 22:26:53 2006 on tty2
jeff:~#
```

This is fairly simple, but it's a powerful mechanism that allows your users to log in to their UML instances on a "hardwired" console without needing accounts on the host. If I kill the network on my instance, I can log in over a console and fix it. Without a mechanism like this, I would have to appeal to the host administrator to log in to my UML instance and fix it for me. Using a UML instance as the console server increases the security of this arrangement by making it unnecessary to provide accounts on the host for the UML users.

skas3 VERSUS skas0

The previous chapter contained a discussion of whether to leave the host unmodified and have the UML instances running in `skas0` mode or to patch the host with the `skas3` patch for better performance. Since we're now talking about a large UML server and we're trying to get

every bit of UML hosting capacity from it, I recommend patching the host with the `skas3` patch.

The reasons were mostly covered in the discussion in the last chapter. You'll get better performance with `skas3` than `skas0` for the following reasons.

- `skas3` creates one host process per UML processor while `skas0` creates one per UML process. This consumes host kernel memory unnecessarily and slows down process creation.
- `skas3` page faulting performance is better because it has a more efficient way to get page fault information from the host and to update the process address space in response to those page faults.

In addition to better performance, `skas3` will have somewhat lower host resource consumption due to the smaller number of processes created on the host.

FUTURE ENHANCEMENTS

A number of host kernel enhancements for improving UML performance host resource consumption are in the works. Some are working and ready to be merged into the mainline kernel, and some are experimental and won't be in the mainline kernel for a while.

sysemu

Starting with the mature ones, the `sysemu` patch adds a `ptrace` option that allows host system calls to be intercepted and nullified with one call to `ptrace`, rather than two. Without this patch, in order to intercept system calls, a process must intercept them both at system call entry and exit. A tool like `strace` needs to make the two calls to `ptrace` on each system call because the tool needs to print the system call when it starts, and it needs to print the return value when it exits. For something like UML, which nullifies the system call and doesn't need to see both the system call entry and exit, this is one `ptrace` call and two context switches too many.

This patch noticeably speeds up UML system calls, as well as workloads that aren't really system call intensive. A `getpid()` loop is faster by about 40%. A kernel build, which is a somewhat more representative workload than a `getpid()` loop, is faster by around 3%.

This improvement is not related to the skas3 patch at all. It is purely to speed up system call nullification, which UML has to do no matter what mode it is running in.

The sysemu patch went into the mainline kernel in 2.6.14, so if your host is running 2.6.14 or later, you already have this enhancement.

PTRACE_FAULTINFO

PTRACE_FAULTINFO is another patch that has been around for a long time. It is part of the skas3 patch but will likely be split out since it's less objectionable than other parts of skas3, such as /proc/mm. PTRACE_FAULTINFO is used by UML in either skas mode in order to extract page fault information from a UML process. skas0 mode has a less efficient way to do this but will detect the presence of PTRACE_FAULTINFO and use it if present on the host.

MADV_TRUNCATE

This is a relatively new patch from Badari Pulavarty of IBM. It allows a process to throw out modified data from a tmpfs file it has mapped. Rather than being a performance improvement like the previous patches, MADV_TRUNCATE reduces the consumption of host memory by its UML instances.

The problem this solves is that memory-mapped files, such as those used by UML for its physical memory, preserve their contents. This is normally a good thing. If you put some data in a file and it later just disappeared, you would be rather upset. However, UML sometimes doesn't care if its data disappears. When a page of memory is freed within the UML kernel, the contents of that page doesn't matter anymore. So, it would be perfectly alright if the host were to free that page and use it for something else. When that page of UML physical memory was later allocated and reused, the host would have to provide a page of its own memory, but it would have an extra page of free memory in the meantime.

I made an earlier attempt at creating a solution, which involved a device driver, /dev/anon, rather than an madvise extension. The driver allowed a process to map memory from it. This memory had the property that, when it was unmapped, it would be freed. /dev/anon mostly worked, but it was never entirely debugged.

Both `/dev/anon` and `MADV_TRUNCATE` are trying to do the same thing—poke a hole in a file. A third proposed interface, a system call for doing this, may still come into existence at some point.

The main benefit of these solutions is that it provides a mechanism for implementing hot-plug memory. The basic idea of hot-plug memory on virtual machines is that the guest contains a driver that communicates with the host. When the host is short of memory and wants to take some away from a guest, it tells the driver to remove some of the guest's memory. The guest does this simply by allocating memory and freeing it on the host. If the guest doesn't have enough free memory, it will start swapping out data until it does.

When the host wants to give memory back to a guest, it simply tells the driver to free some of its allocated memory back to the UML kernel.

This gives us what we need to avoid the pathological interaction between the host and guest virtual memory systems I described in Chapter 2. To recap, suppose that both the host and the guest are short of memory and are about to start swapping memory. They will both look for pages of memory that haven't been recently used to swap out. They will both likely find some of the same pages. If the host manages to write one of these out before the guest does, it will be on disk, and its page of memory will be freed. When the guest decides to write it out to its swap, the host will have to read it back in from swap, and the guest will immediately write it out to its own swap device.

So, that page of memory has made three trips between memory and disk when only one was necessary. This increased the I/O load on the host when it was likely already under I/O pressure. Reading the page back in for the benefit of the guest caused the host to allocate memory to hold it, again when it was already under memory pressure.

To make matters even worse, to the host, that page of memory is now recently accessed. It won't be a candidate for swapping from the host, even though the guest has no need for the data.

Hot-pluggable memory allows us to avoid this by ensuring that either the host or the UML instances swap, but not both. If the UML instances are capable of swapping—that is, the host administrator gave them swap devices—we should manage the host's memory to minimize its swapping. This can be done by using a daemon on the host that monitors the memory pressure in the UML instances and the host. When the host is under memory pressure and on the verge of swapping, the daemon can unplug some memory from an idle UML instance and release it to the host.

Hot-plug memory also allows the UML instances to make better use of the host's memory. By unplugging some memory from an idle UML instance and plugging the same amount into a busy one, it will effectively transfer the memory from one to the other. When some UML instances will typically be idle at any given time, this allows more of them to run on the host without consuming more host memory. When an idle UML instance wakes up and becomes busy again, it will receive some memory from an instance that is now idle.

Since the `MADV_TRUNCATE` patch is new, it is uncertain when it will be merged into the mainline kernel and what the interface to it will be when it is. Whatever the interface ends up being, UML will use it in its hot-plug memory code. If `MADV_TRUNCATE` is not available in a mainline kernel, it will be available as a separate patch.

The interface to plug and unplug UML physical memory likely will remain as it is, regardless of the host interface. This uses the MConsole protocol to treat physical memory as a device that can be reconfigured dynamically. Removing some memory is done like this:

```
host% uml_mconsole debian config mem=-64M
```

This removes 64MB of memory from the specified UML instance.

The relevant memory statistics inside the UML (freshly booted, with 192MB of memory) before the removal look like this:

```
UML# grep Mem /proc/meminfo
MemTotal:       191024 kB
MemFree:        117892 kB
```

Afterward, they look like this:

```
UML# grep Mem /proc/meminfo
MemTotal:       191024 kB
MemFree:         52172 kB
```

Just about what we would expect. The memory can be plugged back in the same way with:

```
host% uml_mconsole debian config mem=+64M
```

That brings us basically back to where we started:

```
UML# grep Mem /proc/meminfo
MemTotal:       191024 kB
MemFree:        117396 kB
```

The main limitation to this currently is that you can't plug arbitrary amounts of memory into a UML instance. It can't end up with more than it had when it was booted because a kernel data structure that is sized according to the physical memory size at boot can't be changed later. It is possible to work around this by assigning UML instances a very large physical memory at boot and immediately unplugging a lot of it.

This limitation may not exist for long. People who want Linux to run on very large systems are doing work that would make this data structure much more flexible, with the effect for UML that it could add memory beyond what it had been booted with.

Since this capability is brand new, the UML management implications of it aren't clear at this point. It is apparent that there will be a daemon on the host monitoring the memory usage of the host and the UML instances and shuffling memory around in order to optimize its use. What isn't clear is exactly what this daemon will measure and exactly how it will implement its decisions. It may occasionally plug and unplug large amounts of memory, or it may constantly make small adjustments.

Memory hot-plugging can also be used to implement policy. One UML instance may be considered more important than another (possibly because its owner paid the hosting company some extra money) and will have preferential access to the host's memory as a result. The daemon will be slower to pull memory from this instance and quicker to give it back.

All of this is in the future since this capability is so new. It will be used to implement both functionality and policy. I can't give recommendations as to how to use this capability because no one has any experience with it yet.

remap_file_pages

Ingo Molnar spent some time looking at UML performance and at ways to increase it. One of his observations was that the large number of virtual memory areas in the host kernel hurt UML performance. If you look in `/proc/<pid>/maps` for the host process corresponding to a UML process, you will see that it contains a very large number of entries. Each of these entries is a virtual memory area, and each is typically a page long. If you look at the corresponding maps for the same process inside the UML instance, you will see basically the same areas

of virtual memory, except that they will be much fewer and much larger.

This page-by-page mapping of host memory creates data structures in the host kernel and slows down the process of searching, adding, and deleting these mappings. This, in turn, hurts UML performance.

Ingo's solution to this was to create a new system call, `remap_file_pages`, that allows pages within one of these virtual memory areas to be rearranged. Thus, whenever a page is mapped into a UML process address space, it is moved around beneath the virtual memory area rather than creating a new one. So, there will be only one such area on the host for a UML process rather than hundreds and sometimes thousands.

This patch has a noticeable impact on UML performance. It has been around for a while, and Paolo Giarrusso has recently resurrected it, making it work and splitting it into pieces for easier review by the kernel development team. It is a candidate for inclusion into Andrew Morton's kernel tree. It was sent once but dropped because of clashes with another patch. However, Andrew did encourage Paolo to keep it maintained and resubmit it again.

VCPU

`VCPU` is another of Ingo's patches. This deals with the inefficiency of the `ptrace` interface for intercepting system calls. The idea, which had come up several times before, is to have a single process with a "privileged" context and an "unprivileged" context. The process starts in the privileged context and eventually makes a system call that puts it in the unprivileged context. When it receives a signal or makes a system call, it returns through the original system call back to the privileged context. Then it decides what to do with the signal or system call.

In this case, the UML kernel would be the privileged context and its processes would be unprivileged contexts. The behavior of regaining control when another process makes a system call or receives a signal is exactly what `ptrace` is used for. In this case, the change of control would be a system call return rather than a context switch, reducing the overhead of doing system call and signal interception.

FINAL POINTS

Managing a large UML server requires attention to a number of areas that aren't of great concern with a smaller server. Security requires some care. In order to run a secure installation, I recommend the following guidelines.

- ☞ The host should be running a fairly recent kernel. This will give you the performance enhancements that are trickling into the mainline kernel. Also consider applying some of the other patches I have mentioned. In particular, the file-punching patch, which is currently `MADV_TRUNCATE`, creates a number of new possibilities for UML management and hosting policy.
- ☞ Configure the UML instances carefully. Loadable module support should definitely be disabled, as should `tt` mode support. If access to host filesystems is provided, those filesystems should be bind-mounted into the UML jail. They should also be read-only if possible.
- ☞ Jail the UML instances tightly. The jail should be as minimal as you can make it, consistent with your other goals. I expect that the jailing will never be exercised since I know of no way for anyone to break out of a properly configured UML instance. However, a good jail will provide another level of security in the event of a configuration error or an exploitable escape from UML.

CHAPTER 11

Compiling UML from Source

So far we have been playing with a UML kernel binary that we had no hand in creating. Now we will see what's involved with building a UML binary from source.

The process is exactly the same as building a kernel for the host—download the source, configure it, and then build it. The kernel build procedure can be daunting for someone who has never done it before; if the kernel isn't configured correctly, it is reasonably likely not to recognize all of the system's hardware. If one of the unrecognized devices is the boot disk, the kernel won't even boot.

With UML, things are simpler. There are far fewer configuration options because there is a much greater variety of physical devices possibly present on the host than there are virtual devices available for UML. For example, there is one UML block driver, which can be used to access anything on the host that looks like a file—normal files, disks and disk partitions, CD and DVD drives, floppies, and so on. This one driver is the functional equivalent of a large menu hierarchy on the host. As a result, the number of choices and the depth of menus are far less for UML than for the host architecture, and the configuration process is much less complicated.

A further benefit is that the default UML configuration will build and boot. It may not be exactly what you want, but it will work. To get what you want may take some tweaking of the configuration and rebuilding. If you get a nonworking configuration at some point, you know what change you need to undo in order to get back to a working configuration.

DOWNLOADING UML SOURCE

Before you can build UML from source, you need some sources to build. UML is in the mainline 2.6 kernel tree. The process of releasing patches to mainline is done so as to make the main releases as stable and as functional as possible. Patches are considered ready to submit from my development tree to mainline when they have been sufficiently tested. For an obvious bug fix, this can involve just booting and running a UML with the patch. Other, more intrusive patches have stayed for many months in my development tree before being sent to mainline.

So, one of the main 2.6 kernel releases, normally the later the better, is a good place to start for UML. The "stable" kernel tree maintained by Greg Kroah-Hartman and Chris Wright is another good choice. This tree is the same as the corresponding release from Linus, except for a few additional small, critical bug fixes. There normally aren't any additional UML bug fixes in this tree, so it is usually identical to the release from Linus as far as UML is concerned. Occasionally, there is a UML fix here, which will be clear from the change log, and then this would be the recommended kernel tree to start with.

Both of these kernel trees are available from http://www.kernel.org and its mirrors—the stable tree is actually more accessible than Linus' releases. When there is a stable version of the latest kernel, it is available from the main page of http://www.kernel.org, while in order to get the corresponding Linus release, you need to go digging. Successful digging will produce the directory http://www.kernel.org/pub/linux/kernel/v2.6, where you will find all of Linus' kernels.

Whichever tree you decide to use, it is one download away:

```
host% wget http://kernel.org/pub/linux/kernel/v2.6/\
   linux-2.6.12.5.tar.bz2
--22:49:05--  http://kernel.org/pub/linux/kernel/v2.6/\
   linux-2.6.12.5.tar.bz2
           => `linux-2.6.12.5.tar.bz2'
```

```
Resolving kernel.org... 204.152.191.37, 204.152.191.5
Connecting to kernel.org[204.152.191.37]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 37,398,284 [application/x-bzip2]

100%[====================================>] 37,398,284   \
   471.35K/s    ETA 00:00

22:50:08 (580.21 KB/s) - `linux-2.6.12.5.tar.bz2' saved \
   [37,398,284/37,398,284]
```

This is a compressed `tar` file, so it needs to be uncompressed and unpacked:

```
host% bunzip2 linux-2.6.12.5.tar.bz2
host% tar xf linux-2.6.12.5.tar
```

At this point, we have a `linux-2.6.12.5` subdirectory in which there is a kernel tree.

```
host% cd linux-2.6.12.5
host% ls
COPYING        MAINTAINERS  REPORTING-BUGS  drivers  init    \
  lib  scripts   usr
CREDITS        Makefile     arch            fs       ipc     \
  mm   security
Documentation  README       crypto          include  kernel  \
  net  sound
```

Now the UML tree is ready to be configured and built.

CONFIGURATION

Before describing the various configuration interfaces, I should point out that it is highly recommended to run `defconfig` before doing anything else. I describe exactly why later in this section, but, for now, suffice it to say that doing so will give you a UML configuration that is much more likely to boot and run.

There are a variety of kernel configuration interfaces, ranging from the almost completely hands-off `oldconfig` to the graphical and fairly user-friendly `xconfig`. Here are the major choices.

- ☞ `xconfig` presents a graphical kernel configuration, with a tree view of the configuration on one side. Selecting a branch there displays the options on that branch in another pane. Selecting one of these options displays the help for that option in a third pane.

Clicking repeatedly on an option causes it to cycle through its possible settings. Normally, these choices are Enable versus Disable or Enable versus Modular versus Disable. Enable means that the option is built into the final kernel binary, Disable means that it's simply turned off, and Modular means that the option is compiled into a kernel module that can be inserted into the kernel at some later point. Some options have numeric or string values. Double-clicking on these opens a little pane in which you can type a new value. The main menu for the UML configuration is shown in Figure 11.1.

☞ `menuconfig` presents the same menu organization as text in your terminal window. Navigation is done by using the up and down arrow keys and by typing the highlighted letters as shortcuts. The Tab key cycles around the Select, Exit, and Help buttons at the bottom of the window. Choosing Select enters a submenu—Exit

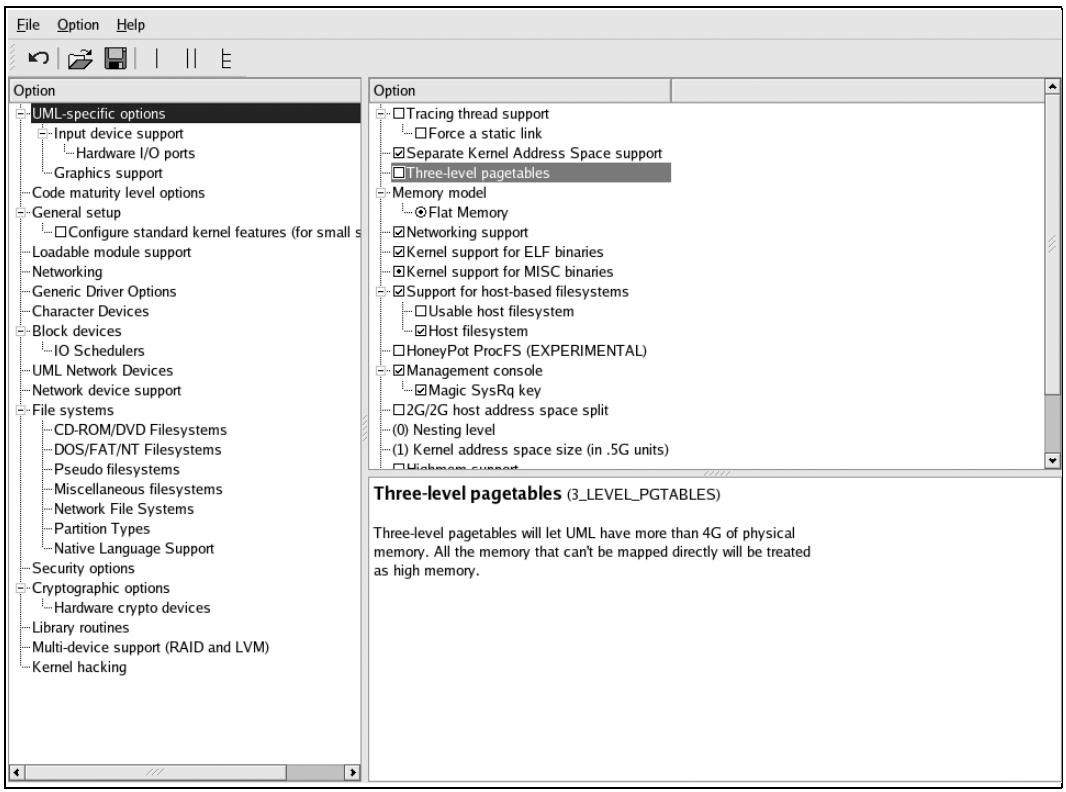


Figure 11.1 The `xconfig` configurator

leaves it and returns to the parent menu. Help displays the help for the current option. Hitting the spacebar will cycle through the settings for the current option. Empty brackets next to an option mean that it is disabled. An asterisk in the brackets means that it is enabled, and an "M" means that it is a module. When you are done choosing options, you select the Exit button repeatedly until you exit the top-level menu and are asked whether to keep this configuration or discard it. Figure 11.2 shows `menuconfig` running in an xterm window displaying the main UML-specific configuration menu.

☞ `config` is the simplest of the interactive configuration options. It asks you about every configuration option, one at a time. On a native x86 kernel, this is good for a soul-deadening afternoon. For UML, it's not nearly as bad, but this is still not the configuration method of choice.

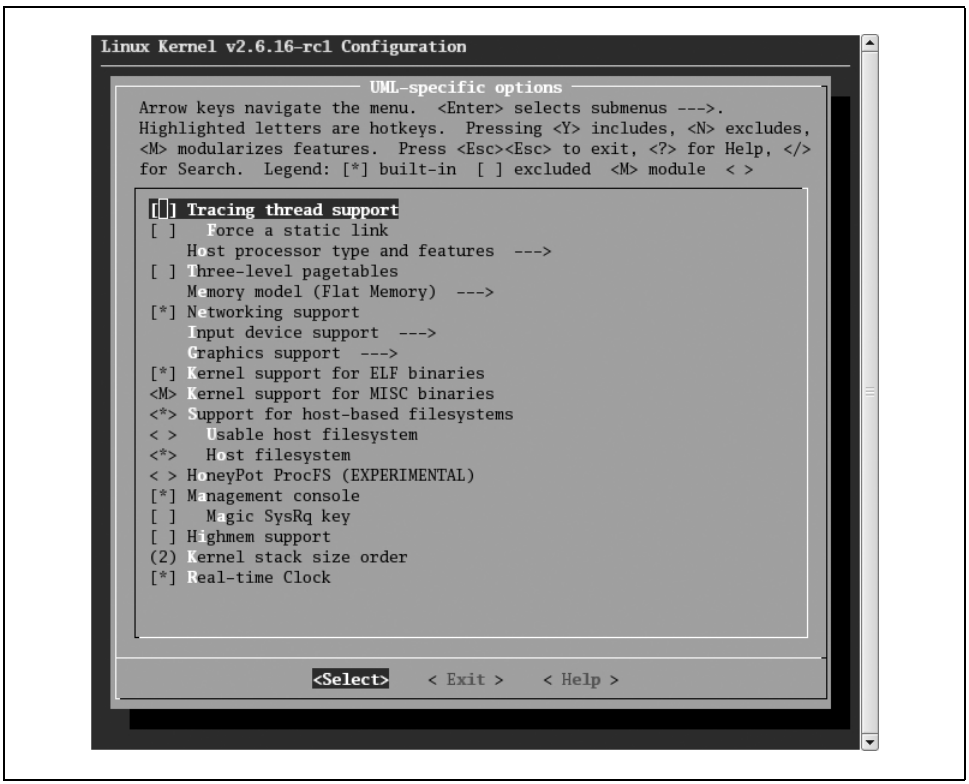


Figure 11.2 The `xconfig` configurator

Some lesser-known configuration choices are useful in some situations.

- ☞ `gconfig` is a graphical configurator that's similar to `xconfig`. It's based on the GTK toolkit (which underlies the GNOME desktop environment) rather than the QT toolkit (which underlies the KDE desktop environment) as `xconfig` is. `gconfig`'s behavior is nearly the same as `xconfig`'s with the exception that checkboxes invite you to click on them, but they do nothing when you do. Instead, there are N, M, and Y columns on the right, as shown in Figure 11.3, which you can click in order to set options.
- ☞ `oldconfig` is one of the mostly noninteractive configurators. It gives you a default configuration, with values taken from `.config` in the kernel tree if it's there, from the host's own configuration, or from the architecture's default configuration when all else fails. It does ask about options for which it does not have defaults.
- ☞ `randconfig` provides a random configuration. This is used to test the kernel build rather than to produce a useful kernel.

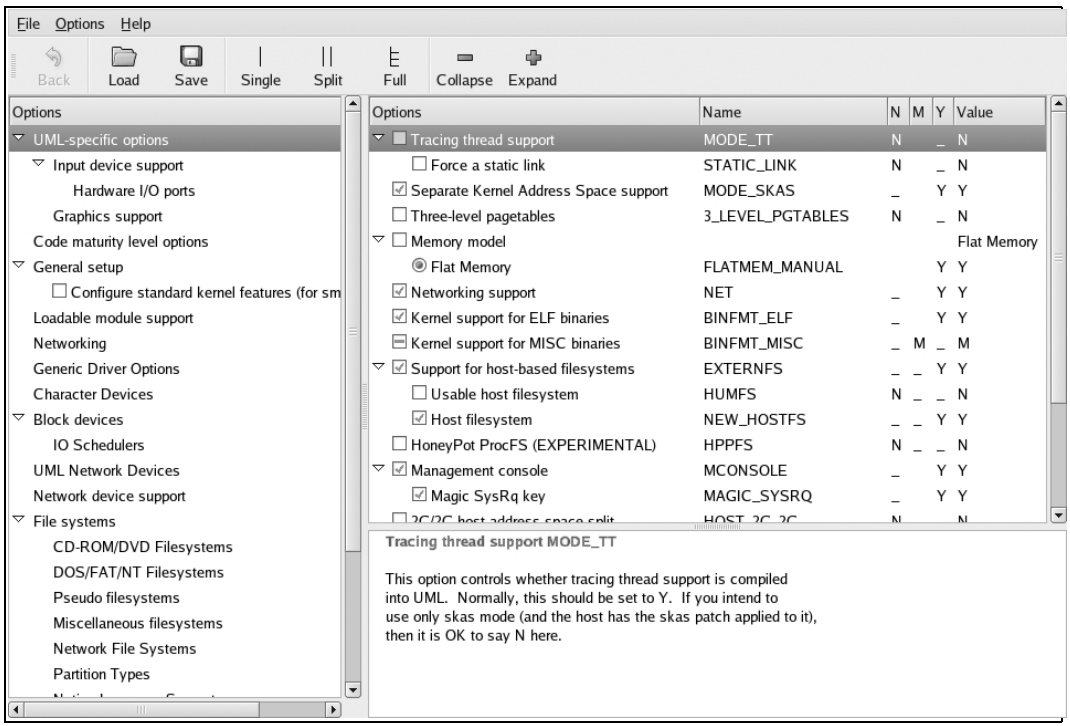


Figure 11.3 The `gconfig` configurator

- ☞ `defconfig` provides a default configuration, using the defaults provided by the architecture.
- ☞ `allmodconfig` provides a configuration in which everything that can be configured as a module is. This is used either to get the smallest possible kernel binary for a given configuration or to test the kernel build procedure.
- ☞ `allnoconfig` provides a configuration with everything possible disabled. This is also used for kernel build testing rather than for producing useful kernels.

One important thing to remember throughout this process is that any time you run `make` in a UML pool, it is essential to put `ARCH=um` on the command line. This is because UML is a different architecture from the host, just like PowerPC (`ppc`) is a different architecture from PC (`i386`). UML's architecture name in the kernel pool is `um`. I dropped the `l` from `uml` because I considered it redundant in a Linux kernel pool.

Because the kernel build procedure will build a kernel for the machine on which it's running unless it's told otherwise, we need to tell it otherwise. This is the purpose of the `ARCH=um` switch—it tells the kernel build procedure to build the `um` architecture, which is UML, rather than the host architecture, which is likely `i386`.

If you forget to add the `ARCH=um` switch at any point, as all of us do once in a while, the tree is likely to be polluted with host architecture data. I clean it up like this:

```
host% make mrproper
host% make mrproper ARCH=um
```

This does a full clean of both UML and the host architectures so that everything that was changed is cleaned up. Then, restart by redoing the configuration.

So, with that in mind, you can configure the UML kernel with something as simple as this:

```
host% make defconfig ARCH=um
```

This will give you the default configuration, which is recommended if this is the first time you are building UML. In this case, feel free to skip over to where we build UML. Otherwise, I'm going to talk about a number of UML-specific configuration options that are useful to know.

I recommend starting with `defconfig` before fine-tuning the UML configuration with another configurator. This is because, when

starting with a clean tree, the other configurators look for a default configuration to start with. Unfortunately, they look in `/boot` and find the configuration for the host kernel, which is entirely unsuitable for UML. Using a configuration that started like this is likely to give you a UML that is missing many essential drivers and won't boot. Running `defconfig` as the first step, before anything else, will start you off with the default UML configuration. This configuration will boot, and the configuration you end up with after customizing it will likely boot as well. If it doesn't, you know what configuration changes you made and which are likely to have caused the problem.

Useful Configuration Options

Execution Mode-Specific Options

A number of configuration options are related to UML's execution mode. These are largely associated with `tt` mode, which may not exist by the time you read this. But it may still be present, and you may have to deal with a version of UML that has it.

- `MODE_TT` and `MODE_SKAS` are the main options controlling UML's execution mode. They decide whether support for `tt` and `skas` modes, respectively, are compiled into the UML kernel. `skas0` mode is part of `MODE_SKAS`. With `MODE_TT` disabled, `tt` mode is not available. Similarly, with `MODE_SKAS` disabled, `skas3` and `skas0` modes are not available. If you know you don't need one or the other, disabling an option will result in a somewhat smaller UML kernel. Having both enabled will produce a UML binary that tests the host's capabilities at runtime and chooses its execution mode accordingly.

  Given a UML with `skas0` support, `MODE_TT` really isn't needed since `skas0` will run on a standard, unmodified host. This used to be the rationale for `tt` mode, but `skas0` mode makes it obsolete. At this point, the only reason for `tt` mode is to see if a UML problem is `skas` specific. In that case, you'd force UML to run in `tt` mode and see if the problem persists. Aside from this, `MODE_TT` can be safely disabled.
- `STATIC_LINK` forces the build to produce a statically linked binary. This is an option only when `MODE_TT` is disabled because a UML kernel with `tt` mode compiled in must be statically linked. With `tt` mode absent, the UML kernel is linked dynamically by

default. However, as we saw in the last chapter, a statically linked binary is sometimes useful, as it simplifies setting up a `chroot` jail for UML.

☞ `NEST_LEVEL` makes it possible to run one UML inside another. This requires a configuration option because UML maps code and data into its own process address spaces in `tt` and `skas0` modes. In `tt` mode, the entire UML kernel is present. In `skas0` mode, there are just the two stub pages.

When the UML process is another instance of UML, they will both want to load that data at the same location in the address space unless something is done to change that. `NEST_LEVEL` changes that. The default value is 0. By changing it to 1, you will build a UML that can run inside another UML instance. It will map its data lower in its process address spaces than the outer UML instance, so they won't conflict with each other.

This is a total nonissue with `skas3` mode since the UML kernel and its processes are in different address spaces. You can run a `skas3` UML inside another UML without needing to specially configure either one.

☞ `HOST_2G_2G` is necessary for running a `tt` or `skas0` UML on hosts that have the 2GB/2GB address space split. With this option enabled on the host, the kernel occupies the upper 2GB of the address space rather than the usual 1GB. This is uncommon but is sometimes done when the host kernel needs more address space for itself than the 1GB it gets by default. This allows the kernel to directly access more physical memory without resorting to Highmem.

The downside of this is that processes get only the lower 2GB of address space, rather than the 3GB they get normally. Since UML puts some data into the top of its process address spaces in both `tt` and `skas0` modes, it will try to access part of the kernel's address space, which is not allowed. The `HOST_2G_2G` option makes this data load into the top of a 2GB address space.

☞ `CMDLINE_ON_HOST` is an option that makes UML management on the host slightly easier. In `tt` mode, UML will make the process names on the host reflect the UML processes running in them, making it easy to see what's happening inside a UML from the host. This is accomplished through a somewhat nasty trick that ensures there is space on the initial UML stack to write this information so that it will be seen on the host. However, this trick, which involves UML changing its arguments and `exec`-ing itself,

confuses some versions of gdb and makes it impossible to debug UML. Since this behavior is specific to tt mode, it is not needed when running in skas mode, even if tt mode support is present in the binary.

This option controls whether the exec takes place. Disabling it will disable the nice process names on the host, but those are present only in tt mode anyway.

☞ PT_PROXY is a tt mode–specific debugging option. Because of the way that UML uses ptrace in tt mode, it is difficult to use gdb to debug it. The tracing thread uses ptrace on all of the other threads, including when they are running in the kernel. gdb uses ptrace in order to control the process it has debugged, and two processes can't simultaneously use ptrace on a single process.

In spite of this, it is possible to run gdb on a UML thread, in a clever but fairly nasty way. The UML tracing thread uses ptrace on gdb, intercepting its system calls. It emulates some of them in order to fake gdb into believing that it has successfully attached to a UML process and is controlling it. In reality, gdb isn't attached to or controlling anything. The UML tracing thread is actually controlling the UML thread, intercepting gdb operations and performing them itself.

This behavior is enabled with the PT_PROXY operation. It gets its name from the ptrace operation proxying that the UML tracing thread does in order to enable running gdb on a tt mode UML. At runtime, this is invoked with the debug switch. This causes the tracing thread to start an xterm window with the captive gdb running inside it.

Debugging a skas mode UML with gdb is much simpler. You can simply start UML under the control of gdb and debug it just as you would any other process.

☞ The KERNEL_HALF_GIGS option controls the amount of address space that a tt mode UML takes for its own use. This is similar to the host 2GB/2GB option mentioned earlier, and the motivation is the same. A larger kernel virtual address space allows it to directly access more physical memory without resorting to Highmem.

The value for this option is an integer, which specifies how many half-gigabyte units of address space that UML will take for itself. The default value is 1—increasing to 2 would cause UML to take the upper 1GB, rather than .5GB, for itself.

In skas mode, with tt mode disabled, this is irrelevant. Since the kernel is in its own address space, it has a full process address space for its own use, and there's no reason to want to carve out a large chunk of its process address spaces.

Generic UML Options

A number of other configuration options don't depend on UML's execution mode. Some of these are shared with other Linux architectures but differ in interesting ways, while others are unique to UML.

☞ The SMP and NR_CPUS options have the same meaning as with any other architecture—SMP controls whether the UML kernel will be able to support multiple processors, and NR_CPUS controls the maximum number of processors the kernel can use.

However, SMP on UML is different enough from SMP on physical hardware to warrant a discussion. Having an SMP virtual machine is completely independent from the host being SMP. An SMP UML instance has multiple virtual processors, which do not map directly to physical processors on the host. Instead, they map directly to processes on the host. If the UML instance has more virtual processors than the host has physical processors, the virtual processors will just be multiplexed on the physical ones by the host scheduler. Even if the host has the same or a greater number of processors than the UML instance, it is likely that the virtual processors will get timesliced on physical processors anyway, due to other demands on the host.

An SMP UML instance can even be run on a uniprocessor host. This will lose the concurrency that's possible on an SMP host, but it does have its uses. Since having multiple virtual processors inside the UML instance translates into an equal number of potentially running processes on the host, a greater number of virtual processors provides a greater call on the host's CPU consumption. A four-CPU UML instance will be able to consume twice as much host CPU time as a two-CPU instance because it has twice as many processes on the host possibly running.

Running an SMP instance on a host with a different number of processors is also useful for kernel debugging. The multiplexing of virtual processors onto physical ones can open up timing holes that wouldn't appear on a physical system. This can expose bugs that would be very hard or impossible to find on physical hardware.

NR_CPUS limits the maximum number of processors that an SMP kernel will support. It does so by controlling the size of some internal data structures that have NR_CPUS elements. Making NR_CPUS unnecessarily large will waste some memory and maybe some CPU time by making the CPU caches less effective but is otherwise harmless.

☞ The HIGHMEM option also means the same thing as it does on the host. If you need more physical memory than can be directly mapped into the kernel's address space, what's left over must be Highmem. It can't be used for as many purposes as the directly mapped memory, and it must be mapped into the kernel's address space when needed and unmapped when it's not. Highmem is perfect for process memory on the host since that doesn't need to be mapped into the kernel's address space.

This is true for tt mode UML instances, as well, since they follow the host's model of having the kernel occupy its process address spaces. However, for skas UML instances, which are in a different address space entirely, kernel access to process memory that has been mapped from the Highmem area is slow. It has to temporarily map the page of memory into its address space before it has access to it. This is one of the few examples of an operation that is faster in tt mode than in skas mode.

The mapping operation is also slower for UML than for the host, making the performance cost of Highmem even greater. However, the need for Highmem is less because of the greater amount of physical memory that can be directly mapped into the skas kernel address space.

☞ The KERNEL_STACK_ORDER option is UML-specific and is somewhat specialized. It was introduced in order to facilitate running valgrind on UML. valgrind creates larger than normal signal frames, and since UML receives interrupts as signals, signal frames plus the normal call stack have to fit on a kernel stack. With valgrind, they often didn't, due to the increased signal frame size.

This was later found to be useful in a few other cases. Some people doing kernel development in UML discovered that their code was overflowing kernel stacks. Increasing the KERNEL_STACK_ORDER parameter is useful in demonstrating that their system crashes are due to stack overflows and not something else, and to allow them to continue working without immediately needing to reduce their stack usage.

- By default, `3_LEVEL_PGTABLES` is disabled on 32-bit architectures and enabled on 64-bit architectures. It is not available to be disabled in the 64-bit case, but it can be enabled for a 32-bit architecture. Doing this provides UML with the capability to access more than 4GB of memory, which is the two-level pagetable limit. This provides a way to experiment with very large physical memory UML instances on 32-bit hosts. However, the portion of this memory that can't be directly mapped will be Highmem, with the performance penalties that I have already mentioned.
- The `UML_REAL_TIME_CLOCK` option controls whether time intervals within UML are made to match real time as much as possible. This matters because the natural way for time to progress within a virtual machine is virtually—that is, time progresses within the virtual machine only when it is actually running on the host. So, if you start a sleep for two seconds inside the UML instance and the host does other things for a few seconds before scheduling the instance, then five seconds or so will pass before the sleep ends. This is correct behavior in a sense—things running within the UML instance will perceive that time flows uniformly, that is, they will see that they can consistently do about the same amount of work in a unit of time. Without this, in the earlier example, a process would perceive the sleep ending immediately because it did no work between the start of the sleep and its end since the host had scheduled something else to run.

  In another sense, this is incorrect behavior. UML instances often have people interacting with them, and those people exist in real time. When someone asks for a five-second pause, it really should end in five real seconds, not five virtual ones. This behavior has actually broken tests. Some Perl regression tests run timers and fail if they take too long to expire. They measure the time difference by using `gettimeofday`, which is tied to the host's `gettimeofday`. When `gettimeofday` is real time and interval timers are virtual, there is bound to be a mismatch.

  So, the `UML_REAL_TIME_CLOCK` option was added to fix this problem. It is enabled by default since that is the behavior that almost everyone wants. However, in some cases it's not desired, so it is a configuration option, rather than hard coded. Intervals are measured by clock ticks, which on UML are timer interrupts from the host. The real-time behavior is implemented by looking at how many ticks should have happened between the last tick and the

current one. Then the generic kernel's timer routine is called that many times. This makes the UML clock catch up with the real one, but it does so in spurts. Time stops for a while, and then it goes forward very quickly to catch up.

When you are debugging UML, you may have it stopped at a `gdb` prompt for a long time. In this case, you don't want the UML instance to spend time in a loop calling the timer routine. For short periods of time, this isn't noticeable. However, if you leave the debugger for a number of hours before continuing it, there will be a noticeable pause while the virtual clock catches up with the real one.

Another case is when you have a UML instance running on a laptop that is suspended overnight. When you wake it up, the UML instance will spend a good amount of time catching up with the many hours of real time it missed. In this case, the UML instance will appear to be hung until it catches up. If either of these situations happens enough to be annoying, and real-time timers aren't critical, you can disable this option.

Virtual Hardware Options

UML has a number of device drivers, each with its own configuration option. I'm going to mention a few of the less obvious ones here.

- The `MCONSOLE` option enables the MConsole driver, which is required in order to control and configure the instance through an MConsole client. This is on by default and should remain enabled unless you have a good reason to not want it.
- The `MAGIC_SYSRQ` option is actually a generic kernel option but is related to `MCONSOLE` through the MConsole `sysrq` command. Without `MAGIC_SYSRQ` enabled, the `sysrq` command won't work.
- The `UML_RANDOM` option enables a "hardware" random number generator for UML. Randomness is often a problem for a server that needs random numbers to seed `ssh` or `https` sessions. Desktop machines can rely on the user for randomness, such as the time between keystrokes or mouse movements. Physical servers rely on randomness, such as the time between I/O interrupts, from their drivers, which is sometimes insufficient.

  Virtual machines have an even harder time since they have fewer sources of randomness than physical machines. It is not uncommon for `ssh` or `https` key generation to hang for a while

until the UML instance acquires enough randomness. The UML random number generator has access to all of the host's randomness from the host's `/dev/random`, rather than having to generate it all itself. If the host has problems providing enough random numbers, key generation and other randomness-consuming operations will still hang. But they won't hang for as long as they would without this driver.

In order to use this effectively, you need to run the `hwrng` tools within the UML instance. This package reads randomness from `/dev/hwrng`, which is attached to this driver, and feeds it into `/dev/random`, from where the randomness is finally consumed.

- The `MMAPPER` option implements a virtual `iomem` driver. This allows a host file to be used as an I/O area that is mapped into the UML instance's physical memory. This specialized option is mostly used for writing emulated drivers and cluster interconnects.
- The `WATCHDOG` and `UML_WATCHDOG` options implement a "hardware" watchdog for UML. The "hardware" portion of it is a process running outside of UML. This process is started when the watchdog device is opened within the UML instance and communicates with the driver through a pipe. It expects to receive some data through that pipe at least every 60 seconds. This happens when the process inside the UML instance that opened the device writes to it. If the external watchdog process doesn't receive input within 60 seconds, it presumes that the UML instance is hung and takes measures to deal with it.

  If it was told on its command line that there is an MConsole `notify` socket, it will send a "hang" notification there. (We saw this in Chapter 8.) Otherwise, it will kill the UML instance itself by sending the main process a sufficiently severe signal to shut it down.

Networking

A number of networking options control how the UML instance can exchange packets with the host and with other UML instances. `UML_NET` enables UML networking—it must be enabled for any network drivers to be available at all. The rest each control a particular packet transport, and their names should be self-explanatory:

- `UML_NET_ETHERTAP`
- `UML_NET_TUNTAP`

- UML_NET_SLIP
- UML_NET_DAEMON
- UML_NET_MCAST
- UML_NET_PCAP
- UML_NET_SLIRP

UML_NET and all of the transport options are enabled by default. Disabling ones that will not be needed will save a small amount of code.

Consoles

A similar set of console and serial line options control how they can be connected to devices on the host. Their names should also be self-explanatory:

- NULL_CHAN
- PORT_CHAN
- PTY_CHAN
- TTY_CHAN
- XTERM_CHAN

The file descriptor channel, which, by default, the main console uses to attach itself to stdin and stdout, is not configurable. It is always on because people were constantly disabling it and sending mail to the UML mailing lists wondering why UML wouldn't boot.

There is an option, SSL, to enable UML serial line support. Serial lines aren't much different from consoles, so having them doesn't do much more than add some variety to the device names through which you can attach to a UML instance.

Finally, the default settings for console zero, the rest of the consoles, and the serial lines are all configurable. These values are strings, and describe what host device the UML devices should be attached to. These, and their default values, are as follows:

- CON_ZERO_CHAN—fd:0,fd:1
- CON_CHAN—xterm
- SSL_CHAN—pty

Debugging

I talked about a number of debugging options in the context of `tt` mode already since they are specific to `tt` mode. A few others allow UML to be profiled by the `gprof` and `gcov` tools. These work only in `skas` mode since a `tt` mode UML instance breaks assumptions made by them.

The `GPROF` option enables `gprof` support in the UML build, and the `GCOV` option similarly enables `gcov` support. These change the compilation flags so as to tell the compiler to generate the code needed for the profiling. In the case of `gprof`, the generated code tracks procedure calls and keeps statistical information about where the UML instance is spending its time. The code generated for `gcov` tracks what blocks of code have been executed and how many times they were executed.

A UML profiling run is just like any other process. You start it, exercise it for a while, stop it, and generate the statistics you want. In the case of UML, the profiling starts when UML boots and ends when it shuts down. Running `gprof` or `gcov` after that is exactly like running it on any other application.

COMPILATION

Now that the UML has been configured, it is time to build it. On 2.6 hosts, we need to take care of one more detail. If the UML instance is to be built to use AIO support on the host, a header file, `include/linux/aio_abi.h` in the UML tree, must be copied to `/usr/include/linux/aio_abi.h` on the host.

With this taken care of, building UML is as simple as this:

```
host% make ARCH=um
```

If you have built Linux kernels before, you will see that the UML build is very similar to what you have seen before. When it finishes, you will get two identical files, called `vmlinux` and `linux`. In fact, they are hard links to the same file. Traditionally, the UML build produced a file called `linux` rather than the `vmlinux` or `vmlinuz` that the kernel build normally produces. I did this on purpose, believing that having the binary be named `linux` was more intuitive than `vmlinux` or `vmlinuz`.

This was true, and most people like the name, but some kernel hackers are very used to an output file named `vmlinux`. Also, the kernel

build became stricter over time, and it became very hard to avoid having a final binary named `vmlinux`. So, I made the UML build produce the `vmlinux` file, and as a final step, link the name `linux` to that file. This way, everyone is happy.

CHAPTER 12

Specialized UML Configurations

So far we have seen UML instances with fairly normal virtual hardware configurations—they are similar to common physical machines. Now we will look at using UML to emulate unusual configurations that can't even be approached with common hardware. This includes configurations with lots of devices, such as block devices and network interfaces, many CPUs, and very large physical memory, and more specialized configurations, such as clusters.

By virtualizing hardware, UML makes it easy to simulate these configurations. Virtual devices can be constructed as long as host and UML memory hold out and no built-in software limits are reached. There are no constraints such as those based on the number of slots on a bus or the number of buses on a system.

UML can also emulate hardware you might not even have one instance of. We'll see an example of this when we build a cluster, which will need a shared storage device. Physically, this is a disk that is somehow multiported, either because it is multiported itself or because it's on a shared bus. Either way, this is an expensive, noncommodity device. However, with UML, a shared device is simply a file on the host to which multiple UML instances have access.

LARGE NUMBERS OF DEVICES

We'll start by configuring a UML instance with a large number of devices. The reasons for wanting to do this vary. For many people, there is value in looking at `/proc/meminfo` and seeing an absurdly large amount of memory, or running `df` and seeing more disk space than you could fit in a room full of disks.

More seriously, it allows you to explore the scalability limits of the Linux kernel and the applications running on it. This is useful when you are maintaining some software that may run into these limits, and your users may have hardware that may do so, but you don't. You can emulate the large configuration to see how your software reacts to it.

You may also be considering acquiring a very large machine but want to know whether it is fully usable by Linux and the applications you envision running on it. UML will let you explore the software limitations. Obviously, any hardware limitations, such as the number of bus slots and controllers and the like, can't be explored in this way.

Network Interfaces

Let's start by configuring a pair of UML instances with a large number of network interfaces. We will boot the two instances, `debian1` and `debian2`, and hot-plug the interfaces into them. So, with the UML instances booted, you do this as follows:

```
host% for i in `seq 0 127`; do uml_mconsole debian1 \
config eth$i=mcast,,224.0.0.$i; done
host% for i in `seq 0 127`; do uml_mconsole debian2 \
config eth$i=mcast,,224.0.0.$i; done
```

These two lines of shell configure 128 network interfaces in each UML instance. You'll see a string of `OK` messages from each of these, plus a lot of console output in the UML instances if kernel output is logged there. Running `dmesg` in one of the instances will show you something like this:

```
Netdevice 124 : mcast backend multicast address: \
224.0.0.124:1102, TTL:1
Configured mcast device: 224.0.0.125:1102-1
Netdevice 125 : mcast backend multicast address: \
224.0.0.125:1102, TTL:1
Configured mcast device: 224.0.0.126:1102-1
Netdevice 126 : mcast backend multicast address: \
```

```
224.0.0.126:1102, TTL:1
Configured mcast device: 224.0.0.127:1102-1
Netdevice 127 : mcast backend multicast address: \
224.0.0.127:1102, TTL:1
```

Running `ifconfig` inside the UML instances will confirm that interfaces `eth0` through `eth127` now exist. If you're brave, run `ifconfig -a`. Otherwise, just do some spot-checking:

```
UML# ifconfig eth120
eth120    Link encap:Ethernet  HWaddr 00:00:00:00:00:00
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 \
frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 \
carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
          Interrupt:5
```

This indicates that we indeed have the network interfaces we asked for. I configured them to attach to multicast networks on the host, so they will be used purely to network between the two instances. They can't talk directly to the outside network unless you configure one of the instances with an interface attached to a TUN/TAP device and use it as a gateway. Each of an instance's interfaces is attached to a different host multicast address, which means they are on different networks. So, taken in pairs, the corresponding interfaces on the two instances are on the same network and can communicate with each other.

For example, the two `eth0` interfaces are both attached to the host multicast IP address `224.0.0.0` and thus will see each other's packets. The two `eth1` interfaces are on `224.0.0.1` and can see each other's packets, but they won't see any packets from the `eth0` interfaces.

Next, we configure the interfaces inside the UML instances. I'm going to put each one on a different network in order to correspond to the connectivity imposed by the multicast configuration on the host. The `eth0` interfaces will be on the `10.0.0.0/24` network, the `eth1` interfaces will be on the `10.0.1.0/24` network, and so forth:

```
UML1# for i in `seq 0 127`; do ifconfig eth$i 10.0.$i.1/24 up; done
UML2# for i in `seq 0 127`; do ifconfig eth$i 10.0.$i.2/24 up; done
```

Now the interfaces in the first UML instance are running and have the `.1` addresses in their networks, and the interfaces in the second

instance have the .2 addresses. Again, some spot-checking will confirm this:

```
UML1# ifconfig eth75
eth75     Link encap:Ethernet  HWaddr FE:FD:0A:00:4B:01
          inet addr:10.0.75.1  Bcast:10.255.255.255  \
Mask:255.0.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  \
Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 \
frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 \
carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
          Interrupt:5

UML2# ifconfig eth100
eth100    Link encap:Ethernet  HWaddr FE:FD:0A:00:64:02
          inet addr:10.0.100.2  Bcast:10.255.255.255  \
Mask:255.0.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  \
Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 \
frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 \
carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
          Interrupt:5
```

Let's see if the interfaces work:

```
UML1# ping 10.0.50.2
PING 10.0.50.2 (10.0.50.2): 56 data bytes
64 bytes from 10.0.50.2: icmp_seq=0 ttl=64 time=56.3 ms
64 bytes from 10.0.50.2: icmp_seq=1 ttl=64 time=15.7 ms
64 bytes from 10.0.50.2: icmp_seq=2 ttl=64 time=16.6 ms
64 bytes from 10.0.50.2: icmp_seq=3 ttl=64 time=14.9 ms
64 bytes from 10.0.50.2: icmp_seq=4 ttl=64 time=16.4 ms

--- 10.0.50.2 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 14.9/23.9/56.3 ms
```

You can try some of the others by hand or check all of them with a bit of shell such as this:

```
UML1# for i in `seq 0 127`; do ping -c 1 10.0.$i.2 ; done
```

This exercise is fun and interesting, but what's the practical use? We have demonstrated that there appears to be no limit, aside from memory, on how many network interfaces Linux will support. To tell for sure, we would need to look at the kernel source. But if you are seriously asking this sort of question, you probably have some hardware limit in mind, and setting up some virtual machines is a quick way to tell whether the operating system or the networking tools have a lower limit.

By poking around a bit more, we can see that other parts of the system are being exercised. Taking a look at the routing table will show you one route for every device we configured. An excerpt looks like this:

```
UML1# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric \
    Ref    Use Iface
10.0.20.0       0.0.0.0         255.255.255.0   U     0      \
    0        0 eth20
10.0.21.0       0.0.0.0         255.255.255.0   U     0      \
    0        0 eth21
10.0.22.0       0.0.0.0         255.255.255.0   U     0      \
    0        0 eth22
10.0.23.0       0.0.0.0         255.255.255.0   U     0      \
    0        0 eth23
```

This would be interesting if you wanted a large number of networks, rather than simply a large number of interfaces.

Similarly, we are exercising the arp cache more than usual. Here is an excerpt:

```
UML# arp -an
? (10.0.126.2) at FE:FD:0A:00:7E:02 [ether] on eth126
? (10.0.64.2) at FE:FD:0A:00:40:02 [ether] on eth64
? (10.0.110.2) at FE:FD:0A:00:6E:02 [ether] on eth110
? (10.0.46.2) at FE:FD:0A:00:2E:02 [ether] on eth46
? (10.0.111.2) at FE:FD:0A:00:6F:02 [ether] on eth111
```

This all demonstrates that, if there are any hard limits in the Linux networking subsystem, they are reasonably high. A related but different question is whether there are any problems with performance scaling to this many interfaces and networks. If you are concerned about this, you probably have a particular application or workload in mind and would do well to run it inside a UML instance, varying the number of interfaces, networks, routes, or whatever its performance depends on.

For demonstration purposes, since I lack such a workload, I will use standard system tools to see how well performance scales as the number of interfaces increases.

Let's look at ping times as the number of interfaces increases. I'll shut down all of the Ethernet devices and bring up an increasing number on each test. The first two rounds look like this:

```
UML# export n=0 ; for i in `seq 0 $n`; \
    do ifconfig eth$i 10.0.$i.1/24 up; done ; \
    for i in `seq 0 $n`; do ping -c 2 10.0.$i.2 ; done ; \
    for i in `seq 0 $n`; do ifconfig eth$i down ; done
PING 10.0.0.2 (10.0.0.2): 56 data bytes
64 bytes from 10.0.0.2: icmp_seq=0 ttl=64 time=36.0 ms
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=4.9 ms

--- 10.0.0.2 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 4.9/20.4/36.0 ms
UML# export n=1 ; for i in `seq 0 $n`; \
    do ifconfig eth$i 10.0.$i.1/24 up; done ; \
    for i in `seq 0 $n`; do ping -c 2 10.0.$i.2 ; \
    done ; for i in `seq 0 $n`; do ifconfig eth$i down ; done
PING 10.0.0.2 (10.0.0.2): 56 data bytes
64 bytes from 10.0.0.2: icmp_seq=0 ttl=64 time=34.0 ms
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=4.9 ms

--- 10.0.0.2 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 4.9/19.4/34.0 ms
PING 10.0.1.2 (10.0.1.2): 56 data bytes
64 bytes from 10.0.1.2: icmp_seq=0 ttl=64 time=35.4 ms
64 bytes from 10.0.1.2: icmp_seq=1 ttl=64 time=5.0 ms

--- 10.0.1.2 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 5.0/20.2/35.4 ms
```

The two-interface ping times are essentially the same as the one-interface times. We are looking at how the times change, rather than their actual values compared to ping times on the host. A virtual machine will necessarily have different performance characteristics than a physical one, but they should scale similarly.

We see the first ping taking much longer than the second because of the arp request and response that have to occur before any ping requests can be sent out. The sending system needs to determine the Ethernet MAC address corresponding to the IP address you are pinging. This requires an arp request to be broadcast and a reply to come

back from the target host before the actual ping request can be sent. The second ping time measures the actual time of a ping round trip.

I won't bore you with the full output of repeating this, doubling the number of interfaces at each step. However, this is typical of the times I got with 128 interfaces:

```
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 6.7/22.7/38.8 ms
PING 10.0.123.2 (10.0.123.2): 56 data bytes
64 bytes from 10.0.123.2: icmp_seq=0 ttl=64 time=39.1 ms
64 bytes from 10.0.123.2: icmp_seq=1 ttl=64 time=8.9 ms

--- 10.0.123.2 ping statistics ---
```

With 128 interfaces, both ping times are around 4 ms greater than with one. This suggests that the slowdown is in the IP routing code since this is exercised once for each packet. The arp requests don't go through the IP stack, so they wouldn't be affected by any slowdowns in the routing code.

The 4-ms slowdown is comparable to the fastest ping time, which was around 5 ms, suggesting that the routing overhead with 128 networks and 128 routes is comparable to the ping round trip time.

In real life, you're unlikely to be interested in how fast pings go when you have a lot of interfaces, routes, arp table entries, and so on. You're more likely to have a workload that needs to operate in an environment with these sorts of scalability requirements. In this case, instead of running pings with varying numbers of interfaces, you'd run your workload, changing the number of interfaces as needed, and make sure it behaves acceptably within the range you plan for your hardware.

Memory

Memory is another physical asset that a system may have a lot of. Even though it's far cheaper than it used to be, outfitting a machine with many gigabytes is still fairly pricy. You may still want to emulate a large-memory environment before splashing out on the actual physical article. Doing so may help you decide whether your workload will benefit from having lots of memory, and if so, how much memory you need. You can determine your memory sweet spot so you spend enough on memory but not too much.

You may have guessed by now that we are going to look at large-memory UML instances, and you'd be right. To start with, here is `/proc/meminfo` from a 64GB UML instance:

```
UML# more /proc/meminfo
MemTotal:     65074432 kB
MemFree:      65048744 kB
Buffers:            824 kB
Cached:            9272 kB
SwapCached:           0 kB
Active:            5252 kB
Inactive:          6016 kB
HighTotal:            0 kB
HighFree:             0 kB
LowTotal:     65074432 kB
LowFree:      65048744 kB
SwapTotal:            0 kB
SwapFree:             0 kB
Dirty:              112 kB
Writeback:            0 kB
Mapped:            2772 kB
Slab:              4724 kB
CommitLimit:  32537216 kB
Committed_AS:      4064 kB
PageTables:         224 kB
VmallocTotal: 137370258416 kB
VmallocUsed:          0 kB
VmallocChunk: 137370258416 kB
```

This output is from an x86_64 UML on a 1GB host. Since x86_64 is a 64-bit architecture, there is plenty of address space for UML to map many gigabytes of physical memory. In contrast, x86, as a 32-bit architecture, doesn't have sufficient address space to cleanly handle large amounts of memory. On x86, UML must use the kernel's Highmem support in order to handle greater than about 3GB of physical memory. This works, but, as I discussed in Chapter 9, there's a large performance penalty to pay because of the requirement to map the high memory into low memory where the kernel can directly access it.

On an x86 UML instance, the `meminfo` output would have a large amount of Highmem in the `HighTotal` and `HighFree` fields. On 64-bit hosts, this is unnecessary, and all the memory appears as `LowTotal` and `LowFree`. The other unusual feature here is the even larger amount of `vmalloc` space, 137 terabytes. This is simply the address space that the UML instance doesn't have any other use for.

There has to be more merit to large-memory UML instances than impressive numbers in `/proc/meminfo`. That's enough for me, but

other people seem to be more demanding. A more legitimate excuse for this sort of exercise is to see how the performance of a workload or application will change when given a large amount of memory.

In order to do this, we need to be able to judge the performance of a workload in a given amount of memory. On a physical machine, this would be a matter of running it and watching the clock on the nearest wall. Having larger amounts of memory improves performance by allowing more data to be stored in memory, rather than on disk. With insufficient memory, the system has to swap data to disk when it's unused and swap it back in when it is referenced again. Some intelligent applications, such as databases, do their own caching based on the amount of memory in the system. In this case, the trade-off is usually still against storing data in memory. For example, a database will read more index data from disk when it has enough memory, speeding lookups.

In the example above, the 64GB UML instance is running on a 1GB host. It's obviously not manufacturing 63GB of memory, so that extra memory is ultimately backed by disk. You can run applications that consume large amounts of memory, and the UML instance will not have to use its own swap. However, since this will exceed the amount of memory on the host, it will start swapping. This means you can't watch the clock in order to decide how your workload will perform with a lot of memory available.

Instead, you need to find a proxy for performance. A proxy is a measurement that can stand in for the thing you are really interested in when that thing can't be measured directly. I've been talking about disk I/O, either by the system swapping or by the application reading in data on its own. So, watching the UML instance's disk I/O is a good way to decide whether the workload's performance will improve. The greater the decrease in disk traffic, the greater the performance improvement you can expect.

As with increasing amounts of any resource, there will be a point of diminishing returns, where adding an increment of memory results in a smaller performance increase than the previous increment did. Graphing performance against memory will typically show a relatively narrow region where the performance levels off. It may still increase, but suddenly at a slower rate than before. This performance "knee" is usually what you aim at when you design a system. Sometimes the knee is too expensive or is unattainable, and you add as much memory as you can, accepting a performance point below the knee. In other cases, you need as much performance as you can get, and you accept the diminishing performance returns with much of the added memory.

As before, I'm going to use a little fake workload in order to demonstrate the techniques involved. I will create a database-like workload with a million small files. The file metadata—the file names, sizes, modification dates, and so on—will stand in for the database indexes, and their contents will stand in for the actual data. I need such a large number of files so that their metadata will occupy a respectable amount of memory. This will allow us to measure how changing the amount of system memory impacts performance when searching these files.

The following procedure creates the million files in three stages, increasing the number by a factor of 100 at each step:

- First, copy 1024 characters from `/etc/passwd` into the file `0` and make 99 copies of it in the files `1` through `99`.
- Next, create a subdirectory, move those files into it, and make 99 copies, creating 10,000 files.
- Repeat this, creating 99 more copies of the current directory, leaving us with a million files, containing 1024 characters apiece.

```
UML# mkdir test
UML# cd test
UML# dd if=/etc/passwd count=1024 bs=1 > 0
1024+0 records in
1024+0 records out
UML# for n in `seq 99` ; do cp 0 $n; done
UML# ls
1   14  19  23  28  32  37  41  46  50  55  6   \
    64  69  73  78  82  87  91  96
10  15  2   24  29  33  38  42  47  51  56  60  \
    65  7   74  79  83  88  92  97
11  16  20  25  3   34  39  43  48  52  57  61  \
    66  70  75  8   84  89  93  98
12  17  21  26  30  35  4   44  49  53  58  62  \
    67  71  76  80  85  9   94  99
13  18  22  27  31  36  40  45  5   54  59  63  \
    68  72  77  81  86  90  95  0
UML# mkdir a
UML# mv * a
mv: cannot move `a' to a subdirectory of itself, `a/a'
UML# mv a 0
UML# for n in `seq 99` ; do cp -a 0 $n; done
UML# mkdir a
UML# mv * a
mv: cannot move `a' to a subdirectory of itself, `a/a'
UML# mv a 0
UML# for n in `seq 99` ; do cp -a 0 $n; done
```

Now let's reboot in order to get some clean memory consumption data. On reboot, log in, and look at `/proc/diskstats` in order to see how much data was read from disk during the boot:

```
UML# cat /proc/diskstats
  98    0 ubda 375 221 18798 2860 55 111 1328 150 0 2740 3010
```

The sixth field (`18798`, in this case) is the number of sectors read from the disk so far. With 512-byte sectors, this means that the boot read around 9.6MB (9624576 bytes, to be exact).

Now, to see how much memory we need in order to search the metadata of the directory hierarchy, let's run a `find` over it:

```
UML# cd test
UML# find . > /dev/null
```

Let's look at `diskstats` again, using `awk` to pick out the correct field so as to avoid taxing our brains by having to count up to six:

```
UML# awk '{ print $6 }' /proc/diskstats
214294
UML# echo $[ (214214 - 18798) * 512 ]
100052992
```

This pulled in about 100MB of disk space. Any amount of memory much more than that will be plenty to hold all of the metadata we will need. To check this, we can run the `find` again and see that there isn't much disk input:

```
UML# awk '{ print $6 }' /proc/diskstats
215574
UML# find . > /dev/null
UML# awk '{ print $6 }' /proc/diskstats
215670
```

So, there wasn't much disk I/O, as expected.

To see how much total memory would be required to run this little workload, let's look at `/proc/meminfo`:

```
UML# grep Mem /proc/meminfo
MemTotal:      1014032 kB
MemFree:        870404 kB
```

A total of 143MB of memory has been consumed so far. Anything over that should be able to hold the full set of metadata. We can check this by rebooting with 160MB of physical memory:

```
UML# cd test
UML# awk '{ print $6 }' /proc/diskstats
18886
UML# find . > /dev/null
UML# awk '{ print $6 }' /proc/diskstats
215390
UML# find . > /dev/null
UML# awk '{ print $6 }' /proc/diskstats
215478
UML# grep Mem /proc/meminfo
MemTotal:       156276 kB
MemFree:         15684 kB
```

This turns out to be correct. We had essentially no disk reads on the second search and pretty close to no free memory afterward.

We can check this by booting with a lot less memory and seeing if there is a lot more disk activity on the second find. With an 80MB UML instance, there was about 90MB of disk activity between the two searches. This indicates that 80MB was not enough memory for optimal performance in this case, and a lot of data that was cached during the first search had to be discarded and read in again during the second. On a physical machine, this would result in a significant performance loss. On a virtual machine, it wouldn't necessarily, depending on how well the host is caching data. Even if the UML instance is swapping, the performance loss may not be nearly as great as on a physical machine. If the host is caching the data that the UML instance is swapping, then swapping the data back in to the UML instance involves no disk activity, in contrast to the case with a physical machine. In this case, swapping would result in a performance loss for the UML instance, but a lot less than you would expect for a physical system.

We measured the difference between an 80MB UML instance and a 160MB one, which are very far from the 64MB instance with which I started. These memory sizes are easily reached with physical systems today (it would be hard to buy a system with less than many times as much memory as this), and this difference could easily have been tested on a physical system.

To get back into the range of memory sizes that aren't so easily reached with a physical machine, we need to start searching the data. My million files, plus the rest of the files that were already present, occupy about 6.5GB.

With a 1GB UML instance, there are about 5.5GB of disk I/O on the first search and about the same on the second, indicating that this is not nearly enough memory and that there is less actual data being read from the disk than `df` would have us believe:

```
UML# awk '{ print $6 }' /proc/diskstats
18934
UML# find . -xdev -type f | xargs cat > /dev/null
UML# awk '{ print $6 }' /proc/diskstats
11033694
UML# find . -xdev -type f | xargs cat > /dev/null
UML# awk '{ print $6 }' /proc/diskstats
22050006
UML# echo $[ (11033694 - 18934) * 512 ]
5639557120
UML# echo $[ (22050006 - 11033694) * 512 ]
5640351744
```

With a 4GB UML instance, we might expect the situation to improve, but with still a noticeable amount of disk activity on the second search.

```
UML# awk '{ print $6 }' /proc/diskstats
89944
UML# find / -xdev -type f | xargs cat > /dev/null
UML# awk '{print $6}' /proc/diskstats
13187496
UML# echo $[ 13187496 * 512 ]
6751997952
UML# awk '{print $6}' /proc/diskstats
26229664
UML# echo $[ (26229664 - 13187496) * 512 ]
6677590016
```

Actually, there is no improvement—there was just as much input during the second search as during the first. In retrospect, this shouldn't be surprising. While a lot of the data could have been cached, it wasn't because the kernel had no way to know that it was going to be used again. So, the data was thrown out in order to make room for data that was read in later.

In situations like this, the performance knee is very sharp—you may see no improvement with increasing memory until the workload's entire dataset can be held in memory. At that point, there will likely be a very large performance improvement. So, rather than the continuous performance curve you might expect, you would get something more like a sudden jump at the magic amount of memory that holds all of the data the workload will need.

We can check this by booting a UML instance with more than about 6.5GB of memory. Here are the results with a 7GB instance:

```
UML# awk '{print $6}' /proc/diskstats
19928
UML# find / -xdev -type f | xargs cat > /dev/null
```

```
UML# awk '{print $6}' /proc/diskstats
13055768
UML# echo $[ (13055768 - 19928) * 512 ]
6674350080
UML# find / -xdev -type f | xargs cat > /dev/null
UML# awk '{print $6}' /proc/diskstats
14125882
UML# echo $[ (14125882 - 13055768) * 512 ]
547898368
```

We had about a half gigabyte of data read in from disk on the second run, which I don't really understand. However, this is far less than we had with the smaller memory instances. On a physical system, this would have translated into much better performance. The UML instance didn't run any faster with more memory because real time is going to depend on real resources. The real resource in this case is physical memory on the host, which was the same for all of these tests. In fact, the larger memory instances performed noticeably worse than the smaller ones. The smallest instance could just about be held in the host's memory, so its disk I/O was just reading data on behalf of the UML instance. The larger instances couldn't be held in the host's memory, so there was that I/O, plus the host had to swap a large amount of the instance itself in and out.

This emphasizes the fact that, in measuring performance as you adjust the virtual hardware, you should not look at the clock on the wall. You should find some quantity within the UML instance that will correlate with performance of a physical system with that hardware running the same workload. Normally, this is disk I/O because that's generally the source for all the data that's going to fill your memory. However, if the data is coming from the network, and increasing memory would be expected to reduce network use, then you would look at packet counts rather than disk I/O.

If you were doing this for real in order to determine how much memory your workload needs for good performance, you wouldn't have created a million small files and run `find` over them. Instead, you'd copy your actual workload into a UML instance and boot it with varying amounts of memory. A good way to get an approximate number for the memory it needs is to boot with a truly large amount of memory, run the workload, and see how much data was read from disk. A UML instance with that amount of memory, plus whatever it needs during boot, will very likely not need to swap out any data or read anything twice.

However, this approximation may overstate the amount of memory you need for decent performance—a good amount of it may be hold-

ing data that is not important for performance. So, it would also be a good idea, after checking this first amount of memory to see that it gives you good performance, to decrease the memory size until you see an increase in disk reads. At this point, the UML instance can't hold all of the data that is needed for good performance.

This, plus a bit more, is the amount of memory you should aim at with your physical system. There may be reasons it can't be reached, such as it being too expensive or the system not being able to hold that much. In this case, you need to accept lower than optimal performance, or take some more radical steps such as reworking the application to require less memory or spreading it across several machines, as with a cluster. You can use UML to test this, as well.

CLUSTERS

Clusters are another area where we are going to see increasing amounts of interest and activity. At some point, you may have a situation where you need to know whether your workload would benefit in some way from running on a cluster.

I am going to set up a small UML cluster, using Oracle's `ocfs2` to demonstrate it. The key part of this, which is not common as hardware, is a shared storage device. For UML, this is simply a file on the host that multiple UML instances can share. In hardware, this would require a shared bus of some sort, which you quite likely don't have and which would be expensive to buy, especially for testing. Since UML requires only a file on the host, using it for cluster experiments is much more convenient and less expensive.

Getting Started

First, since `ocfs2` is somewhat experimental (it is in Andrew Morton's `-mm` tree, not in Linus' mainline tree at this writing), you will likely need to reconfigure and rebuild your UML kernel. Second, procedures for configuring a cluster may change, so I recommend getting Oracle's current documentation. The user guide is available from http://oss.oracle.com/projects/ocfs2/.

The `ocfs2` configuration script requires that everything related to `ocfs2` be built as modules, rather than just being compiled into the kernel. This means enabling `ocfs2` (in the Filesystems menu) and

configfs (which is the "Userspace-driven configuration filesystem" item in the Pseudo Filesystems submenu). These options both need to be set to "M."

After building the kernel and modules, you need to copy the modules into the UML filesystem you will be using. The easiest way to do this is to loopback-mount the filesystem on the host (at ./rootfs, in this example) and install the modules into it directly:

```
host% mkdir rootfs
host# mount root_fs.cluster rootfs -o loop
host# make modules_install INSTALL_MOD_PATH=`pwd`/rootfs
  INSTALL fs/configfs/configfs.ko
  INSTALL fs/isofs/isofs.ko
  INSTALL fs/ocfs2/cluster/ocfs2_nodemanager.ko
  INSTALL fs/ocfs2/dlm/ocfs2_dlm.ko
  INSTALL fs/ocfs2/dlm/ocfs2_dlmfs.ko
  INSTALL fs/ocfs2/ocfs2.ko
host# umount rootfs
```

You can also install the modules into an empty directory, create a tar file of it, copy that into the running UML instance over the network, and untar it, which is what I normally do, as complicated as it sounds.

Once you have the modules installed, it is time to set things up within the UML instance. Boot it on the filesystem you just installed the modules into, and log into it. We need to install the ocfs2 utilities, which I got from http://oss.oracle.com/projects/ocfs2-tools/. There's a Downloads link from which the source code is available. You may wish to see if your UML root filesystem already has the utilities installed, in which case you can skip down to setting up the cluster configuration file.

My system doesn't have the utilities, so, after setting up the network, I grabbed the 1.1.2 version of the tools:

```
UML# wget http://oss.oracle.com/projects/ocfs2-tools/dist/\
files/source/v1.1/ocfs2-tools-1.1.2.tar.gz
UML# gunzip ocfs2-tools-1.1.2.tar.gz
UML# tar xf ocfs2-tools-1.1.2.tar
UML# cd ocfs2-tools-1.1.2
UML# ./configure
```

I'll spare you the configure output; I had to install a few packages, such as e2fsprogs-devel (for libcom_err.so), readline-devel, and glibc2-devel. I didn't install the python development package, which is needed for the graphical ocfs2console. I'll be demonstrating everything on the command line, so we won't need that.

After configuring `ocfs2`, we do the usual `make` and `install`:

```
UML# make && make install
```

`install` will put things under `/usr/local` unless you configured it differently.

At this point, we can do some basic checking by looking at the cluster status and loading the necessary modules. The guide I'm reading refers to the control script as `/etc/init.d/o2cb`, which I don't have. Instead, I have `./vendor/common/o2cb.init` in the source directory, which seems to behave as the fictional `/etc/init.d/o2cb`.

```
UML# ./vendor/common/o2cb.init status
Module "configfs": Not loaded
Filesystem "configfs": Not mounted
Module "ocfs2_nodemanager": Not loaded
Module "ocfs2_dlm": Not loaded
Module "ocfs2_dlmfs": Not loaded
Filesystem "ocfs2_dlmfs": Not mounted
```

Nothing is loaded or mounted. The script makes it easy to change this:

```
UML# ./vendor/common/o2cb.init load
Loading module "configfs": OK
Mounting configfs filesystem at /config: OK
Loading module "ocfs2_nodemanager": OK
Loading module "ocfs2_dlm": OK
Loading module "ocfs2_dlmfs": OCFS2 User DLM kernel \
    interface loaded
OK
Mounting ocfs2_dlmfs filesystem at /dlm: OK
```

We can check that the status has now changed:

```
UML# ./vendor/common/o2cb.init status
Module "configfs": Loaded
Filesystem "configfs": Mounted
Module "ocfs2_nodemanager": Loaded
Module "ocfs2_dlm": Loaded
Module "ocfs2_dlmfs": Loaded
Filesystem "ocfs2_dlmfs": Mounted
```

Everything looks good. Now we need to set up the cluster configuration file. There is a template in `documentation/samples/cluster.conf`, which I copied to `/etc/ocfs2/cluster.conf` after creating `/etc/ocfs2` and which I modified slightly to look like this:

```
UML# cat /etc/ocfs2/cluster.conf
node:
        ip_port = 7777
        ip_address = 192.168.0.253
        number = 0
        name = node0
        cluster = ocfs2

node:
        ip_port = 7777
        ip_address = 192.168.0.251
        number = 1
        name = node1
        cluster = ocfs2

cluster:
        node_count = 2
        name = ocfs2
```

The one change I made was to alter the IP addresses to what I intend to use for the two UML instances that will form the cluster. You should use IP addresses that work on your network.

The last thing to do before shutting down this instance is to create the mount point where the cluster filesystem will be mounted:

```
UML# mkdir /ocfs2
```

Shut this instance down, and we will boot the cluster, after taking care of one last item on the host—creating the device that the cluster nodes will share:

```
host% dd if=/dev/zero of=ocfs seek=$[ 100 * 1024 ] bs=1K count=1
```

Booting the Cluster

Now we boot two UML instances on COW files with the filesystem we just used as their backing file. So, rather than using `ubda=rootfs` as we had before, we will use `ubda=cow.node0,rootfs` and `ubda=cow.node1,rootfs` for the two instances, respectively. I am also giving them `umid`s of `node0` and `node1` in order to make them easy to reference with `uml_mconsole` later.

The reason for mostly configuring `ocfs2`, shutting the UML instance down, and then starting up the cluster nodes is that the filesystem changes we made, such as installing the `ocfs2` tools and the configuration file, will now be visible in both instances. This saves us from having to do all of the previous work twice.

With the two instances running, we need to give them their separate identities. The `cluster.conf` file specifies the node names as `node0` and `node1`. We now need to change the machine names of the two instances to match these. In Fedora Core 4, which I am using, the names are stored in `/etc/sysconfig/network`. The node part of the value of `HOSTNAME` needs to be changed in one instance to `node0` and in the other to `node1`. The domain name can be left alone.

We need to set the host name by hand since we changed the configuration file too late:

```
UML1# hostname node0
```

and

```
UML2# hostname node1
```

Next, we need to bring up the network for both instances:

```
host% uml_mconsole node0 config eth0=tuntap,,,192.168.0.254
OK
host% uml_mconsole node1 config eth0=tuntap,,,192.168.0.252
OK
```

When configuring `eth0` within the instances, it is important to assign IP addresses as specified in the `cluster.conf` file previously. In my example above, `node0` has IP address `192.168.0.253` and `node1` has address `192.168.0.251`:

```
UML1# ifconfig eth0 192.168.0.253 up
```

and

```
UML2# ifconfig eth0 192.168.0.251 up
```

At this point, we need to set up a filesystem on the shared device, so it's time to plug it in:

```
host% uml_mconsole node0 config ubdbc=ocfs
```

and

```
host% uml_mconsole node1 config ubdbc=ocfs
```

The `c` following the device name is a flag telling the block driver that this device will be used as a clustered device, so it shouldn't lock

the file on the host. You should see this message in the kernel log after plugging the device:

```
Not locking "/home/jdike/linux/2.6/ocfs" on the host
```

Before making a filesystem, it is necessary to bring the cluster up in both nodes:

```
UML# ./vendor/common/o2cb.init online ocfs2
Loading module "configfs": OK
Mounting configfs filesystem at /config: OK
Loading module "ocfs2_nodemanager": OK
Loading module "ocfs2_dlm": OK
Loading module "ocfs2_dlmfs": OCFS2 User DLM kernel interface loaded
OK
Mounting ocfs2_dlmfs filesystem at /dlm: OK
Starting cluster ocfs2: OK
```

Now, on one of the nodes, we run `mkfs`:

```
mkfs.ocfs2 -b 4K -C 32K -N 8 -L ocfs2-test /dev/ubdb
mkfs.ocfs2 1.1.2-ALPHA
Overwriting existing ocfs2 partition.
(1552,0):__dlm_print_nodes:380 Nodes in my domain \
    ("CB7FB73E8145436EB93D33B215BFE919"):
(1552,0):__dlm_print_nodes:384  node 0
Filesystem label=ocfs2-test
Block size=4096 (bits=12)
Cluster size=32768 (bits=15)
Volume size=104857600 (3200 clusters) (25600 blocks)
1 cluster groups (tail covers 3200 clusters, rest cover 3200 clusters)
Journal size=4194304
Initial number of node slots: 8
Creating bitmaps: done
Initializing superblock: done
Writing system files: done
Writing superblock: done
Writing lost+found: done
mkfs.ocfs2 successful
```

This specifies a block size of 4096 bytes, a cluster size of 32768 bytes, a maximum cluster size of eight nodes, and a volume label of `ocfs2-test`.

At this point, we can mount the device in both nodes, and we have a working cluster:

```
UML1# mount /dev/ubdb /ocfs2 -t ocfs2
(1618,0):ocfs2_initialize_osb:1165 max_slots for this device: 8
(1618,0):ocfs2_fill_local_node_info:836 I am node 0
(1618,0):__dlm_print_nodes:380 Nodes in my domain \
    ("B01E29FE0F2F43059F1D0A189779E101"):
```

```
(1618,0):__dlm_print_nodes:384  node 0
(1618,0):ocfs2_find_slot:266 taking node slot 0
JBD: Ignoring recovery information on journal
ocfs2: Mounting device (98,16) on (node 0, slot 0)

UML2# mount /dev/ubdb /ocfs2 -t ocfs2
(1442,0):o2net_set_nn_state:417 connected to node node0 \
    (num 0) at 192.168.0.253:7777
(1522,0):ocfs2_initialize_osb:1165 max_slots for this device: 8
(1522,0):ocfs2_fill_local_node_info:836 I am node 1
(1522,0):__dlm_print_nodes:380 Nodes in my domain \
    ("B01E29FE0F2F43059F1D0A189779E101"):
(1522,0):__dlm_print_nodes:384  node 0
(1522,0):__dlm_print_nodes:384  node 1
(1522,0):ocfs2_find_slot:266 taking node slot 1
JBD: Ignoring recovery information on journal
ocfs2: Mounting device (98,16) on (node 1, slot 1)
```

Now we start to see communication between the two nodes. This is visible in the output from the second mount and in the kernel log of `node0` when `node1` comes online.

To quickly demonstrate that we really do have a cluster, I will copy a file into the filesystem on `node0` and see that it's visible on `node1`:

```
UML1# cd /ocfs2
UML1# cp ~/ocfs2-tools-1.1.2.tar .
UML1# ls -al
total 2022
drwxr-xr-x   3 root root    4096 Oct 14 16:24 .
drwxr-xr-x  28 root root    4096 Oct 14 16:17 ..
drwxr-xr-x   2 root root    4096 Oct 14 16:15 lost+found
-rw-r--r--   1 root root 2058240 Oct 14 16:24 \
    ocfs2-tools-1.1.2.tar
```

On the second node, I'll unpack the `tar` file to see that it's really there.

```
UML2# cd /ocfs2
UML2# ls -al
total 2022
drwxr-xr-x   3 root root    4096 Oct 14 16:15 .
drwxr-xr-x  28 root root    4096 Oct 14 16:18 ..
drwxr-xr-x   2 root root    4096 Oct 14 16:15 lost+found
-rw-r--r--   1 root root 2058240 Oct 14 16:24 \
    ocfs2-tools-1.1.2.tar
UML2# tar xf ocfs2-tools-1.1.2.tar
UML2# ls ocfs2-tools-1.1.2
COPYING         aclocal.m4     fsck.ocfs2     mount.ocfs2    \
rpmarch.guess
CREDITS         config.guess   glib-2.0.m4    mounted.ocfs2  \
runlog.m4
```

```
Config.make.in  config.sub     install-sh     o2cb_ctl       \
sizetest
MAINTAINERS     configure      libo2cb        ocfs2_hb_ctl   \
tunefs.ocfs2
Makefile        configure.in   libo2dlm       ocfs2cdsl      \
vendor
Postamble.make  debian         libocfs2       ocfs2console
Preamble.make   debugfs.ocfs2  listuuid       patches
README          documentation  mkfs.ocfs2     python.m4
README.O2CB     extras         mkinstalldirs  pythondev.m4
```

This is the simplest possible use of a clustered filesystem. At this point, if you were evaluating a cluster as an environment for running an application, you would copy its data into the filesystem, run it on the cluster nodes, and see how it does.

Exercises

For some casual usage here, we could put our users' home directories in the `ocfs2` filesystem and experiment with having the same file accessible from multiple nodes. This would be a somewhat advanced version of NFS home directories.

A more advanced project would be to boot the nodes into an `ocfs2` root filesystem, making them as clustered as they can be, given only one filesystem. We would need to solve a couple of problems.

- The cluster needs to be running before the root filesystem can be mounted. This would require an `initramfs` image containing the necessary modules, initialization script, and tools. A script within this image would need to bring up the network and run the `ocfs2` control script to bring up the cluster.
- The cluster nodes need some private data to give them their separate identities. Part of this is the network configuration and node names. Since the network needs to be operating before the root filesystem can be mounted, some of this information would be in the `initramfs` image.
- The rest of the node-private information would have to be provided in files on a private block device. These files would be bind-mounted from this device over a shared file within the cluster filesystem, like this:

```
UML# mount --bind /private/network /etc/sysconfig/network
```

Without having done this myself, I am no doubt missing some other issues. However, none of this seems insurmountable, and it would make a good project for someone wanting to become familiar with setting up and running a cluster.

Other Clusters

I've demonstrated UML's virtual clustering capabilities using Oracle's `ocfs2`. This isn't the only clustering technology available—I chose it because it nicely demonstrates the use of a host file to replace an expensive piece of hardware, a shared disk. Other Linux cluster filesystems include Lustre from CFS, GFS from Red Hat, and, with a generous definition of clustering, NFS.

Further, filesystems aren't the only form of clustering technology that exists. Clustering technologies have a wide range, from simple failover, high-availability clusters to integrated single-system image clusters, where the entire cluster looks and acts like a single machine.

Most of these run with UML, either because they are architecture-independent and will run on any architecture that Linux supports, or because they are developed using UML and are thus guaranteed to run with UML. Many satisfy both conditions.

If you are looking into using clusters because you have a specific need or are just curious about them, UML is a good way to experiment. It provides a way to bring multiple nodes up without needing multiple physical machines. It also lets you avoid buying exotic hardware that the clustering technology may require, such as the shared storage required by `ocfs2`. UML makes it much more convenient and less expensive to bring in multiple clustering technologies and experiment with them in order to determine which one best meets your needs.

UML as a Decision-Making Tool for Hardware

In this chapter, I demonstrated the use of UML in simulating hardware that is difficult or expensive to acquire in order to make decisions about both software and hardware. By simulating a system with a great deal of devices of a particular sort, it is possible to probe the limits of the software you might run on such a machine. These limits could involve either the kernel or applications. By running the software stack on an

appropriately configured UML instance, you can see whether it is going to have problems before you buy the hardware.

I demonstrated this with a UML instance with a very large number of Ethernet interfaces and some with varying amounts of memory, up to 64GB. The same could have been done with a number of other types of devices, such as CPUs and disks.

With memory, the objective was to analyze the memory requirements of a particular workload without actually having a physical system with the requisite memory in it. You must be careful about doing performance measurements in this case. Looking at wall-clock time is useless because real time will be controlled by the availability of real resources, such as physical memory in the host. A proxy for real time is needed, and when memory is concerned, disk I/O inside the virtual machine is usually a good choice.

The UML instance will act as though it has the memory that was configured on the command line, and the host will swap as necessary in order to maintain that illusion. Therefore, the virtual machine will explicitly swap only when that illusory physical memory is exhausted. A physical machine with that amount of memory will behave in the same way, so a lower amount of disk I/O in the virtual machine will translate into lower real time for the workload on a physical machine.

Finally, I demonstrated the configuration of a cluster of two UML instances. This substituted the use of a host file, rather than a shared disk device, as the cluster interconnect. The ability to substitute a free virtual resource for an expensive physical one is a good reason to prototype a cluster virtually before committing to a physical one. You can see whether your workload will run on a cluster, and if so, how well, with the earlier caveats about making performance measurements.

In a number of ways, a virtual machine is a useful tool for helping you make decisions about software intended to run on physical hardware and about the hardware itself. UML lets you simulate hardware that is expensive or inconvenient to acquire, so you can test-run the applications or workloads you intend to run on that hardware. By doing so, you can make more informed decisions about both the hardware and the software.

CHAPTER 13

The Future of UML

Currently, a UML instance is a standard virtual machine, hard to distinguish from a Linux instance provided by any of the other virtualization technologies available. UML will continue to be a standard virtual machine, with a number of performance improvements. Some of these have been discussed in earlier chapters, so I'm not going to cover them here. Rather, I will talk about how UML is also going to change out of recognition. Being a real port of the Linux kernel that runs as a completely normal process gives UML capabilities not possessed by any other virtualization technology that provides virtual machines that aren't standard Linux processes.

We discussed part of this topic in Chapter 6, when we talked about `humfs` and its ability to store file metadata in a database. The capabilities presented there are more general than we have talked about. `humfs` is based on a UML filesystem called `externfs`, which imports host data into a UML instance as a filesystem. By writing plugin modules for `externfs`, such as `humfs`, anything on the host that even remotely resembles a filesystem can be imported into a UML instance as a mountable filesystem.

Similarly, external resources that don't resemble filesystems but do resemble processes could be imported, in a sense, into a UML instance as a process. The UML process would be a representative of the outside resource, and its activity and statistics would be represented as the activity and statistics of the UML process. Operations performed on the UML process, such as sending it signals or changing its priority, would be reflected back out to the outside in whatever way makes sense.

An extension of this idea is to import the internal state of an application into UML as a filesystem. This involves embedding a UML instance into the application and modifying the application sufficiently to provide access to its data from the captive UML instance through a filesystem. Doing so requires linking UML into the application so that the UML instance and the application share an address space, making it easy for them to share data.

It may not be obvious why this is useful, but it has potential that may turn out to be revolutionary for two major reasons.

1. A small number of applications have some of the attributes of an operating system, and a larger number would benefit from gaining those attributes. As an operating system that is already in user-space, UML is available to provide those attributes very easily in comparison to implementing them from scratch. For example, there are a small number of clusterized applications, such as some databases. As Linux gains clustering capabilities, UML will acquire them, and those capabilities will become available to applications that embed a UML instance. A number of other capabilities exist, such as good SMP scaling, filesystems, and a full network stack.
2. A UML instance embedded in an application with filesystem access to the application's internal data provides a standard development environment. This will make it easy to customize the application's behavior, add features to it, and make it interoperate with other applications that themselves contain embedded UML instances. All the application needs to do is embed the UML instance and provide it with access to whatever data it wishes to expose. At that point, the information is available through the standard Linux file interfaces and can be manipulated using standard Linux tools. Furthermore, applications within the embedded UML instance can use any development tools and environments available for Linux.

Some prominent individual applications would also benefit from embedding UML instances; I'll describe those later.

Another area of future work comes from UML being a virtualized Linux kernel, rather than a userspace Linux kernel. As a virtualized kernel, a UML instance (and all of the subsystems within it) operates as a guest, in the sense that it knows it's a guest and explicitly uses the resources provided by the host. This comes in handy because of the benefits of using pieces of UML, such as the scheduler, as guests on their own.

For example, I have prototyped a guest scheduler patch to the Linux kernel that runs the scheduler as a guest on the normal scheduler. The guest scheduler runs as a process on the host, and processes controlled by it compete for slices of the CPU time that the host scheduler provides to it. Thus, processes controlled by the guest scheduler are jailed with respect to their CPU consumption but unlimited in other respects.

Similarly, other subsystems pulled out of UML will jail processes in different ways. Combining these will allow the system administrator to confine processes and partition the system's resources in arbitrary ways.

THE externfs FILESYSTEM

humfs is a special case of a more general filesystem called externfs. The purpose of externfs is to allow any reasonable external data to be imported as a UML filesystem. externfs doesn't import anything by itself—it simply makes it easy to import external data by implementing an interface, defined by externfs, to the Linux filesystem layer. externfs provides the glue between that interface and the Linux kernel VFS interface, allowing the data to appear to be a Linux filesystem.

This will allow you to mount this data as a UML filesystem and use standard utilities and scripts to examine and manipulate it. The filesystem interface hides the specialized interface normally used to access the data. By providing a common way to access the information, data sources that are normally disjointed and isolated from each other can be made to interoperate. Data can be copied from one database to a completely different database merely by copying files.

The sqlfs example in Chapter 6 as a possible humfs metadata format demonstrates this by allowing you to examine and change a

database using normal Linux utilities rather than a SQL monitor. Of course, the SQL interface is still there, but it has been hidden under the Linux filesystem interface by the UML filesystem that imported it.

Essentially any structured data anywhere can be represented somehow as files and directories, and a plugin for `externfs` that maps the structure onto files and directories will import that data as a UML filesystem.

This is a large universe of possibilities, but which of them will actually prove to be useful? Representing data this way would be useful for any database whose contents are not readily accessible as text. Having the database available as a set of directories and files allows you to use standard utilities such as `find` and `grep` on it. It would not be so useful for any database that already uses text, such as any of the ones in `/etc` (e.g., the password and group files). These can already be easily analyzed and searched with the standard text utilities.

A package database might be a good candidate for this sort of treatment. `rpm` and `dpkg` have their own syntaxes for querying their databases. However, having the host's package database, including installed and available packages and the information associated with them, as a set of text files would make it unnecessary to use those syntaxes. Instead, you would use `ls`, `cat`, and `find` to tell you what you need to know.

For example, in order to figure out which package owns a particular file, such as `/etc/passwd`, you would do something like this:

```
UML% find /host-packages -name passwd
/host-packages/installed/setup-2.5.46-1/files/etc/passwd
```

The output tells you that `/etc/passwd` is a part of the `setup-2.5.46-1` package. Similarly, you could find the package's description like this:

```
UML% cat /host-packages/installed/setup-2.5.46-1/description
The setup package contains a set of important system
configuration and setup files, such as passwd, group, and
profile.
```

There's no reason that the package database filesystem would be limited to importing the host's package database. The package databases of other hosts on the network could also be imported into the UML using a network-aware version of this filesystem. Mounting another host's package database would involve communicating with a daemon on the remote side. So, via this daemon, you could have a set of filesystems

such as /packages/my-host, /packages/bob-host, /packages/jane-host, and /packages/web-server.

Having the package information for all the hosts on the network in one place would turn the UML into a sort of control center for the network in this regard. Then you could perform some useful operations.

- Compare the configurations of different machines:

```
UML% ls -1 /packages/my-host/installed > /tmp/x
UML% ls -1 /packages/bob-host/installed > /tmp/y
UML% diff /tmp/x /tmp/y
```

- Ensure that all machines on the network have the same versions of their packages installed by comparing the version files of the package subdirectories in the host package filesystems.
- Install and delete packages:

```
UML% rm -rf /packages/my-host/installed/bc-1.06-18
UML% mv firefox-1.0.4-5.i386.rpm /packages/my-host/installed
```

  These two operations would translate into a package removal and a package installation on the host. In the installation example, the firefox RPM file would be copied out to the host and installed. Then a firefox subdirectory would appear in the /packages/my-host/installed directory.

If you wanted to enforce a policy that all configuration changes to any machine on the network would have to be done from this UML control console, the daemon on each host would maintain a lock on the package database. This would prevent any changes from happening locally. Since these daemons would be controlled from the UML instance, configuration changes to any of the hosts could be done only from the UML instance through this filesystem.

If a number of machines needed to have the same configurations, you could also have them all mounted in the same place in the UML control console. Operations within this filesystem would be multiplexed to all of the hosts. So, installing a new package through this filesystem would result in the package being copied to all of the hosts and installed on all of them. Similarly, removing a package would result in it being removed from all the hosts.

You can consider using a UML as a similar control console for any other system administration database. Using it to manage the host's

password or group files is probably not practical, as I mentioned earlier. However, it may be useful to manage the password or group files for a network, if you're not using an existing distributed mechanism, such as NIS, for them.

You could take this control console idea further and use an `externfs` plugin to front a number of databases on the network, not just one. For example, consider a large organization with several levels of management and an `externfs`-based filesystem that allows a mirror of this organization to be built in it. So, every manager would be represented by a directory that contains a directory for each person who reports directly to that manager. If some of these reporting people were also managers, there would be another level of directories further down. Hiring a new person would involve creating a directory underneath the hiring manager. The filesystem would see this directory creation and perform the necessary system administration tasks, such as:

- Creating login and mail accounts
- Adding the new person to the appropriate groups and mailing lists
- Updating online organization charts and performing other organization-specific tasks

Similarly, removing a person's directory would result in the reversal of all of these tasks.

Performing these tasks would not need to be done by hand, nor would it require a specialized application to manage the whole process. It would be done by changing files and directories in this special filesystem and tying those changes to the necessary actions on the network. I'm not suggesting that someone would be literally running the `mkdir` and `rmdir` utilities in a shell whenever someone is hired or leaves, although that would work. There would likely be a graphical interface for doing this, and it would likely be customized for this task, to simplify the input of the required information. However, putting it in a filesystem makes this information available in a standardized way at a low enough level that any sort of application, from a shell script to a customized graphical interface, can be written to manipulate it.

If the filesystem contains sensitive data, such as pay rates or home addresses, Linux file permissions can help prevent unauthorized people from seeing that information. Each piece of data about an employee could potentially be in its own file, with user and group ownership and permissions that restrict access to people who are allowed to view the information.

This example seems to fit a filesystem particularly well. No doubt there are others. UML's externfs allows this sort of information to be plugged into a UML as a filesystem, where it can be viewed and manipulated by any tools that know how to deal with files and directories.

This scenario is not as far out in left field as it may appear. Practically every Linux system in the world is doing something similar by providing a unified interface to a number of disparate databases. A typical Linux system contains the following:

- At least one, and often more, disk-based filesystems such as ext2, ext3, reiserfs, or xfs
- A number of virtual, kernel-based filesystems such as procfs, sysfs, and devpts
- Usually at least one CD or DVD filesystem
- Often some devices such as MP3 players or cameras that represent themselves as storage devices with FAT or HFS filesystems

You can think of all of these as being different kinds of databases to which the Linux VFS layer is providing a uniform interface. This lets you transparently move data between these different databases (as with ls -l /proc > /tmp/processes copying data from the kernel to /tmp) and transparently search them. You don't need to be concerned about the underlying representation of the data, which differs greatly from filesystem to filesystem.

What I described above is close to the same thing, except that my example uses the Linux VFS interface to provide the same sort of access to a different class of databases: personnel databases, corporate phone books, and so on. In principle, these are no different from the on-disk databases your files are stored in. I'd like to see access to these be as transparent and unified as access to your disks, devices, and internal kernel information is now.

externfs provides the framework for making this access possible. Each different kind of database that needs to be imported into a UML instance would need an externfs plugin that knows how to access it. With that written, the database can be imported as a Linux filesystem. At that point, the files and directories can be rearranged as necessary with Linux bind mounts. In the example above, the overall directory hierarchy can be imported from the corporate personnel database. Information like phone numbers and office locations may be in another database. Those files can be bind-mounted into the employee hierarchy, so that when you look at the directory for an employee, all of

that person's information is present there, even though it's coming from a number of different databases.

The infrastructure to provide a transparent, unified interface to these different databases already exists. The one thing lacking is the modules needed to turn them into filesystems.

VIRTUAL PROCESSES

Some things don't map well to files, directories, or filesystems, but you may wish to import them into Linux at a low level in order to manipulate them in a similar way. Many of these may resemble processes.

- They start at a certain time and may stop at some point.
- They consume various sorts of resources while running.
- They may be in various states at different times, such as actively running, stopped, or waiting for an event.

It may make sense to represent such things as Linux processes, and it will be possible to create UML processes that represent the state of something external to the UML instance. This could be something very close to a process, such as a server on the host, or it could be something very unlike a process, such as a project.

This "virtual" process would appear in the UML instance's process list with all the attributes of a normal process, except that these would be fabricated from whatever it is representing. As with the filesystem example, actions performed on one of these processes would be reflected out to the real thing it represents. So, sending a signal to a virtual process that represents a service on some machine elsewhere on the network could shut down that service. Changing the virtual process's priority would have the analogous effect on the processes that belong to that service.

Representing a project as a "virtual" process is not as good a fit. It is hard to imagine that a high-level manager would sit in front of a process listing, look at processes representing projects within the company, and change their priorities or cancel them by clicking on a Linux process manager. Some things resemble processes, but their attributes don't map well onto Linux processes.

Representing network services as UML processes and managing them as such doesn't seem to me far fetched. Neither does representing

hosts as a whole. Machines can respond to signals on their process representatives within the UML instance by shutting down or rebooting, and the status of a machine seems to map fairly well onto the status of a process.

Processes are more limited in this regard than filesystems are since they can't contain arbitrary data, such as names and file contents, and they have a limited number of attributes with fairly inflexible semantics. So, while I can imagine a synthetic filesystem being used to manage personnel in some sense, I don't think synthetic processes can be used in a similar way. Nevertheless, within those limits, I think there is potential for managing some process-like entities as synthetic UML processes and using that capability of UML to build a control console for those entities.

CAPTIVE UML

So far I've talked about using special filesystems to import the external state of outside entities into a UML instance where it can be manipulated through a filesystem. An extension of this is to import the internal state of an application into a UML instance to be manipulated in the same way.

This would be done by actually embedding the UML instance within the application. The application would link UML in as a library, and a UML instance would be booted when the application runs. The application would export to the UML instance whatever internal state it considers appropriate as a filesystem. Processes or users within that UML instance could then examine and manipulate that state through this filesystem, with side effects inside the application whenever anything is changed.

Secure `mod_perl`

Probably the best example of a real-world use for a captive UML that I know of is Apache's `mod_perl`. This loadable module for Apache contains a Perl interpreter and allows the use of Perl scripts running inside Apache to handle requests, generate HTML, and generally control the server. It is very powerful and flexible, but it can't be used securely in a shared Apache hosting environment, where a hosting

company uses a single Apache server to serve the Web sites of a number of unrelated customers.

Since a Perl script runs in the context of the Apache server and can control it, one customer using `mod_perl` could take over the entire server, cause it to crash or exit, or misbehave in any number of other ways. The only way to generate HTML dynamically with a shared Apache is to use CGI, which is much slower than with `mod_perl`. CGI creates a new process for every HTML request, which can be a real performance drag on a busy server. This is especially the case when the Web site is generated with Perl, or something similar, because of the overhead of the Perl interpreter.

With some captive UML instances inside the Apache server, you could get most of the performance of standard `mod_perl`, plus a lot of its flexibility, and do so securely, so that no customer could interfere with other sites hosted on the same server or with the server itself. You would do this by having the customer's Perl scripts running inside the instances, isolating them from anything outside. Communication with the Apache server would occur through a special filesystem that would provide access to some of Apache's internal state.

The most important piece of state is the stream of requests flowing to a Web site. These would be available in this filesystem, and in a very stripped-down implementation, they would be the only thing available. So, with the special Apache filesystem mounted on `/apache`, there could be a file called `/apache/request` that the Perl script would read. Whenever a request arrived, it would appear as the contents of this file. The response would be generated and written back to that file, and the Apache server would forward it to the remote browser.

One advantage of this approach is immediately evident. Since the HTML generation is happening inside a full Linux host and communication with the host Apache server is through a set of files, the script can be written in any language—Perl, Python, Ruby, shell, or even compiled C, if maximum performance is desired. It could even be written in a language that didn't exist at the time this version of Apache was released. The new language environment would simply need to be installed in the captive UML instance.

Another advantage is that the Web site can be monitored in real time, in any manner desired, from inside the UML instance. This includes running an interactive debugger on the script that's generating the Web site, in order to trace problems that might occur only in a production deployment. Obviously, this should be done with caution,

considering that debuggers generally slow down whatever they're debugging and can freeze everything while stopped at a breakpoint. However, for tracking down tricky problems, this is a capability that doesn't exist in `mod_perl` currently but comes for free with a captive UML instance.

So far, I've talked about using a single file, `/apache/request`, to receive HTTP requests and to return responses. This Apache filesystem can be much richer and can provide access to anything in the `mod_perl` API, which is safe within a shared server. For example, the API provides access to information about the connection over which a request came, such as what IP the remote host has and whether the connection supports keepalives. This information could be provided through other files in this filesystem.

The API also provides access to the Apache configuration tree, which is the in-memory representation of the `httpd.conf` file. Since this information is already a tree, it can be naturally represented as a directory hierarchy. Obviously, full access to this tree should not be provided to a customer in a shared server. However, the portions of the tree associated with a particular customer could be. This would allow customers to change the configuration of their own Web sites without affecting anyone else or the server as a whole.

For example, the owner of a `VirtualHost` could change its configuration or add new `VirtualHosts` for the same Web site. Not only would this be more convenient than asking the hosting company to change the configuration file, it also could be done on the fly. This would allow the site to be reconfigured as much and as often as desired without having to involve the hosting company.

It is common to have Apache running inside a UML instance. This scheme turns that inside-out, putting the UML instance inside Apache. Why do things this way instead of the standard Apache-inside-UML way? The reasons mirror the reasons that people use a shared Apache provider rather than colocating a physical machine and running a private Apache on it.

It's cheaper since it involves less hardware, and it doesn't require a separate IP address for every Web site. The captive UML instance has less running in it compared to running Apache inside UML. All Web sites on the server share the same Apache instance, and the only resources they don't share are those dedicated to generating the individual Web sites. Also, it's easier to administrate. The hosting company manages the Apache server and the server as a whole, and the customers are responsible only for their own Web sites.

Evolution

Putting a UML instance inside Apache is probably the most practical use of a captive UML instance, but my favorite example is Evolution. I use Evolution on a daily basis, and there are useful things that I could make it do if there were a UML instance inside it with access to its innards. For example, I have wanted an easy way to turn an e-mail message into a task by forwarding the e-mail to some special address. With a UML instance embedded inside Evolution, I would have the instance on the network with a mail server accepting e-mail. Then a `procmail` script, or something similar, would create the task via the filesystem through which the UML instance had access to Evolution's data.

So, given an e-mail whose title is "`frobnitz` is broken" and whose message is "The `frobnitz` utility crashes whenever I run it," the script would do something like this:

```
UML% cat > /evolution/tasks/"frobnitz is broken" << EOF
The frobnitz utility crashes whenever I run it
EOF
```

This would actually create this task inside Evolution, and it would immediately appear in the GUI. Here, I am imagining that the "Evolution filesystem" would be mounted on `/evolution` and would contain subdirectories such as `tasks`, `calendar`, and `contacts` that would let you examine and manipulate your tasks, appointments, and contacts, respectively. Within `/evolution/tasks` would be files whose names were the same as those assigned to the tasks through the Evolution GUI. Given this, it's not too much of a stretch to think that creating a new file in this directory would create a new task within Evolution, and the contents of the task would be the text added to the file.

In reality, an Evolution task is a good deal more complicated and contains more than a name and some text, so tasks would likely be represented by directories containing files for their attributes, rather than being simple files.

This example demonstrates that, with a relatively small interface to Evolution and the ability to run scripts that use that interface, you can easily make useful customizations. This example, using the tools found on any reasonable Linux system, would require a few lines of `procmail` script to provide Evolution with a fundamental new capability—to receive e-mail and convert it into a new task.

The new script would also make Evolution network-aware in a sense that it wasn't before by having a virtual machine embedded within it that is a full network node.

I can imagine making it network-aware in other ways as well:

- By having a bug-tracking system send it bug reports when they are assigned to you so they show up automatically in your task list, and by having it send a message back to the bug-tracking system to close a bug when you finish the task
- By allowing a task to be forwarded from one person to another with one embedded UML sending it to another, which recreates the task by creating the appropriate entries in the virtual Evolution filesystem

The fact that the captive UML instance could be a fully functional network node means that the containing application could be, too. The data exported through the filesystem interface could then be exported to the outside world in any way desired. Similarly, any data on the outside could be imported to the application through the filesystem interface. The application could export a Web interface, send and receive e-mail, and communicate with any other application through its captive UML instance.

Any application whose data needs to be moved to or from other applications could benefit from the same treatment. Our bug-tracking system could forward bugs to another bug tracker, receive bug reports as e-mail, or send statistics to an external database, even when the bug tracker couldn't do any of these itself. If it can export its data to the captive UML instance, scripts inside the instance can do all of these.

Given sufficient information exported to the captive UML instance, any application can be made to communicate with any other application. An organization could configure its applications to communicate with each other in suitable ways, without being constrained by the communication mechanisms built into the applications.

Application Administration

Some applications, such as databases and those that contain databases, require dedicated administration, and sometimes dedicated administrators. These applications try to be operating systems, in the sense that they duplicate and reimplement functionality that is

already present in Linux. A captive UML within the application could provide these functions for free, allowing it to either throw out the duplicated functionality or avoid implementing it in the first place.

For example, databases and many Web sites require that users log in. They have different ways to store and manage account information. Almost everyone who uses Linux is familiar with adding users and changing passwords, but doing the same within a database requires learning some new techniques. However, with a captive UML instance handling this, the familiar commands and procedures suffice. The administrator can log in to the UML instance and add or modify accounts in the usual Linux way.

The captive UML instance can handle authentication and authorization. When a user logs in to such a Web site, the site passes the user ID and password to the UML instance to be checked against the password database.

If there are different levels of access, authorization is needed as well. After the captive UML instance validates the login, it can start a process owned by that user. This process can generate the HTML for requests from that user. With the site's data within this UML instance and suitably protected, authorization is provided automatically by the Linux file permission system. If a request is made for data that's inaccessible to the user, this process will fail to access it because it doesn't have suitable permissions.

The same is true with other tasks such as making backups. Databases have their own procedures for doing this, which differ greatly from the way it's done on Linux. With a captive UML instance having access to the application's data, the virtual filesystem that the instance sees can be backed up in the same way as any other Linux machines. The flip side of this is restoring a backup, which would also be done in the usual Linux way.

The added convenience of not having to learn new ways to perform old tasks is obvious. Moreover, there are security advantages. Doing familiar tasks in a familiar way reduces the likelihood of mistakes, for example, making it less likely that adding an account, and doing it wrong, will inadvertently open a security hole.

There is another security benefit, namely, that the application administrator logs in to the application's captive UML instance to perform administration tasks. This means that the administrator doesn't need a special account on the host, so there are fewer accounts, and thus fewer targets, on the host. When the administrator doesn't need

root privileges on the host, there is one fewer person with root access, one fewer person who can accidentally do something disastrous to the host, and one fewer account that can be used to as a springboard to root privileges.

A Standard Application Programming Interface

Another side of a captive UML instance can be inferred from the discussion above, but I think it's worth talking about it specifically. A Linux environment, whether physical or virtual, normally comes with a large variety of programming tools. Add to this the ability of a captive UML instance to examine and manipulate the internal state of its application, and you have a standard programming environment that can be imported into any application.

The current state of application programmability and extensibility is that the application provides an API to its internals, and that API can be used by one of a small number of programming languages. To extend Emacs, you have to use Lisp. For GIMP, you have Scheme, TCL, and Perl. For Apache, there is Perl and Python. With a reasonable Linux environment, you get all of these and more. With an API based on the virtual filesystem I have described, application development and extension can be done with any set of tools that can manipulate files.

With an embedded UML instance providing the application's development environment, the developers don't need to spend time creating an API for every language they wish to support. They spend the time needed to embed a UML instance and export internal state through a UML virtual filesystem, and they are done. Their users get to choose what languages and tools they will use to write extensions.

Application-Level Clustering

A captive UML can also be used to provide application access to kernel functionality. Clustering is my favorite example. In Chapter 12 we saw two UML instances being turned into a little cluster, which is a simple example of process-level clustering. There is at least one real-world, commercial example of this—Oracle clusters, where the database instances on multiple systems cooperate to run a single database.

There would be more examples like this if clustering were easier to do. Oracle did its own clustering from scratch, and any other product,

commercial or open source, would have to do the same. With the clustering technologies that are currently in Linux and those that are on their way, UML can provide a much easier way to "clusterize" an application.

With UML, any clustering technology in the Linux kernel is automatically running in a process, assuming that it is not hardware-dependent. To clusterize an application, we need to integrate UML into the application in such a way that it can use that technology.

Integrating UML into the application is a matter of making UML available as a linkable library. At that point, the application can call into the UML library to get access to any functionality within it.

I am envisioning this as an enabling technology for much deeper Internet-wide collaborations than we've seen so far. At this point, most such collaborations have been Web-based. Why isn't that sufficient? Why do we need some new technology? The answer is the same as that for the question of why you don't do all of your work within a Web browser. You create a lot—likely all—of your work with other applications because these other tools are specialized for the work you are doing, and your Web browser isn't. Your tools have interfaces that make it easy to do your work, and they understand your work in ways that enable them to help. Web browsers don't. Even when it is possible to do the same work in your Web browser, the Web interface is invariably slower, harder to use, and less functional than that for the specialized application.

Imagine taking one of these applications and making it possible for many people to work within it at the same time, working on the same data without conflicting with each other. Clusterizing the application would allow this.

To make our example a bit more concrete, let's take the `ocfs2` UML cluster we saw in Chapter 12 and assume that an application wants to use it as the basis for making a cluster from multiple instances of itself. The `ocfs2` cluster makes a shared disk accessible to multiple nodes in such a way that all the nodes see the same data at all times. The application shares some of its data between instances by storing it in an `ocfs2` volume.

Let us say that this application is a document editor, and the value it gains from being clusterized is that many people can work on the same document at the same time without overwriting each other's work. In this case, the document is stored in the cluster filesystem, which is stored in a file on the host.

When an instance of this editor starts, the captive UML inside it boots enough that the kernel is initialized. It attaches itself to the

`ocfs2` shared disk and brings itself up as a cluster node. The editor knows how the document is stored within the shared disk and accesses it by directly calling into the Linux filesystem code rather than making system calls, such as open and read, as a normal process would.

With multiple instances of the editor attached to the same document, and the captive UML instances as nodes within the cluster, a user can make changes to the document at the same time as other users, without conflicting with them.

The data stored within the cluster filesystem needs to be the primary copy of the document, in the sense that changes are reflected more or less immediately in the filesystem. Otherwise, two users could change the same part of the document, and one would end up overwriting the other when the changes made it to the filesystem.

How quickly changes need to be reflected in the filesystem is affected to some extent by the organization of the document and the properties of the cluster being used. A cluster ensures that two nodes can't change the same data at the same time by locking the data so that only one node has access to it at any given time. If the locking is done on a per-file basis, and this editor stores its document in a single file, then the first user will have exclusive access to the entire document for the entire session. This is obviously not the desired effect.

Alternatively, the document could be broken into pieces, such as a directory hierarchy that reflects the organization of the document. The top-level directories could be volumes, with chapter subdirectories below that, sections below the chapters, and so on. The actual contents would reside within files at the lowest level. These would likely be at the level of paragraphs. A cluster that locks at the file level would let different people work on different paragraphs without conflict.

There are other advantages to doing this. It allows the Linux file permission system to be applied to the document with any desired granularity. When each contributor to the document is assigned a section to work on, this section would be contained inside some directory. The ownerships on these directories and files would be such that those people assigned to the section can edit it, and others can't, although they may have permission to read it. Groups can be set up so that some people, such as editors, can modify larger pieces of the document.

At first glance, it would appear that this could be implemented by running the application within a cluster, rather than having the cluster inside the application, as I am describing. However, for a number of reasons, that wouldn't work.

The mechanics of setting up the cluster require that it be inside the application. Consider the case where this idea is being used to support an Internet-wide collaboration. Running the application within a cluster requires the collaboration to have a cluster, and everyone contributing to it must boot their systems into this cluster. This immediately runs into a number of problems.

First, many people who would be involved in such an effort have no control over the systems they would be working from. They would have to persuade their system administrators to join this cluster. For many, such as those in a corporate environment, with systems installed with defined images, this would be impossible. However, when the application comes with its own clustering, this is much less of a problem. Installing a new application is much less problematic than having the system join a cluster.

Even if you can get your system to join this cluster, you need your system either to be a permanent member or to join when you run the application that needs it. These requirements pose logistical and security problems. To be a cluster node means sharing data with the other nodes, so having to do this whenever the system is booted is undesirable. To join the cluster only when the application is running requires the application to have root privileges or to be able to call on something with those privileges. This is also impossible for some types of clustering, which require that nodes boot into the cluster. Both of these options are risky from a security perspective. With the cluster inside the application, these problems disappear. The application boots into the cluster when it is started, and this requires no special privileges.

Second, there may be multiple clustered applications running on a given system. Having the system join a different cluster for each one may be impossible, as this would require that the system be a member of multiple clusters at the same time. For a cluster involving only a shared filesystem, this may be possible. But it also may not. If the different clusters require different versions of the same cluster, they may be incompatible with each other. There may be stupid problems like symbol conflicts with the two versions active on the host at the same time. For any more intrusive clustering, being a member of multiple clusters at once just won't work. The extreme case is a Single-System Image (SSI) cluster where the cluster acts as a single machine. It is absolutely impossible to boot into multiple instances of these clusters at once. However, with the cluster inside the application, this is not an issue. There can't be conflicts between different versions of the same

clustering software between different clusters, or between different types of clusters, because each cluster is in its own application. They are completely separate from each other and can't conflict.

Consider the case where the large-scale collaboration decides to upgrade the cluster software it is using or decides to change the cluster software entirely. This change would require the administrators of all the involved systems to upgrade or change them. This logistical nightmare would knock most of the collaboration offline immediately and leave large parts of it offline for a substantial time. The effects of attempting this could even kill the collaboration. An upgrade would create two isolated groups, and the nonupgrading group could decide to stay that way, forking the collaboration. With the cluster as part of the application, rather than the other way around, an upgrade or change of cluster technologies would involve an upgrade of the application. This could also fail to go smoothly, but it is obviously less risky than upgrading the system as a whole.

Security also requires that the cluster be within the application. Any decent-size collaboration needs accountability for contributions and thus requires members to log in. This requires a unified user ID space across the entire cluster. For any cluster that spans organization boundaries, this is clearly impossible. No system administrator is going to give accounts to a number of outsiders for the benefit of a single application. It may also be mathematically impossible to assign user IDs such that they are the same across all of the systems in the cluster. With the application being its own cluster, this is obviously not a problem. With the captive UML instances being members of the cluster, they have their own separate, initially empty, user ID space. Assigning user IDs in this case is simple.

Now, consider the case where the application requires an SSI cluster. For it to require the system to be part of the cluster is impossible for logistical reasons, as I pointed out above. It's also impossible from a security standpoint. Every resource of every member of the cluster would be accessible to every other member. This is unthinkable for any but the smallest collaborations. This is not a problem if the cluster is inside the application. The application boots into the cluster, and all of its resources are available to the cluster. Since the application is devoted to contributing to this collaboration, it's expected that all of its information and resources are available to the other cluster nodes.

Earlier, I used the example of a UML cluster based on `ocfs2` to show that process-level clustering using UML is possible and is the

most practical way to clusterize an application. To implement the large-scale collaborations I have described, `ocfs2` is inadequate for the underlying cluster technology for a number of reasons.

- ☞ It requires a single disk shared among all of its nodes. For a UML cluster, this means a single file that's available to all nodes. This is impractical for any collaboration that extends much beyond a single host. It could work for a local network, sharing the file with something like NFS, but won't work beyond that. What is needed for a larger collaboration is a cluster technology that enables each node to have its own local storage, which it would share with the rest of the cluster as needed.
- ☞ `ocfs2` clusters are static. The nodes and their IP addresses are defined in a cluster-wide configuration file. The shared filesystem has a maximum cluster size built into it. This can't work for a project that has contributors constantly coming and going. What is required is something that allows nodes to be added and removed dynamically and that does not impose a maximum size on the cluster.
- ☞ `ocfs2` doesn't scale anywhere near enough to underlie a large collaboration. I am envisioning something with the scale of Wikipedia, with hundreds or thousands of contributors, requiring the clustering to scale to that number of nodes. `ocfs2` is used for sharing a database, which is typically done with a number of systems in the two-digit range or less.

While `ocfs2` doesn't have the ability to power such a project, I know of one clustering technology, GFS, that might. It stores data throughout the cluster. It claims to scale to tens of thousands of clients, a level that would support a Wikipedia-scale collaboration. It does seem to require good bandwidth (gigabit Ethernet or better) between nodes, which the Internet as a whole can't yet provide. Whether this is a problem probably depends on the quantity of data that needs to be exchanged between nodes, and that depends on the characteristics of the collaboration.

These projects probably will not be well served by existing technologies, at least at first. They will start with something that works well enough to get started and put pressure on the technology to develop in ways that serve them better. We will likely end up with clusters with different properties than we are familiar with now.

VIRTUALIZED SUBSYSTEMS

I plan to take advantage of UML's virtualization in one more way: to use it to provide customizable levels of confinement for processes. For example, you may wish to control the CPU consumption of a set of processes without affecting their access to other machine resources. Or you may wish to make a container for some processes that restricts their memory usage and filesystem access but lets them consume as much CPU time as they like.

I'm going to use UML to implement such a flexible container system by breaking it into separate subsystems (e.g., the scheduler and the virtual memory system). When you break UML, which is a virtual kernel, into separate pieces, those pieces are virtual subsystems. They can run on or within their nonvirtual counterparts, and they require a nonvirtual counterpart to host them in order to function at all.

For example, my virtualized scheduler runs a guest scheduler inside a host process. The host process provides the context needed for the guest scheduler to function. The guest scheduler requires some CPU cycles to allocate among the processes in its care. These cycles come from its host process, which is competing with other processes for CPU cycles from the host scheduler.

Similarly, a guest virtual memory system would allocate memory to its processes from a pool of memory provided by the host virtual memory system.

You would construct a container by building a set of virtualized subsystems, such as the scheduler and virtual memory system, loading it into the host kernel, and then loading processes into it. Those processes, in the scheduler and virtual memory system case, would get their CPU cycles from the guest scheduler and their memory from the guest virtual memory system. In turn, these would have some allocation of cycles and memory from the host.

Let's take the guest scheduler as an example since it has been implemented. A new guest scheduler is created by a process opening a special file in `/proc`:

```
host% cat /proc/schedulers/guest_o1_scheduler &
Created sched_group 290 ('guest_o1_scheduler')
```

The contents of `/proc/schedulers` are the schedulers available for use. In this case, there is only one, `guest_o1_scheduler`. This creates a `sched_group`, which is the set of processes controlled by this

scheduler. When the system boots, all processes are in `sched_group 0`, which is the host scheduler. `sched_group 290` is the group controlled by the `cat` process we ran, which had process ID 290.

Once we have a guest scheduler, the next step is to give it some processes to control. This is done by literally moving processes from `sched_group 0` to `sched_group 290`. Let's create three infinite shell loops and move two of them into the new scheduler:

```
host% bash -c 'while true; do true; done' &
[2] 292
host% bash -c 'while true; do true; done' &
[3] 293
host% bash -c 'while true; do true; done' &
[4] 294
host% mv /proc/sched-groups/0/293 /proc/sched-groups/290/
host% mv /proc/sched-groups/0/294 /proc/sched-groups/290/
```

Now process 290, which is the host representative of the guest scheduler, is competing with the other host processes, including the busy loop with process ID 292, for CPU time. Since those are the only two active processes on the host scheduler, they will each get half of the CPU. The guest scheduler, inside process 290, is going to take its half of the CPU and split it between the two processes under its control. Thus, processes 293 and 294 will each get half of that, or a quarter of the CPU each:

```
host% ps uax
...
root       292 49.1  0.7  2324  996 tty0     R    21:51  \
    14:40 bash -c
root       293 24.7  0.7  2324  996 tty0     R    21:51  \
     7:23 bash -c
root       294 24.7  0.7  2324  996 tty0     R    21:51  \
     7:23 bash -c
...
```

The guest scheduler forms a CPU compartment—it gets a fixed amount of CPU time from the host and divides it among its processes. If it has many processes, it gets no more CPU time than if it had only a few. This is useful for enforcing equal access to the CPU for different users or workloads, regardless of how many processes they have running.

By loading each group of processes, whether a user, an application, a workload, or an arbitrary set of processes, into one of these compartments, you guarantee that the groups as a whole get treated equally by the scheduler.

I've described the guest schedulers as being loaded into the host kernel, but I can also see a role for userspace guest schedulers. Most obviously, by making the scheduler of a UML instance visible to the host as a guest scheduler, its processes become visible to the host in the same way as the host's own processes. Their names and resource usage become visible on the host. They also become controllable in the same way—a signal can be sent to one from the host and the corresponding UML process will receive it.

Making a UML scheduler visible as a host guest scheduler requires an interface for a process to register itself as a guest scheduler. This interface would be the mechanism for telling the host about the guest's processes and their data. Once we have an interface like this, there's no reason that UML has to be the only user of it.

A number of other applications have internal process-like components and could use this interface. Anything with internal threads could make them visible on the host in the same way that UML processes would be. They would be controllable in the same way, and the attributes the host sees would be provided by the application.

A Web server could make requests or sessions visible as host processes. Similarly, a mail server could make incoming or outgoing e-mail messages look like processes. The ability to monitor and control these servers with this level of granularity would make them much more manageable.

The same ideas would apply to any other sort of compartment. A memory compartment would be assigned some amount of memory when created, just as a CPU compartment has a call on a certain amount of CPU time. Processes would be loaded into it and would then have their memory allocations satisfied from within that pool of memory.

If a compartment runs out of memory, it has to start swapping. It is required to operate on the fixed amount of memory it was provided and can't allocate more from the host when it runs short. It has to swap even if there is plenty of memory free on the rest of the system. In this way, the memory demands of the different groups of processes on the host are isolated from each other. One group can't adversely affect the performance of another by allocating all of the memory on the system.

Compartmentalization is an old subject, and there are many ways to do it, including some that are currently being implemented on Linux, principally, CKRM, or Class-based linux Kernel Resource Management. These projects add resource allocation and control infrastructures to the kernel and add interfaces that allow users to control the resulting compartments.

These approaches necessarily involve modifying the basic algorithms in the kernel, such as the scheduler and the virtual memory system. This adds some overhead to these algorithms even when compartments aren't being used, which is likely to be the common case. There is a bias in the Linux kernel development community against making common things more expensive in order to make uncommon things cheap. Compartmentalization performed in these ways conflicts with that ethos.

More importantly, these algorithms have been carefully tuned under a wide range of workloads. Any perturbations to them could throw off this tuning and require repeating all this work.

In contrast, the approach of gaining compartmentalization through virtualization requires no changes to these core algorithms. Instead of modifying an algorithm to accommodate compartments, a new copy of the same algorithm is used to implement them. Thus, there is no performance impact and no behavior perturbation when compartments are not being used.

The price of having two complete implementations of an algorithm instead of a single modified one is that compartments will tend to be more expensive. This is the trade-off against not affecting the performance and behavior when compartments are not being used.

CONCLUSION

UML differs from other virtualization technologies in implementing virtualization completely in userspace. This gives it capabilities that have not been realized yet, but I believe UML will ultimately be more widely used for purposes other than just as a virtual machine.

The fact that UML implements a virtual machine in a set of processes means that it can be repackaged as a library and linked into other applications, which gain an embedded virtual machine. This gives them a standard development and extension environment that is familiar to everyone who does Linux development. This may make those applications more useful than they would be otherwise. They gain the ability to communicate with each other in arbitrary ways, allowing them to adapt to the workflow rather than forcing the workflow to adapt to them.

For some specific applications, this may open up new markets. I described how shared Apache configurations could benefit from this.

They would gain the ability to securely host multiple dynamic Web sites using `mod_perl`, which currently requires a dedicated system for each site. This has obvious economic advantages, as a single system could replace the many systems currently hosting these sites. Other advantages flow from this approach, such as being freed from having to use a specific language for development and being able to interactively debug a Web site inside the live server.

The use of UML for compartmentalization demonstrates another aspect of userspace virtualization. While I demonstrated the guest scheduler being loaded into the kernel, it is not necessarily required to be there. It should be possible to have a guest scheduler running in a process, in userspace, doing all the things that the in-kernel guest scheduler does. The fact that the scheduler and the other subsystems can be virtualized at all is a result of the fact that they started from UML, in userspace. Since UML is already a virtualized Linux kernel, any pieces of it will be similarly virtualized.

APPENDIX A

UML Command-Line Options

There are a number of UML-specific command-line options. The largest group configures the hardware and devices that the virtual machine will have. The rest are used to specify how the instance will be managed from the host, to set debugging options, or to print information about the UML instance.

DEVICE AND HARDWARE SPECIFICATIONS

The following options set configurations for virtual devices and hardware.

- `dsp=`*`dsp device`* and `mixer=`*`mixer device`*—These two options specify the host audio interfaces for the use of the UML audio pass-through driver. The default values are `/dev/sound/dsp` and `/dev/sound/mixer`, respectively. If you wish to play sound from within your UML instance, and the host digital signal processor (dsp) or mixer devices are different from these, you'll need to use these switches.

- `xterm=`*`terminal emulator,title switch,exec switch`*—This switch allows the use of terminal emulators besides `xterm` for UML consoles and serial lines. The arguments specify how to invoke the emulator with a given title and command running within it. The defaults, for `xterm`, are `-T` and `-e`, meaning that the title is specified with the `-T` switch and the command to run within the xterm follows the `-e` switch. The values for `gnome-terminal` are `-t` and `-x`, so `xterm=gnome-terminal,-t,-x` would make UML use gnome-terminal instead of xterm.
- `initrd=`*`initrd image`*—This switch makes UML boot from an initial ramdisk (initrd) image. The image must be specified as a filename on the host.
- `iomem=`*`name,file`*—This specifies the use of a host file as an I/O memory (iomem) region. *`name`* is the name of the driver that is going to own the region of memory. *`file`* is the name of the host file to be mapped into the UML instance's physical memory region. A demo iomem driver can be found in `arch/um/drivers/mmapper_kern.c` in the UML source tree.
- `mem=`*`size`*—Use this to specify the size of the UML instance's physical memory as a certain number of kilobytes, megabytes, or gigabytes via the `K`, `M`, or `G` suffixes, respectively. This has no relation to the amount of physical memory on the host. The UML instance's memory size can be either less or more than the host's memory size. If the UML memory size is more, and it is all used by the UML instance, the host will swap out the portion of the UML instance's memory that it thinks hasn't been used recently.
- `root=`*`root device`*—This option specifies the device containing the root filesystem. By default, it is `/dev/ubda`.
- `ncpus=`*`n`*—With `CONFIG_SMP` enabled, this switch specifies the number of virtual processors in the UML instance. If this is less than or equal to the number of processors on the host, the switch will enable that many threads to be running simultaneously, subject to scheduling decisions on the host. If there are more virtual processors than host processors, you can use this switch to determine the amount of host CPU power the UML instances can consume relative to each other. For example, a UML instance with four processors is entitled to twice as much host CPU time as an instance with two processors.
- `eth`*`n`*`=`*`interface configuration`*—This configures the host side of a network interface, making the device available and able to

receive and transmit packets. The interface configuration is summarized in Table 8.1 and described completely in Chapter 7.

- ☞ `fake_ide`—This switch creates IDE entries in `/proc` that correspond to the `ubd` devices in the UML instance, which sometimes helps make distribution install procedures work inside UML.
- ☞ `ubd<`*`n`*`><`*`flags`*`>=`*`filename`*`[:`*`filename`*`]`—This configures a UML block device on the host. *`n`* specifies the device to be configured; either letters (`a` through `z`) or numbers can be used. Letters are preferred because they don't encourage the belief that the unit number on the command line is the same as the minor number within UML. `ubda` (and `ubd0`) has minor number 0 and `ubdb` (and `ubd1`) has minor number 16 since each device can have up to 16 partitions.

  `flags` can be one or more of the following.

  - `r`—The device is read-only—read-write mounts will fail, as will any attempt to write anything to the device.
  - `s`—All I/O to the host will be done synchronously (`O_SYNC` will be set).
  - `d`—This device is to be considered as strictly data (i.e., even if it looks like a COW file, it is to be treated as a standalone device).
  - `c`—This device will be shared writable between this UML instance and something else, normally another UML instance. This would generally be done through a cluster filesystem.

  Either one or two filenames may be provided, separated by either a comma or a colon. If two filenames are specified, the first is a COW file and the second is its backing file. You can obtain the same effect by specifying the COW file by itself, as it contains the location of its backing file. Separating the two files by a colon allows shell filename completion to work on the second file.
- ☞ `udb`—This option exists for the sole purpose of catching `ubd` to `udb` typos, which can be impossible to spot visually unless you are specifically looking for them. Adding this to the UML command line will simply cause a warning to be printed, alerting you to the typo.

DEBUGGING OPTIONS

The debugging options come in two groups—those that make kernel debugging possible in `tt` mode and those that disable use of host features in order to narrow down UML problems.

In the first group are two options for specifying that we want debugging and whether we want to use an already-running debugger.

- `debug`—In `tt` mode, this causes UML to bring up `gdb` in an xterm window in order to debug the UML kernel.
- `gdb-pid=<pid>`—In `tt` mode, this switch specifies the process ID of an already-running debugger that the instance should attach to.

These may go away in the future if `tt` mode support is removed from UML.

The second group of options allows you to selectively disable the use of various host capabilities.

- `aio=2.4`—This switch causes UML to avoid the use of the AIO support on the host if it's present and to fall back to its own I/O thread, which can keep one request in flight at a time.
- `mode=tt`—This specifies that the UML instance should use `tt` mode rather than `skas` mode.
- `mode=skas0` and `skas0`—Both of these switches avoid the use of the `skas3` patch if it's present on the host, causing UML to use `skas0` mode, unless `mode=tt` is also specified, in which case `tt` mode will be used.
- `nosysemu`—This avoids the use of the `sysemu` patch if it's present on the host.
- `noprocmm`—This avoids the use of `/proc/mm` if the `skas3` patch is present on the host.
- `noptracefaultinfo`—This avoids the use of `PTRACE_FAULTINFO` if the `skas3` patch is present on the host.

MANAGEMENT OPTIONS

Several options control how you manage UML instances. The following control the location of the MConsole request and notification sockets and the `pid` file.

- `mconsole=notify:`*`socket`*—This specifies the UNIX domain socket that the `mconsole` driver will send notifications to.
- `umid=`*`name`*—This assigns a name to the UML instance, making it more convenient to control with an MConsole client.

- `uml_dir=`*`directory`*—This specifies the directory within which the UML instance will put the subdirectory containing its `pid` file and MConsole control socket. The name of this subdirectory is taken from the `umid` of the UML instance.

  These two control the behavior of the UML `tty` logging facility.

- `tty_log_dir=`*`directory`*—With `tty` logging enabled, this specifies the directory within which logging data will be stored.
- `tty_log_fd=`*`descriptor`*—This specifies that `tty` log data should be sent to an already-opened file descriptor rather than a file. For example, adding `10>tty_log tty_log_fd=10` to the UML command line will open file descriptor 10 onto the file `tty_log` and have all logging data be written to that descriptor.

INFORMATIONAL OPTIONS

Finally, three options cause UML to simply print some information and exit.

- `--showconfig`—This option prints the configuration file that the UML was built with and exits.
- `--version`—This switch causes the UML instance to print its version and then exit.
- `--help`—This option prints out all UML-specific command-line switches and their help strings, then exits.

UML Utilities Reference

humfsify

humfsify makes an existing directory structure mountable as a humfs filesystem. The directory hierarchy must have been copied to the data subdirectory of the current working directory with all file and directory ownerships and permissions preserved. The common usage of this would be to convert a ubd filesystem image into a humfs filesystem by loopback-mounting the image, copying it to ./data, and invoking humfsify.

humfsify has the following usage:

```
humfsify user group size
```

- user is a user ID, which can be a username or numeric user ID.
- group is a group ID, which can be a group name or numeric group ID.
- size is the size of the humfs filesystem, specified as a number of bytes, with the K, M, and G suffixes meaning kilobytes, megabytes, and gigabytes, respectively.

All of the files and directories under `data` will be made readable and writeable by, and owned by, the specified user and group. The previous ownerships and permissions will be recorded under two new directories, `file_metadata` and `dir_metadata`. The file `superblock` will be created in the current directory. This contains information about the metadata format and the amount of space available and used within the mount.

uml_moo

`uml_moo` merges a COW file with its backing file. It can do an in-place merge, where the new blocks from the COW file are written directly into the backing file, or create a new file, leaving the existing backing file unchanged.

Create a new merged file like this:

```
uml_moo [-b backing file] COW-file new-backing-file
```

Here's the usage for doing an in-place merge:

```
uml_moo [-b backing file] -d COW-file
```

The `-b` switch is used when the COW file doesn't correctly specify the backing file. This can be required when the COW file was created in a `chroot` jail, in which case the path to the backing file stored in the COW file header will be relative to the jail.

uml_mconsole

`uml_mconsole` is the UML control utility. It allows a UML instance to be controlled from the host and allows information to be extracted from the UML instance. It is one client of several for the MConsole protocol, which communicates with a driver inside UML.

It can be run in either single-shot mode, where the request is specified on the command line, or in command-line mode, where the user interacts with the `uml_mconsole` command line to make multiple requests of a UML instance.

The single-shot usage is:

```
uml_mconsole umid request
```

- umid is the name given to the UML instance. This is specified on the UML command line. If none is provided there, the instance will create a random umid, which will be visible in the boot log.
- request is what will be sent to the UML instance. This is described fully below.

A single-shot request will send the request to the UML instance, wait for a response, and then exit. The exit code will be zero if the request succeeded and nonzero otherwise.

The command-line usage is:

```
uml_mconsole umid
```

uml_mconsole will present a prompt consisting of the umid of the UML instance that requests will be sent to. In this mode, there are two commands available that are handled by uml_mconsole and are not sent to the UML instance.

- switch *new-umid* changes the UML instance to which requests will be sent to the one whose umid is new-umid. The prompt will change to reflect this.
- quit exits uml_mconsole.

A few commands are implemented within the uml_mconsole client and are available in both modes.

- mconsole-version prints the version of the uml_mconsole client. This is different from the UML version that the version command returns.
- help prints all of the available commands and their usage.
- int sends an interrupt (SIGINT) to the UML instance. If it is running under gdb, this will break out to the gdb prompt. If it isn't, this will cause a shutdown of the UML instance.

The commands sent to the UML instance are as follows.

- version returns the kernel version of the UML instance.
- halt performs a shutdown of the kernel. This will not perform a clean shutdown of the distribution. For this, see the cad command below. halt is useful when the UML instance can't run a full shutdown for some reason.

- `reboot` is similar to `halt` except that the UML instance reboots.
- `config` *`dev=config`* adds a new device to a UML instance. See Table 8.1 for a list of device and configuration syntax.
- `config` *`dev`* queries the configuration of a UML device. See Table 8.1 for a list of device syntax.
- `remove` *`dev`* removes a device from a UML instance. See Table 8.1 for a list of device syntax.
- `sysrq` *`letter`* performs the `sysrq` action specified by the given letter. This is the same as you would type on the keyboard to invoke the host's `sysrq` handler. These are summarized in Table 8.2.
- `cad` invokes the Ctrl-Alt-Del handler in the UML instance. The effect of this is controlled by the `ca` entry in the instance's `/etc/inittab`. Usually this is to perform a shutdown. If a reboot is desired, `/etc/inittab` should be changed accordingly.
- `stop` pauses the UML instance until it receives a `go` command. In the meantime, it will do nothing but respond to MConsole commands.
- `go` continues the UML instance after a `stop`.
- `log` *`string`* makes the UML instance enter the string into its kernel log.
- `log -f` *`filename`* is a `uml_mconsole` extension to the `log` command. It sends the contents of *`filename`* to the UML instance to be written to the kernel log.
- `proc` *`file`* returns the contents of the UML instance's `/proc/`*`file`*. This works only on normal files, so it can't be used to list the contents of a directory.
- `stack` *`pid`* returns the stack of the specified process ID within the UML instance. This is duplicated by one of the SysRq options—the real purpose of this command is to wake up the specified process and make it hit a breakpoint so that it can be examined with `gdb`.

tunctl

`tunctl` is used to create and delete TUN/TAP devices. The usage for creating a device is:

```
tunctl [-b] [-u owner] [-t device-name] [-f tun-clone-device]
```

- ☞ The -b switch causes tunctl to print just the new device name. This is useful in scripts, so that they don't have to parse the longer default output in order to find the device name.
- ☞ -u specifies the user that should own the new device. If unspecified, the owner will be the user running the command. This can be specified either as a username or a numeric user. You can specify a user other than yourself, but only that user or root will be able to open the device or delete it.
- ☞ -t specifies the name of the new device. This is useful for creating descriptive TUN/TAP device names.
- ☞ -f specifies the location of the TUN/TAP control device. The default is /dev/net/tun, but on some systems, it is /dev/misc/net/tun.

The usage for deleting a TUN/TAP device is:

```
tunctl -d device-name [-f tun-clone-device]
```

More precisely, the -d switch makes the device nonpersistent, meaning that it will disappear when it is no longer opened by any process. The -f switch works as described above.

uml_switch

uml_switch is the UML virtual switch and has the following usage:

```
uml_switch [ -unix control-socket ] [ -hub ] [ -tap tap-device ]
```

- ☞ The -unix switch specifies an alternate UNIX domain socket to be used for control messages. The default is /tmp/uml/ctl, but Debian changes this to /var/run/uml-utilities/uml_switch.ctl.
- ☞ -hub specifies hub rather than switch behavior. With this enabled, all frames will be forwarded to all ports, rather than the default behavior of forwarding frames to only one port when it is known that the destination MAC is associated with that port.
- ☞ -tap is used to connect the switch to a previously configured TUN/TAP device on the host. This gives a uml_switch-based network access to the host network.

INTERNAL UTILITIES

A few of the UML utilities are used by UML itself and are not meant to be used on their own.

- `port-helper` helps a UML instance use the host's `telnetd` server to accept `telnet` connections. This is used when attaching UML consoles and serial lines to host portals and xterms.
- `uml_net` is the setuid network setup helper. It is invoked by a UML instance whenever it needs to perform a network setup operation that it has no permissions for. This includes configuring network interfaces and establishing routes and proxy arp on the host. This is to ease the use of UML networking in casual use, where the root user inside the UML instance can be trusted. A secure UML configuration should not use `uml_net` and should instead use preconfigured TUN/TAP devices or `uml_switch` to communicate with the host.
- `uml_watchdog` is an external process used to track when a UML instance is running. It communicates with the UML harddog driver, expecting some communication at least once a minute. If that doesn't happen, `uml_watchdog` takes some action, either to kill the UML instance or to notify the administrator with an MConsole hang notification.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

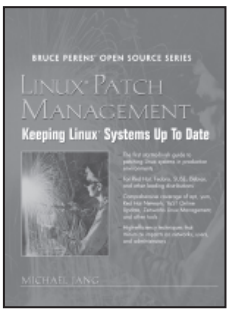
X

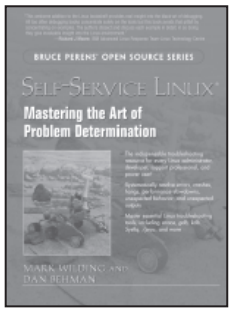
Z

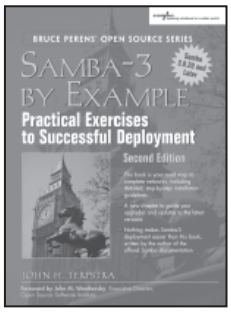
Bruce Perens' Open Source Series

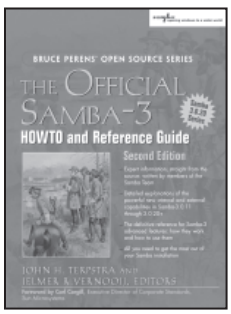
www.prenhallprofessional.com/perens

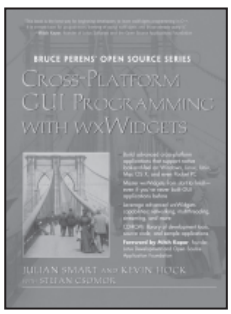
Bruce Perens' Open Source Series is a definitive series of books on Linux and open source technologies, written by many of the world's leading open source professionals. It is also a voice for up-and-coming open source authors. Each book in the series is published under the Open Publication License (www.opencontent.org), an open source compatible book license, which means that electronic versions will be made available at no cost after the books have been in print for six months.

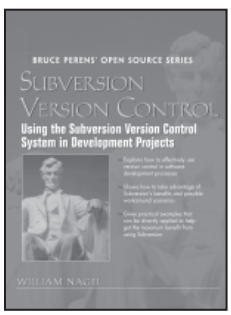


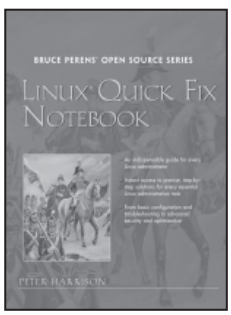












THIS BOOK IS SAFARI ENABLED

INCLUDES FREE 45-DAY ACCESS TO THE ONLINE EDITION

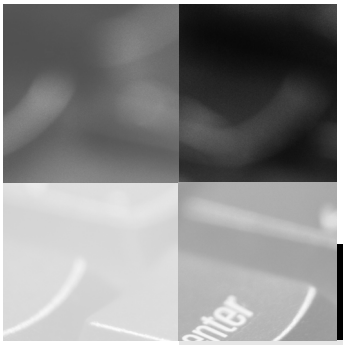
The Safari® Enabled icon on the cover of your favorite technology book means the book is available through Safari Bookshelf. When you buy this book, you get free access to the online edition for 45 days.

Safari Bookshelf is an electronic reference library that lets you easily search thousands of technical books, find code samples, download chapters, and access technical information whenever and wherever you need it.

TO GAIN 45-DAY SAFARI ENABLED ACCESS TO THIS BOOK:

- Go to **http://www.prenhallprofessional.com/safarienabled**
- Complete the brief registration form
- Enter the coupon code found in the front of this book on the "Copyright" page

If you have difficulty registering on Safari Bookshelf or accessing the online edition, please e-mail customer-service@safaribooksonline.com.



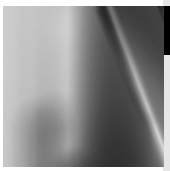
informIT

www.informit.com

YOUR GUIDE TO IT REFERENCE

Articles

Keep your edge with thousands of free articles, in-depth features, interviews, and IT reference recommendations – all written by experts you know and trust.



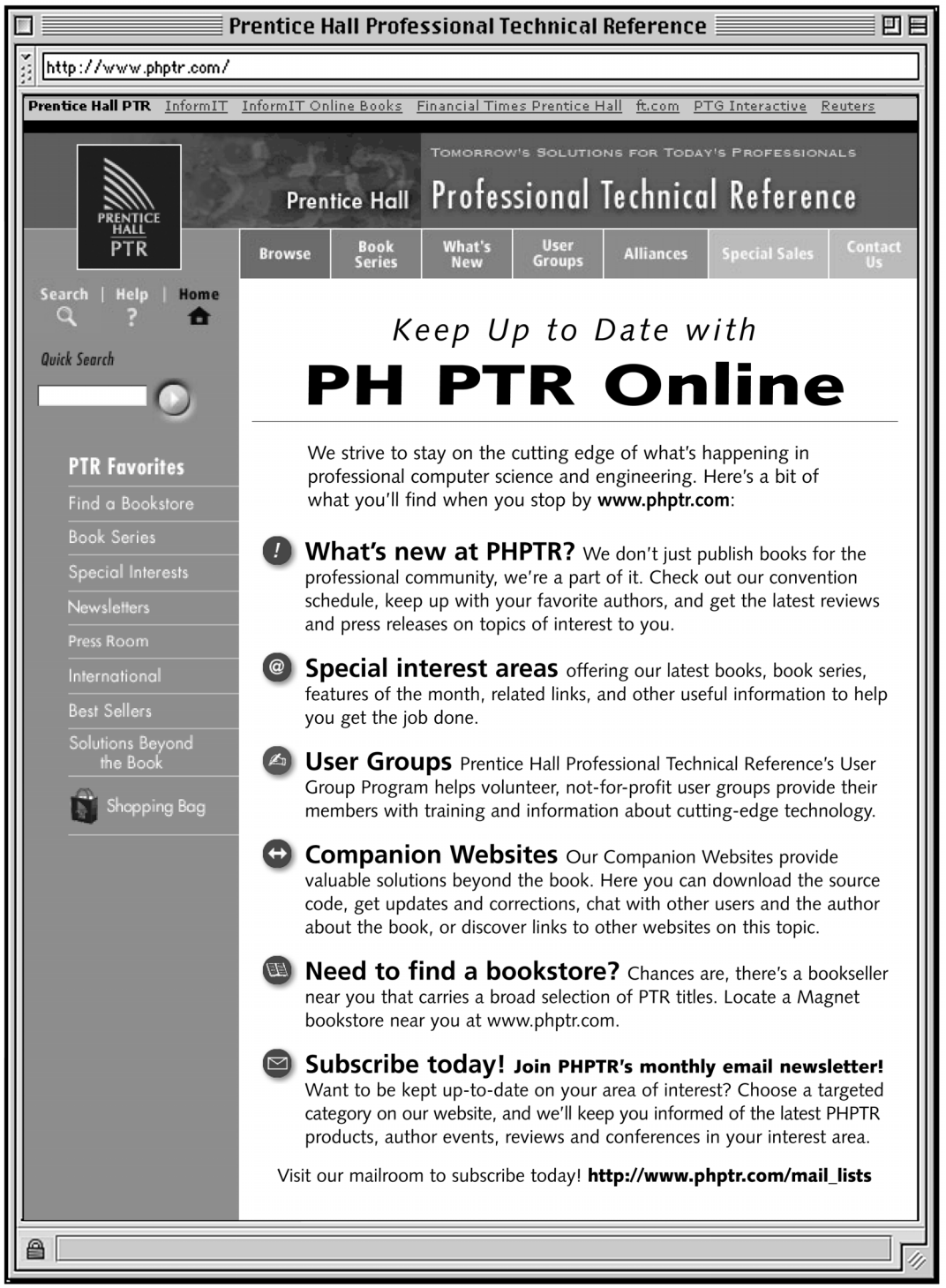
Online Books

Answers in an instant from **InformIT Online Book's** 600+ fully searchable on line books. For a limited time, you can get your first 14 days **free**.

POWERED BY Safari® TECH BOOKS ONLINE®

Catalog

Review online sample chapters, author biographies and customer rankings and choose exactly the right book from a selection of over 5,000 titles.


Prentice Hall Professional Technical Reference
http://www.phptr.com/
Prentice Hall PTR InformIT InformIT Online Books Financial Times Prentice Hall ft.com PTG Interactive Reuters
PRENTICE HALL PTR
Prentice Hall
Tomorrow's Solutions for Today's Professionals
Professional Technical Reference
Browse
Book Series
What's New
User Groups
Alliances
Special Sales
Contact Us
Search | Help | Home
Quick Search
PTR Favorites
Find a Bookstore
Book Series
Special Interests
Newsletters
Press Room
International
Best Sellers
Solutions Beyond the Book
Shopping Bag
Keep Up to Date with
PH PTR Online
We strive to stay on the cutting edge of what's happening in professional computer science and engineering. Here's a bit of what you'll find when you stop by www.phptr.com:
What's new at PHPTR? We don't just publish books for the professional community, we're a part of it. Check out our convention schedule, keep up with your favorite authors, and get the latest reviews and press releases on topics of interest to you.
Special interest areas offering our latest books, book series, features of the month, related links, and other useful information to help you get the job done.
User Groups Prentice Hall Professional Technical Reference's User Group Program helps volunteer, not-for-profit user groups provide their members with training and information about cutting-edge technology.
Companion Websites Our Companion Websites provide valuable solutions beyond the book. Here you can download the source code, get updates and corrections, chat with other users and the author about the book, or discover links to other websites on this topic.
Need to find a bookstore? Chances are, there's a bookseller near you that carries a broad selection of PTR titles. Locate a Magnet bookstore near you at www.phptr.com.
Subscribe today! Join PHPTR's monthly email newsletter! Want to be kept up-to-date on your area of interest? Choose a targeted category on our website, and we'll keep you informed of the latest PHPTR products, author events, reviews and conferences in your interest area.
Visit our mailroom to subscribe today! http://www.phptr.com/mail_lists