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