User-mode Linux

Jeff Dike

30th June 2005

Abstract

User-mode Linux is the port of the Linux kernel to userspace. It runs a Linux virtual machine in a set of processes on a Linux host. A UML virtual machine is capable of running nearly the same set of processes as the host. It lends itself to a variety of applications, such as kernel development, security applications like sandboxing and jailing, and virtual networking. This paper describes the design and implementation of UML, current and future applications for it, and future work.

1 Introduction

UML is a port of Linux to itself. That is, it treats Linux as a platform to which the kernel can be ported, like platforms such as Intel and Alpha. The architecture-dependent code which comprises UML implements all of the low-level hardware support that the generic kernel needs in terms of Linux system calls. It is implemented using only system calls - there are no special patches or hooks needed in the host kernel.

It runs the same userspace as the host kernel, and is able to run almost same range of applications - the exceptions are a few highly non-portable things such as

- installation procedures which aggressively probe for hardware using privileged instructions
- emulators such as dosemu, wine, and plex86

Userspace code runs natively on the processor without any sort of instruction emulation.

Processes running inside UML have no access to host resources that were not explicitly provided to the virtual machine.

2 Devices

All devices accessible inside the virtual machine are themselves virtual. They are constructed from the appropriate abstractions provided by the host. UML supports the full range of devices expected of a Linux box:

Consoles and serial lines UML has a main console, which is typically the window in which it was run, as well as virtual consoles and serial lines, which are exactly analogous to their counterparts on a physical machine. However, on a physical machine, consoles and serial ports are different physical devices. On UML, there is no real distinction, so they share most of their code. Both can be attached to a variety of host devices, including ttys, ptys, pts devices, xterms, sockets, and already-existing file descriptors.

Block devices UML has a single block driver which provides access to anything on the host which can be mounted. Normally, it is used to mount filesystems from images in files in the host filesystem. However, it may also be used to provide access to host block devices such as CD-ROMs, floppies, or raw disk partitions.

Network devices There is a single network driver which, through a number of backends, provides UML network access via a number of host mechanisms. These include slip, ethertap,

TUN/TAP, and a socket to a routing daemon. The slip backend is able to exchange IP packets with the host and other machines with the host as router. The ethertap and TUN/TAP backends exchange ethernet frames with the host and outside network. The routing daemon may be used to set up a totally virtual ethernet with no connection with the host or physical network, as well as a virtual network which is connected to the physical net through the daemon.

3 Design and implementation

3.1 Kernel mode and user mode

UML, like all Linux ports, has to provide to the generic kernel all of the facilities that it needs in order to run. A basic platform facility is a distinction between an unprivileged user mode and a privileged kernel mode. Hardware platforms provide a built-in mechanism for switching between these two modes and enforcing the lack of privileges in user mode. However, Linux provides no such mechanism to its processes, so UML constructs it using the ptrace system call tracing mechanism.

UML has a special thread whose main job is to ptrace almost all of the other threads. When a process is in user space, its system calls are being intercepted by the tracing thread. When it's in the kernel, it's not under system call tracing. This is the distinction between user mode and kernel mode in UML.

The transition from user mode to kernel mode is done by the tracing thread. When a process executes a system call or receives a signal, the tracing thread forces the process to run in the kernel if necessary and continues it without system call tracing.

The transition back is also performed by the tracing thread, but it's requested by the process. When it's finished executing in the kernel, because it finished either a system call or trap, it sends a signal (`SIGUSR1`) to itself. This is intercepted by the tracing thread, which restores the process state if necessary and continues the process with tracing on.

3.2 System call virtualization

With a mechanism in place to intercept process system calls and switch between user and kernel mode and back, virtualizing system calls is fairly straightforward.

The system call obviously must be annulled in the host kernel. This is done by changing the register containing the system call number to `__NR_getpid`. When this is done, the tracing thread saves the process registers in the thread structure, and imposes some previously saved state. This new state causes the process to start executing the system call handler, which reads the system call and arguments from its saved registers and calls the system call code.

When the system call is finished, the process stores the return value in its saved registers, and requests that the tracing thread return it to user mode. The tracing thread restores the saved registers, and continues the process with system call tracing on.

3.3 Traps and faults

A processor trap is the other mechanism which can cause a process to enter the kernel. On UML, these are implemented with Linux signals. The entry to and exit from the kernel are similar to the case of a system call. When a process receives a signal, the tracing thread sees it before the process does. When this happens, the process is continued in kernel mode, but without saving the process state or imposing any new state. UML establishes its own handlers for all important signals, so when the process is continued, it continues into one of these handlers, which implement the kernel's interpretation of the signal.

3.3.1 Device interrupts

External device interrupts are implemented with `SIGIO`. The drivers arrange that whenever input arrives, a `SIGIO` is generated. The `SIGIO` handler figures out (currently using select) which file descriptors have waiting input, and from that determines what IRQ is associated with each descriptor. At that point, it calls into the standard IRQ code in order to handle the interrupt.

3.3.2 Timer interrupts

The clock is implemented using Linux timers. These deliver `SIGALRM` or `SIGVTALRM`,depending on whether the interrupted process is the idle thread or not. The difference is that the idle thread sleeps while other threads don't, so it needs to receive `SIGALRM`. These signals get the same treatment as device interrupts - they are passed into the IRQ code for processing.

3.3.3 Memory faults

Memory faults are implemented with `SIGSEGV`. When a UML process makes an invalid memory access, the host will generate a `SIGSEGV` for it. The kernel `SIGSEGV` handler figures out whether the access is legitimate and it faulted only because that page had not yet been mapped into the process or whether it is just an illegal access. The first case is handled by calling the generic page fault handler and mapping the new page into the process. If the access was illegal, then a SIGSEGV is queued to the process, and it will either die or handle the signal when it tries to return to userspace.

3.4 Context switching

Each UML process has its own thread in the host kernel. So, a context switch from one to another involves stopping the outgoing process and continuing the incoming one.

UML also gives each of its processes its own address space. This speeds up context switching. If there were only one address space multiplexed among all processes, then a context switch would involve walking it and totally remapping it for the incoming process.

Pages in the incoming process may have been unmapped and reused while it was switched out. The host process will still have them mapped, so it is necessary to unmap them before it is allowed to return to userspace. These pages are identified by a pair of UML-specific pte bits which are set in ptes that are out-of-date with respect to the host process address space. When the pte is modified, these bits are set appropriately and they are cleared when the host mapping is updated.

A final subtlety with context switching is the handling of `SIGIO`. A `SIGIO` on a file descriptor will be queued to the process registered as its recipient of asynchronous notification. So, when a context switch happens, that `SIGIO` registration must be changed from the outgoing process to the incoming one.

3.5 Virtual memory emulation

Linux requires access to the platform's physical memory, kernel virtual memory, and a virtual address space for each process. UML implements this by creating a physical memory sized file and mapping it as a block into its address space. This provides the virtual machine's physical memory area.

Kernel and process virtual memory are implemented by mapping individual pages from that file into the appropriate places in the virtual address spaces.

The kernel's text and data are located in the process address space. So, effectively, there are some unusable holes in the process virtual memory area.

3.6 Host filesystem access

UML has a virtual filesystem, hostfs, which provides direct access to the host filesystem. This is done by implementing the VFS interface in terms of file access calls on the host. Most VFS operations translate directly into equivalent libc calls on the host.

hostfs can also be the root filesystem. This is done by registering it as a device filesystem as well as a virtual filesystem, and having the block driver recognize when it's booting from a directory rather than a file and faking a hostfs superblock. When hostfs sees the faked superblock, it claims the filesystem and performs all subsequent operations itself, without further involvement from the block driver.

4 Applications

4.1 Kernel development

UML was originally done as a kernel development tool. I wanted to do kernel work on my single machine and couldn't think of any reasonable way of doing

that except by doing a userspace port of Linux. This has turned out to be its most popular use so far. A number of kernel developers and projects have started using UML as their primary development platform, and booting their work on physical machines later in the debugging process.

Any project which doesn't involve machine-dependent code or direct hardware access can do their development under UML and the code will run unchanged in a native kernel.

It is also possible to do driver development under UML. For some types of hardware, such as SCSI and USB devices, a userspace driver can claim a physical device and control it. At the time of writing, there was a USB driver and several people were expressing interest in writing a similar SCSI driver.

Other types of devices can be controlled by a userspace driver, with some help from the host kernel. PCI devices, for example, can have their I/O memory mapped into the address space of the userspace driver, which can then directly read and write the device memory. There would need to be a stub driver in the kernel to do this ioremap and to do a few other things, such as probing the device, converting device interrupts into `SIGIO`s to the userspace driver, and possibly doing other things which need to be done in the kernel for some reason. Given such a stub, a driver in UML would have enough access to the device that it could be developed and debugged with confidence that it would work when put into a native kernel.

4.1.1 Profiling and coverage analysis

Since UML is a fairly normal set of processes, it is also possible to do profiling and test coverage analysis on it using the standard GNU tools, gprof and gcov. Support for them are part of the UML kernel configuration process. Once support is configured, using the tools is exactly the same as with any other process. In both cases, after the kernel is booted, tests run, and the kernel is halted, files are written out which are analyzed by the appropriate tool. The gprof runtime library writes out the standard gmon.out file, which is analyzed by gprof, while the gcov runtime writes out a set of hit counts for each file in the kernel which is used by gcov to produce an annotated listing of the file showing how many times each line was executed and which were not executed at all.

4.1.2 gprof and gcov support

Supporting gprof and gcov does require some work inside UML. gprof allocates a buffer to store its profiling information. This buffer must be shared among all of the UML threads, or each would get its own private copy of it. This is done by locating that buffer and replacing it with a segment of shared memory. Also, SIGPROF and the profiling timer need to be initialized properly for each new UML thread.

The gcov runtime outputs its accumulated data when the process exits normally. Unfortunately, in a multithreaded process, the first normal exit causes that to happen. So, UML needed to be changed slightly so that the only thread that exits normally is the tracing thread when the virtual machine halts. All other threads are killed when they are no longer needed.

4.1.3 gdb support

Supporting gdb was significantly more complicated. Since use of ptrace is essential for the virtualization of system calls, a process can only be under ptrace by one other process, and gdb also requires ptrace, there is a problem attaching gdb to a UML process.

This is fixed by having gdb started and ptraced by the tracing thread and, by intercepting its system calls, faking it into believing that it is attached to UML. gdb's calls to ptrace and a few other system calls are intercepted, emulated by the tracing thread, and the return values imposed on gdb. This works well enough that the user can't tell that gdb isn't really attached to UML.

This support has also been extended to allow external debuggers to be attached. This enables UML to be debugged by an already-running gdb, such as one running under a front-end like emacs or ddd. It also allows debuggers other than gdb, like strace, to be attached to UML.

4.2 Hosting and sandboxing

Since UML provides a full-blown virtual machine, there has been a lot of interest in it from the hosting industry. Providing customers with individual virtual machines promises to combine the advantages of a dedicated machine for each customer with the administrative convenience of a small number of servers.

UML would give customers their own virtual machine that they could set up any way they want, with services that normally aren't provided by a hosting service because of potential resource consumption or security concerns. These would be hosted on a small number of larger servers, greatly simplifying the administrative work on the part of the hosting company.

It's also possible that UML virtual machines can produce greater performance at lower cost to the customer. For example, if a number of virtual machines are serving relatively low-traffic web sites, then only one may be active at any given time. This virtual machine will have the entire large server to itself. In constrast, with a normal colocation arrangement, this site would served by a single small physical machine. Depending on the virtualization overhead imposed by UML, it is possible that the virtual machine running on the large server could outperform the smaller physical machine.

A related application is sandboxing or jailing. Since UML is going to be a completely secure jail for whatever is running inside it, it has obvious uses for confining untrusted users or processes. A service that provides accounts for the public could isolate each user inside a virtual machine, preventing them from damaging the host or harrassing each other in any way. They could be given root access inside the virtual machine, which would let them destroy anything inside it, but they couldn't touch anything else.

UML can also be used to confine system services whose security is suspect. Prominent example include bind and sendmail. A sysadmin who wants to be sure that someone can't break in through one of these servers can run it inside a virtual machine. If someone cracks it, they gain access to the virtual machine, not the host. So in order to do any actual damage, they'd also need an exploit to break out of UML.

4.3 Multiple environments

There are a number of tools and systems which realistically require a separate machine to maintain. Examples include distribution installation procedures, package managers, and network services. Commonly, people who maintain or develop multiple versions of these tools keep one physical machine for each version.

UML offers the ability to move all that activity back on to a single machine by putting each version in a different virtual machine. Aside from the logistical convenience of not having to maintain multiple physical machines, there are also the advantages that come from virtual machines being more convenient to manage than physical ones. Resource allocation between them is far more flexible, they are much quicker to boot up and shut down, and they can be created and destroyed at will.

These make it much more convenient to do this sort of development inside multiple UML instances than on separate physical machines.

4.4 Linux compatibility for other operating systems

Since UML is a full-blown Linux kernel, it provides a completely authentic Linux environment to its processes. This is not exactly a revolutionary concept, but it becomes interesting when UML is ported to operating systems other than Linux.

A number of commercial Unixes are starting to acquire various levels of Linux compatibility. However, the highest level of compatibility would be gained by actually running Linux on those other operating systems. This is what exactly what a UML port would do.

How much work it would be depends on the target. The other Unixes might be fairly easy porting targets because of their similarity to Linux. They would just need something equivalent to the Linux ptrace system call tracing mechanism and a few other things. More foreign operating systems such as Windows would obviously be harder, but they would also be interesting.

There has been some work done on a Windows

port, but it is fairly preliminary at the time of writing. UML can boot up to the point of starting init, but that was accomplished by stubbing out a lot of code that will turn out to be important.

5 Future work

5.1 SMP

Currently, UML implements a uniprocessor virtual machine. The only barrier to enabling SMP is making the UML architecture layer SMP safe. Virtual SMP involves letting UML have one process per virtual processor runnable on the host at a a time. This is done by starting one idle thread per processor and then letting them schedule normally.

Once SMP is working, a number of interesting possibilities open up. Obviously, this would allow developers to do SMP development without needing access to SMP hardware. It would also allow developers who have SMP hardware to test kernels on more processors than they have.

It would also allow more exotic hardware to be simulated. This is especially interesting for cluster hardware. The cluster topology and other features could be emulated with UML, allowing support to be done by people without access to the hardware.

Another interesting possibility is running a single UML instance across multiple physical hosts. This would be done by partitioning the UML physical memory between the nodes, and faulting those pages from node to node as needed. This can be done with a fairly simple change to the memory fault handler. It would keep track of what pages are resident on the current node, and when a fault occurred on a non-resident physical page, it would ask the node that owned it for the data. That other node would unmap it and pass the data over, where it would be copied in and mapped in to the physical memory area on that node.

This is an extreme example of NUMA, and it won't perform at all well until Linux has pretty good NUMA support. On the other hand, this would provide a simulated NUMA platform to anyone running Linux, so it would potentially bring a lot more talent to bear on the problem of NUMA support, which could make it reality more quickly.

5.2 hostfs extensions

Currently, hostfs translates VFS operations into libc file operations on the host. However, there are other possibilities. The userspace side of hostfs could just as easily be operating on a different machine or on a totally different type of data.

It would be straightforward to put a network link between the kernel and usermode pieces of hostfs, allowing UML to directly mount remote filesystems. This would be the equivalent of the usermode nfs server that currently exists on Linux.

The userspace piece of hostfs could easily be used to mount something other than a filesystem inside UML. For example, it could be used to mount a SQL database as a filesystem inside UML.

hostfs could also be used to mount multiple external resources on the same UML mount point. An example of this would be to mount a number of nearly identical external filesystems inside UML and install software onto all of them simultaneously.

5.3 Performance

The performance of UML is dominated by the context switches back and forth between its processes and the tracing thread. So, any major performance improvements have to focus on eliminating the tracing thread.

This would require a mechanism for doing system call interception without using a separate thread. I'm planning on doing this by adding a new system call path in the host which delivers a signal to the process whenever it does a system call. The signal handler would be the current UML system call handler which reads the system call and arguments and executes the system call.

Another area for performance improvement is context switching. The problem is the address space scan which is required to order to bring the host address space up-to-date with UML. This could be eliminated by allowing address spaces to be created, manipulated, and switched from userspace. This would allow

the address spaces of all UML processes to be kept up-to-date, which would allow the address space scan to be eliminated.

5.4 Ports

5.4.1 Architecture ports

UML currently runs only on Linux/i386. There is no fundamental reason that it can't be ported to other Linux platforms. The main obstacle is likely to be a ptrace limitation that existed in the i386 port before I fixed it and which exists in several other ports. ptrace can be used to arbitrarily change system call arguments, but on some ports, it can't change the actual system call number. This is needed to nullify system calls in the host because that it done by turning them into getpid. This is still a problem on at least sparc and IA64. It is not a problem on ppc.

There is an ongoing ppc port effort as well as a somewhat moribund IA64 port. At the time of writing, the ppc port had booted up to the point of starting to exec init.

5.4.2 Operating system ports

As already stated, this is interesting because UML running on another operating system would provide it with a completely authentic Linux environment. UML is fairly Linux-specific at this point, but not so much so that ports are out of the question. It uses only Linux system calls, with no special kernel hooks required. However, some of those system calls are not found on all other platforms.

The principal one is the system call interception and modification capability of ptrace. That is essential for UML to run. If UML is to be ported to another operating system, it needs to provide the ability to intercept the system calls that Linux binaries make on that architecture.

Linux has a very general mmap, which UML takes advantage of. It is helpful if the target platform allows mmap with page granularity and multiple mappings of the same page into the same address space. UML also unmaps portions of its executable image and replaces them with shared pages containing the same data.

A little-used, but very useful feature of Linux is the ability to create a new process and choose which pieces of the parent process will be shared with the child. UML uses this capability to share file descriptors, but nothing else, between all UML processes. This is useful because communication with the host is done largely through file descriptors which are opened by one process and accessed by another. A good example of this is the file descriptor to a file containing a UML filesystem. It is opened in the context of the mount which mounts the filesystem inside UML. That process will have died by the time an ls tries to read it. With the ability to share file descriptors, the descriptor opened by the mount process is preserved in all the other processes, and is ultimately inherited by the ls that comes along to read the filesystem.

This can be done by making UML processes be threads. In this case, everything, including address spaces will be shared by all UML threads. This will work, but it will kill context switch performance, since the address space would have to be completely remapped for the incoming process. It would also make SMP support problematic since two or more threads will be running entirely different processes simultaneously, and that can't be done in a single address space.

5.5 UML as a development platform

A recent possibility, and possibly the most interesting, is that UML might make a good process-level development platform. The Linux kernel includes a good threads implementation, general interrupt-driven I/O, memory management, and a variety of other things. UML, as a Linux kernel, makes all of those available in userspace.

Obvious applications for this sort of platform would be ones that require a virtual-machine-like environment. Examples include interpreted language environments such as Java and Perl. These would not require the full-blown virtual machine that's isolated from the host that UML currently provides. So, this could provide an impetus to figure out how to configure out pieces of the kernel which are now considered essential, like virtual memory support or the scheduler. Making the kernel more configurable in this way

would make UML a better candidate for this sort of application.

The facilities in the kernel, particularly threads and high-performance I/O, are also required by high-performance servers. So it's possible that UML would make a good platform for userspace servers.

The fact that the core kernel is a relatively small piece of code that's been worked on by a large number of talented programmers adds to the attraction. Any application that's based on UML would get for free the speed, efficiency, and robustness of the Linux kernel.

There is also the possibility of using existing kernel subsystems to manage completely different sorts of resources. For example, the memory management system could be used to manage language objects instead of raw memory and the scheduler could be used to schedule task-like things which are not processes. Since the true nature of the resources that the generic kernel is managing is not totally evident to it, but it evident to the architecture layer beneath it, a fair amount of fakery can be pulled off by the architecture layer. It would be coded to know what the resources really are, while the generic kernel remains unchanged and manages them just like it currently manages raw memory and machine processes.

6 Conclusion

User-mode Linux is significant in a number of ways. From a theoretical perspective, it has demonstrated that the Linux system call interface is sufficient to implement itself. From a somewhat more practical perspective, the actual implementation has showed areas where the current Linux functionality is a bit lacking. These include the ability for ptrace to change system call numbers, the ability to manipulate address spaces, and for a process to intercept its own system calls. Fixing these will not only improve the performance (and on some platforms, the existence, in the case of ptrace) of UML, they will also help other applications. Fixing ptrace on i386 has made several other applications possible, and this may happen again with the other changes in the generic kernel that UML needs.

It provides a significant capability that previously didn't exist, which is making a number of interesting applications possible for Linux. The kernel debugging capabilities could speed up the overall development of Linux, and the gcov support may allow the development of coverage test suites which would allow the code to be exhaustively tested.

The possibilities for UML in the hosting industry may make it more popular than it already is and move it into segments where it is currently not popular. It is also possible that it will open up opportunities that didn't exist before, giving Linux ownership of those areas by default.

The wildcard application is the use of UML as a general purpose development platform. It may come to nothing, or it may be the killer app for UML. If it is the killer app, then Linux, in the form of UML, would become more prevalent than it is today, even on non-Linux platforms. This in turn would drive more changes in the code as userspace developers and kernel developers cooperate on the same code base. The presence of this as a significant platform in the industry would inevitably have repercussions on other platforms.

UML has a great deal of as-yet unrealized potential, and it will be exciting if it actually realizes that potential.