A user-mode port of the Linux kernel[*]

Jeff Dike

Abstract

The Linux kernel has been ported so that it runs on itself, in a set of Linux processes. The result is a user space virtual machine using simulated hardware constructed from services provided by the host kernel. A Linux virtual machine is capable of running nearly all of the applications and services available on the host architecture. This paper describes the capabilities of a user-mode virtual machine, the design and implementation of the port, some of the applications that it lends itself to, and some of the work that remains to be done.

1 Overview

1.1 Description of functionality

The user-mode kernel is a port of the Linux kernel to the Linux system call interface rather than to a hardware interface. The code that implements this is under the arch interface, which is the internal kernel interface which separates architecture-independent code from architecture-dependent code. This kernel is a full Linux kernel, lacking only hardware-specific code such as drivers.

It runs the same user space as the native kernel. Processes run natively until they need to enter the kernel. There is no emulation of user space code.

Processes running inside it see a self-contained environment. They have no access to any host resources other than those explicitly provided to the virtual machine.

1.2 Device support

All devices seen by the user-mode kernel are virtual from the point of view of the host. They are constructed from software abstractions provided by the host. The following types of devices are provided:

Consoles The main console is whatever terminal the kernel was invoked in. In addition, virtual consoles are supported. By default, they execute an xterm when opened. Optionally, they can just allocate a pseudo-terminal which the user can connect to with a terminal program such as minicom or kermit.

Block devices The block device driver operates within a file on the host. Normally, this is a file containing a filesystem or swap space. However, any file on the host that is seekable is suitable. Among other things, this means that devices on the host can be accessed through their device files.

Serial lines The serial line driver allocates a pseudo-terminal. Users wanting to connect to the virtual machine via a serial line can do so by connecting to the appropriate pseudo-terminal with a terminal program.

Network devices There are two network device drivers. The old network driver communicates with the host networking system through a slip device in the host. The virtual machine's side of the connection is a pseudo-terminal in the host which appears as a network device inside. There is also a newer network driver which uses an external daemon to pass Ethernet frames between virtual machines. This daemon can also attach this virtual network to the host's physical Ethernet by way of an ethertap device. With an appropriate packet forwarding policy in the daemon, the virtual Ethernet can be transparently merged with the physical Ethernet, totally isolated from it, or anything in between.

2 Design and implementation

2.1 Overview

The final and most important piece of hardware that needs to be implemented virtually is the processor itself, including memory management, process management, and fault support. The kernel's arch interface is dedicated to this purpose, and essentially all of this port's code, except for the drivers, is under that interface.

[*]This paper is to be published in the *Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta*

A basic design decision is that this port will directly run the host's unmodified user space. If processes are going to run exactly the same way in a virtual machine as in the host, then their system calls need to be intercepted and executed in the virtual kernel. This is because those processes are going to trap directly into the host kernel, rather than the user-mode kernel, whenever they do a system call. So, the user-mode kernel needs a way of converting a switch to real kernel mode into a switch to virtual kernel mode. Without it, there is no way to virtualize system calls, and no way to run this kernel.

This is implemented with the Linux ptrace system call tracing facility. A special thread is used to ptrace all of the other threads. This thread is notified when a thread is entering or leaving a system call, and has the ability to arbitrarily modify the system call and its return value. This capability is used to read out the system call and its arguments, annull the system call, and divert the process into the user space kernel code to execute it.

The other mechanism for a process to enter the kernel is through a trap. On physical machines, these are caused by some piece of hardware like the clock, a device, or the memory management hardware forcing the CPU into the appropriate trap handler in the kernel. This port implements traps with Linux signals. The clock interrupt is implemented with the `SIGALRM` and `SIGVTALRM` timers, I/O device interrupts with `SIGIO`, and memory faults with `SIGSEGV`. The kernel declares its own handlers for these signals. These handlers must run in kernel mode, which means that they must run on a kernel stack and with system call interception off. The first is done by registering the handler to run on an alternate stack, the process kernel stack, rather than the process stack. The second is accomplished by the handler requesting that the tracing thread turn off system call tracing until it is ready to re-enter user mode.

When a process is enters kernel mode, it is branching into a different part of its address space. On the host, processes automatically switches address spaces when they enter the kernel. The user-mode port has no such ability. So, the process and kernel coexist within the same address space. The design of the VM system is partly a question of address space allocation. Conflicts with process memory are avoided by placing the kernel text and data in areas that processes are not likely to use. The kernel image itself is linked so that it loads at `0x10000000`. The kernel expects the machine to have physical memory and kernel virtual memory areas. The physical memory area consists of a file mapped into each address space starting at `0x50000000`. The kernel virtual memory area is immediately after the end of the physical memory area. Virtual memory, both kernel and process, is implemented by re-mapping pages from the physical memory file into the appropriate place in the address space.

Each process within a virtual machine gets its own process in the host kernel. Even threads sharing an address space in the user-mode kernel will get different address spaces in the host.

Even though each process gets its own address spaces, they must all share the kernel data. Unless something is done to prevent it, every process will get a separate, copy of the kernel data. So, what is done is that the data segment of the kernel is copied into a file, unmapped, and that file is mapped shared in its place. This converts a copy-on-write segment of the address space into a shared segment.

To balance that awkwardness, the separate address space design allows context switches to be largely implemented by host context switches, with preemption driven by the `SIGVTALRM` timer.

`SIGIO` is used to deliver the other asynchronous events that the kernel must handle, namely device interrupts. The console driver, network drivers, serial line driver, and block device driver use the Linux asynchronous I/O mechanism to notify the kernel of available data.

2.2 Virtual machine initialization and shutdown

Before the kernel itself starts booting, some initialization needs to be done in order to make the process look enough like a real machine that the kernel can boot it up. This is analogous to the boot loader on a physical machine doing some hardware setup before running the kernel.

The process arguments are concatenated into the buffer in which the kernel expects to find its command line. Some arguments, which affect the configuration of the virtual machine and how it's to be debugged, are parsed at this point. The physical memory area is set up, some initialization of the task structure and stack of the idle thread is done, the idle thread is started, and the initial thread settles down to its permanent job as the tracing thread.

The idle thread calls `start_kernel` and the virtual machine boots itself up. There is some more architecture-specific initialization that needs to be done. `mem_init` makes memory available to the initial boot process, `paging_init` makes all free memory available to `kmalloc`, and the various drivers are registered and initialized.

At the other end of the virtual machine's lifespan, when `halt` or `reboot` are run, an architecture-specific routine is eventually called to do the actual machine halt or restart. In this port, that involves killing all processes which are still alive, including any helper threads which weren't associated with any virtual machine processes, and asking the tracing thread to finish the shutdown.

If the machine is being halted, the tracing thread simply exits. If it's being rebooted, it loops back to calling the machine initialization code. At that point, the machine boots back up just as it did when it was first run.

2.3 Process creation and destruction

When a new process is created, the generic kernel creates the task structure for the new process and calls the arch layer to do the machine-dependent part. This port creates a new process in the host for each new process in the virtual machine. This is done by the tracing thread. It needs to ptrace all new processes, and that is simplified if it's their parent.

The new thread starts life in a trampoline which does some initialization. It sets up its signal handlers for `SIGSEGV`, `SIGIO`, and `SIGVTALRM`, initializes the timer, and sets itself to be ptraced by its parent.

Once this initialization is done, it sends itself a `SIGSTOP`. When the tracing thread sees the thread stop itself, it sets the thread's system call return value to zero, while the same system call in the forking process returns the pid of the new process.

At this point, the parent's `fork` or `clone` is finished, and it returns.

At some later point, the new process will be scheduled to run. It is necessary that a process call `schedule_tail` before it's rescheduled. Also, the kernel stack state needed to start a system needs to be saved. Both of these are done at this point. Another signal is delivered to the new process, and the handler calls `schedule_tail`, goes into the system call handler, and stops itself. The tracing thread captures the process state at this point. Then the registers of the forking process (except for the one reserved for the system call return value, which is now zero) are restored, and the process is continued. Since the generic kernel arranged for the new address space to be a copy of the parent address space, and the new process has the same registers as the old one, except for the zero return value from the system call, it is a copy of its parent. At this point, its initialization is finished, and it's just like any other process in the virtual machine.

The other end of a process lifespan is fairly simple. The only resources that need to be cleaned up are some kmalloced buffers in the thread structure, which are freed, and the process in the host, which is killed.

2.4 System calls

System call virtualization is implemented by the tracing thread intercepting and redirecting process system calls into the virtual kernel. It reads out the system call and its arguments, then annuls it in the host kernel by changing it into `getpid`. In order to execute the system call in user space, the process is made to execute the system call switch on its kernel stack. This can be (and has been) done in two different ways.

The first is to use ptrace to impose a new set of register values and stack context on the process. This context represents an execution context which puts the process at the beginning of the system call switch. This context is constructed by the process, just after its creation, calling the switch procedure and sending itself a `SIGSTOP`. The tracing thread sees the `SIGSTOP` and saves the process register and stack state in the task structure. When this state is restored and the process continued, it emerges from the call to `kill` that it used to stop itself.

The second is to deliver a signal to the process before it's continued into the `getpid`. The Linux system call path checks for signals just before returning to user space, so that signal will be delivered immediately. The signal handler is the system call switch and it is installed so that it executes on an alternate stack (the process kernel stack). This has the same effect as manually restoring the context, but the host kernel does most of the work.

Regardless of which mechanism is used to impose the kernel execution context on the process, the tracing thread continues it with system call tracing turned off. The process reads the system call and its arguments from its thread structure and calls the actual system call procedure.

When it finishes, the return value is saved in the thread structure, and the process notifies the tracing thread that it needs to go back to user space. The tracing thread stores the return value into the appropriate register and continues the process again, with system call tracing turned back on. Now, the process starts executing process code again with the system call return value in the right place, and the user space code can't tell that anything unusual happened. Everything is exactly the same as if the system call had executed in the host kernel.

2.5 Context switching

If a process sleeps instead of returning immediately from a system call, it calls schedule. The scheduler selects a new process to run and calls the architecture-specific context switching code to actually perform the switch. In this port, that involves the running process sending a message to the tracing thread that it is being switched out in favor of another process. Since each process in the virtual machine is also a process in the host, the tracing thread performs the switch by stopping the old process and continuing the new one. The new process returns from the context switch that it entered when it last ran and continues whatever it was doing.

Sometimes, after a process is switched back in, its address space will need some updating. If some of its address space had been paged out while it was sleeping, those physical pages, with their new contents, will still be mapped. So, in this situation, the process will need to update its address space by unmapping pages which are no longer valid in its context. These pages are listed in a circular buffer whose address is stored in the process mm_struct. When a page is swapped out from a process while it's asleep, its address is appended to this buffer. When the process wakes up, it looks at this buffer to see if there are any changes to its address space that it hasn't applied yet. It then updates its address space and sets an index in its thread structure to point to the end of the buffer. This private index is necessary because many processes might share a virtual address space and an mm_struct. In general, their host address spaces will be in different states, depending on how long it's been since they've been updated. So, each process keeps track of how many address changes in this buffer it has seen.

There is a possibility that the buffer might wrap around while a process is asleep and some of the address space changes it needs to make have been lost. To avoid this, the index into the buffer that each process maintains is really an absolute count. The index is obtained by dividing the count by the buffer size and taking the remainder. If the buffer has wrapped, then the process count of address space changes will differ from the actual count by more than the number of slots in the buffer. In this case, the entire address space will be scanned and compared to the process page tables. Any differences will be fixed by remapping or unmapping the page concerned.

2.6 Signal delivery

When a signal has been sent to a process, it is queued in the task structure, and the queue is periodically checked. In this port, the check happens on every kernel mode exit. If a signal needs to be delivered, a SIGUSR2 is sent to the underlying process. The SIGUSR2 handler runs on the process stack, and it executes the process signal handler by simply making a procedure call to it.

Things are a little more complicated if the signal is to be delivered on a different, process-specified, stack. In this case, the alternate stack state, which is composed of register values and stack state, is imposed on the process with ptrace. This new state puts the process in the signal delivery code on the alternate stack, which invokes the process signal handler.

2.7 Memory faults

Linux implements demand loading of process code and data. This is done by trapping memory faults when a nonexistent page is accessed. In this port, a memory fault causes a SIGSEGV to be delivered to the process. The segmentation fault handler does the necessary checking to determine whether it was a kernel-mode or user-mode fault, and in the user-mode case, whether the page is supposed to be present or not. If the page is supposed to be present, then the kernel's page fault mechanism is called to allocate the page and read in its data if necessary. The segmentation fault handler then actually maps in the new page with the correct permissions and returns.

If the fault was not a legitimate page fault, then the machine either panics or sends a SIGSEGV to the process depending on whether it was in kernel mode or user mode.

An exception to this is when a kernel mode fault happens while reading or writing a buffer passed in to a system call by the process. For example, a process may call read with a bogus address as the buffer. The fault will happen inside the kernel, in one of the macros which copy data between the kernel and process. The macro has a code path that executes when a fault happens during the copy and returns an appropriate error value. The address of this path is put in the thread structure before the buffer is accessed. The fault handler checks for this address, and if it is there, it puts the fault address in the thread structure, copies the error code address into the ip of the sigcontext structure, and returns. The host will restore the process registers from the sigcontext structure, effectively branching to the error handler. The error handler uses the fault address in some cases to determine the macro return value. When this happens, the system call will usually react by returning EFAULT to the process.

2.8 Locking

There are three types of locking. On a uniprocessor, interrupts must be blocked during critical sections of code. In this port, interrupts are Linux signals, which are blocked and enabled using sigprocmask.

SMP locking involves a processor locking other processors out of a critical section of code until it has left it. The instructions needed to do this are not privileged on i386, so the SMP locking primitives are simply inherited from the i386 port.

The same is true of semaphores. The i386 semaphore primitives work in user space as well as in the kernel, so they are inherited from the i386 port.

2.9 IRQ handling

The IRQ system was copied verbatim from the i386 port. It works with almost no modifications. When a signal handler is invoked, it figures out what IRQ is represented by the signal, and calls `do_IRQ` with that information. `do_IRQ` proceeds to call the necessary handlers in the same way as on any other port.

The one difference between this port and others is that some IRQs are associated with file descriptors. When input arrives, the `SIGIO` handler selects on the descriptors that it knows about in order to decide what IRQs need to be invoked.

3 A virtual machine

The result of all of this infrastructure is that the Linux kernel runs in a set of Linux processes just as it does on physical hardware. The machine-independent portions of the kernel can't tell that anything strange is happening. As far as they are concerned, they are running in a perfectly normal machine.

When user-mode Linux is started, the normal kernel boot-up messages are written to its console, which is the window in which it was run. When the kernel has initialized itself, it runs `init` from the filesystem that it's booting from. What happens from that point is decided by the distribution that was installed in that filesystem.

Essentially all applications that run on the native kernel will run in a virtual machine in the same way. Examples include all of the normal daemons and services, including the Apache web server, `sendmail`, `named`, all of the network services, and X, both displaying as a client on the host X server, and as a local X server.

4 Applications

4.1 Kernel debugging

This port was originally conceived as a kernel debugging tool. A user-mode kernel port makes it possible to do kernel development without needing a separate test machine. It also enables the use of the standard suite of process development and debugging tools, like `gdb`, `gcov`, and `gprof`, on the kernel.

A difficulty with using `gdb` is that the kernel's threads are already being ptraces by the tracing thread. This makes it impossible for `gdb` to attach to them to debug them. Early on, this problem was dealt with by having the tracing thread detach from the thread of interest. Then, `gdb` could attach to it, and it could be debugged as a normal process. Rarely, when this was over, the thread was re-attached by the tracing thread, and the kernel continued. More often, the whole kernel was killed at the end of that debugging session.

Now, there is a mechanism that allows `gdb` to debug a kernel thread without needing to attach to it, and without needing to detach the tracing thread from it. This works by having the tracing thread start `gdb` under system call tracing. The tracing thread intercepts ptrace system calls and a few others made by `gdb`, executes them itself, nullifies the `gdb` system call, and writes the return value into the appropriate register in `gdb`. In this way, `gdb` is faked into believing that it is attached to the thread and is debugging it.

4.2 Isolation

Other uses of this port became apparent later. A number of applications involve isolating users of virtual machines from each other and from the host.

There are several reasons to want isolation. One is to protect the physical machine and its resources from a potentially hostile process. The process would be run in a virtual machine which is given enough resources to run. Those resources would not be valuable, so they could be easily replaced if destroyed. It would be given a copy of an existing filesystem. If it trashes that filesystem, then it would just be deleted, and a new copy made for the next sandbox. It would have no access to valuable information, and its use of the machine's resources would be easily limited. The virtual Ethernet driver also makes it easy to control its access to the net. The daemon that does packet routing could be made to do packet filtering in order to control what traffic the sandboxed process is allowed to send and receive.

A variation on this theme is to put a non-hostile, but untrusted service in a virtual machine. A service is untrusted if it's considered to be vulnerable to being used to break into a machine. `named` is such a service, since it has had at least one hole which led to a spectacular number of breakins. An administrator not wanting to see this happen again would run `named` in a virtual machine and set that machine to be the network's name server. `named` requests from outside would be passed directly from the host to the virtual machine. So, anyone successfully breaking into that service would be breaking into a virtual machine. If they realized that, they'd need to find another exploit to break out of the virtual machine onto the host.

Another use of this isolation is to allocate machine resources, whether they be CPU time, memory, or disk space, between competing users. A virtual machine is given access to a certain amount of machine resources when it's booted, and it will not exceed those resources. So, if a user runs a very memory-intensive process inside a virtual machine, the virtual machine, and not the physical machine, will swap. Performance inside the virtual machine may be bad, but no one else using the physical machine will notice. The same is true of the other types of resources. CPU time can be allocated through the assignment of virtual processors to virtual machines. If a virtual machine is given one processor, it will never have more than one process running on the host, even if it's running a fork bomb. Again, life will be miserable inside the virtual machine, but no one outside will notice.

This level of isolation may find a large market in the hosting industry. Current hosting arrangements vary from application-specific hosting such as Apache virtual hosting to `chroot` environments to dedicated, colocated machines. Dedicated machines are used by customers who want complete control over their environments, but they have the disadvantage that they require a physical machine and they consume rack space and power. Running many virtual machines on a large server offers the advantages of a dedicated machine together with the administrative conveniences of having everything running on a single machine.

4.3 Prototyping

Many sorts of services are more convenient to set up on a virtual machine, or a set of them, before rolling them out on physical machines. Network and clustering services are good examples of this. Setting up a virtual network is far more convenient that setting up a physical one. Once a virtual network is running, new services can be configured and tested on it. When the configuration is debugged, it can be copied to physical machines with confidence that it will work. If it doesn't, then it is likely that the hardware has been set up incorrectly.

So, prototyping first in a virtual environment allows software configuration problems to be separated from hardware configuration problems. If the software has been configured properly in a virtual environment and it doesn't work in a physical environment, there is a high probability that something is wrong with the physical configuration. So, time would not need to be wasted looking at the software; attention would be focussed on the hardware.

Another possibility is to use a virtual machine as a test environment to make sure that a service is working properly before running it on physical machines. This is especially attractive if there are a variety of environments that the new service will need to run in. They can all be tested in virtual machines without needing to dedicate a number of physical machines to testing.

4.4 Multiple environments without multiple machines

It is often convenient to have multiple environments at one's disposal. For example, it may not be obvious that a piece of software will work the same on different Linux distributions. The same is true of different versions of a distribution, library, or service. It might be necessary to test a piece of software in the various environments in which it is expected to run.

Normally, in order to check this, it is necessary to have multiple machines or a single multi-boot machine. However, this port makes it possible to run different environments in different virtual machines, allowing testing in those environments to happen on a single physical machine without rebooting it.

4.5 A Linux environment for other operating systems

When this port is made to run on another operating system, it implements a virtual Linux machine within that OS. As such, it represents an environment in which that OS can run Linux binaries.

A number of operating systems already have some amount of binary compatibility with Linux, and several have Linux compatibility environments. This is potentially another way of achieving the same goal. With Linux becoming the Unix development platform of choice, other Unixes are going to be looking for ways to run Linux executables, and this may be a good way of accomplishing that.

5 Future work

5.1 Protection of kernel memory

Since the kernel's memory is in each process address space, it is vulnerable to being changed by user space code. This is a security hole as well as making the entire virtual machine vulnerable to a badly written process. Kernel memory needs to be write-protected whenever process code is running, and write-enabled when the kernel is running. The one tricky aspect of this is that the code which write-enables kernel data will run on a kernel stack, which needs to be writable already. So, that stack page will be left writable when the process is running. It's not a problem if the process manages to modify it because it is fully initialized before any code starts running on it. Nothing depends on anything left behind on the stack.

5.2 Miscellaneous functionality

At this writing, the user-mode port is nearly fully functional. There's a major omission and some minor ones:

SMP Emulation This is the most serious missing functionality. This will allow a virtual machine to be configured with multiple virtual processors, regardless of how many physical processors are present on the host. This will allow SMP development and testing to happen on uniprocessor machines. It will also enable scaling experiments which test the scalability of the kernel on many more processors than are present on the host. This will possibly allow the kernel to get ahead of the available hardware in terms of processor scalability. SMP emulation is also a way of allocating CPU time to virtual machines. If one virtual machine is given one processor and another is given two, and they're both busy, the two-processor machine will get twice as much CPU time as the uniprocessor machine because it will have two processes running on the host versus the one being run by the uniprocessor machine.

`SA_SIGINFO` **support** The `SA_SIGINFO` flag allows processes to request extra information about signals that they've received. There is nothing difficult about supporting this option - it just hasn't been needed so far.

Privileged instruction emulation A number of i386 instructions require that the process executing them have special permission from the processor to run them. The most common examples are the `in` and `out` set of instructions and `sti` and `cli`. These make no sense to run as-is because there is no hardware for `in` and `out` to talk to, and because `sti` and `cli` are not how interrupts are enabled and masked. So, they will have to be emulated using the equivalent user-space mechanisms. Very few applications actually use these instructions, which is why this hasn't been implemented yet.

Virtual Ethernet enhancements The virtual Ethernet daemon can exchange packets with the host and directly with the host's Ethernet, but it also needs to be able to talk to its peers on other machines. This will allow a virtual Ethernet to span multiple physical machines without putting the Ethernet frames directly on the Ethernet, allowing a multi-machine virtual Ethernet to remain isolated from the physical Ethernet.

Nesting Currently, the user-mode kernel can not run itself. The inner kernel hangs after getting fifty or sixty system calls into init. To some extent, this is just a stupid kernel trick, but it does have some utility. The main one is testing. It is a demanding process. It exercises many features of the host kernel that aren't used very often. So if it can run itself correctly, that is a good indication that it is functional and correct. The other main use for nesting is testing itself on multiple distributions. There have been bugs and permission problems which show up one some distributions and not others. So, this would be a convenient way of making sure it works on many distributions.

5.3 Performance improvements

5.3.1 Eliminating the tracing thread

The tracing thread is a performance bottleneck in several ways. Every system call performed by a virtual machine involves switching from the process to the tracing thread and back twice, for a total of four context switches. Also, every signal received by a process causes a context switch to the tracing thread and back, even though the tracing thread doesn't care about the vast majority of signals, and just passes them along to the process.

The one thing that the tracing thread is absolutely needed for is intercepting system calls. The current plan for eliminating it involves creating a third system call path in the native kernel which allows processes to intercept their own system calls. This would allow a process to set a flag which requests a signal delivery on each system call that it makes. The signal handler would be invoked with that flag turned off. Once in the handler, the process would examine its own registers to determine the system call and its arguments, and call the appropriate system call function as it does currently. When it returns, it would write the return value into the appropriate field in its sigcontext structure, and return from the signal handler.

At that point, it would return back into user space with the correct system call return value, and, again, there would be no way to tell that anything strange had happened.

5.3.2 Block driver

The block driver is currently only able to have one outstanding I/O request at a time. This greatly hurts I/O intensive applications. Fixing this would involve either having more than one thread to do I/O operations or using an asynchronous I/O mechanism to allow multiple requests to be outstanding without needing to block.

5.3.3 Native kernel

In order to get the best performance with this port, some work will likely be needed in the host kernel. The need for system call interception without context switches has already been mentioned. In addition, access to the host memory context switching mechanism would probably speed up context switches greatly. The ability to construct and modify `mm_struct` objects from user-space and switch an address space between them would eliminate the potential address space scan from context switches.

Another area to look at is the double-caching of disk data. The host kernel and the user-mode kernel both implement buffer caches, which will contain a lot of the same data. This is obviously wasteful, and tuning the host to be the best possible platform will probably require that this be addressed somehow.

5.4 Ports

5.4.1 Linux ports

Currently, this port only runs on Linux/i386. There are Linux/ppc and Linux/ia64 ports underway, but not completed. The other architectures should also be supported. Ports pose no major problems since Linux hides most of the hardware from its processes. The main things that show through are register names. The system call dispatcher needs to extract the system call number and arguments from specific registers, and put the return value in a specific register afterwards. This is obviously machine-dependent. This is handled by using symbolic register names in the generic code and using a architecture-dependent header file to map those names to real machine registers.

The other major portability problem is the system call vector. Most system calls are common to all architectures, although the assignments to system call numbers are different. A minority of system calls are present on some architectures and not others. It is desirable to keep the bulk of the system call vector in generic code while allowing the underlying architecture to add in its own system calls. This is done by having the vector initialization include, via a macro, architecture-specific slots.

There are a few other machine-dependent details like how a `sigcontext_struct` is passed into signal handlers, how a faulting address is made available to the `SIGSEGV` handler, and conversion between `pt_regs` and `sigcontext_struct` structures. These are handled by a small number of architecture-dependent macros and functions.

5.4.2 Other OS ports

User-mode Linux can be ported to other operating systems that have the necessary functionality. The ability to virtualize Linux system calls by intercepting and nullifying them in the host kernel is essential. An OS that can't do that can't run this port. Also needed is the ability to map pages into an address space in multiple places. Without this, virtual memory emulation can't be done. If those two items are available on a particular OS, then this port can probably run on it.

It would be convenient to have the equivalent of `CLONE_FILES`, which allows processes to share a file descriptor table without sharing address spaces. This is used to open a file descriptor in one process and have it be available automatically in all the other processes in the kernel. An example of this is the file descriptors that are used by the block device driver. They are opened when the corresponding block device is opened, usually in the context of a `mount` process. After that, any process that performs operations on that filesystem is going to need to be able to access the file descriptor. Without `CLONE_FILES`, there will need to be some mechanism to keep track of what file descriptors are open in what processes.

Beyond those, this port makes heavy use of standard Unix interfaces. So ports to other Unixes will be significantly easier than ports to non-Unixes. However, those interfaces have equivalents on most other operating systems, so non-Unix ports are possible.

5.5 Access to physical hardware

One potential use of this port in kernel development that hasn't been explored yet is driver writing and debugging. This requires access to physical devices rather than the virtualized, simulated devices that are currently supported. There are two main types of access that would be required:

I/O memory access Linux currently supports, via the `io_remap` facility, mapping I/O memory into a process virtual address space. This would give a driver access to the device's memory and registers.

Interrupts Device interrupts need to be forwarded to the user-mode driver somehow.

The current plan for supporting this involves a stub driver in the native kernel which can probe for the device at boot time. This driver would recognize the device and provide some mechanism for the real user-mode driver to gain access to it, such as an entry in `/proc` or `/dev`. The user-mode driver would open that file and make requests of the stub driver with calls to `ioctl`. The file descriptor would also provide the mechanism to forward interrupts from the stub driver to the user-mode driver. The stub driver's interrupt routine would raise a `SIGIO` on the file descriptor. One potential problem with this would be if the device needed some kind of response to an interrupt from the driver before interrupts are re-enabled. This would preclude that response from coming from the user-mode driver because the process couldn't be run with hardware interrupts disabled. The stub driver would have to respond, which would require that code be put in the host kernel, and it couldn't be tested in user space.

6 Conclusion

The user-mode port of the Linux kernel represents an interesting and potentially significant addition to the kernel pool. It is the first virtual kernel running on Linux, except possibly for VMWare. As such, it places new demands on the host kernel, possibly resulting in new functionality, which may then be used by other applications. This has already happened. Until 2.2.15 and 2.3.22, ptrace on Linux/i386 was not able to modify system call numbers (and on other architectures, it still can't). The demands of this port prompted Linus and Alan to add that feature, at which point several other applications started using it. At least one of those applications was completely impossible beforehand.

Aside from this port, the only options for running virtual machines are VMWare and IBM VM. In both of those cases, potential applications maybe be ruled out for economic reasons or because VM doesn't run on the application platform. The availability of a free virtual machine running on Linux may open up new opportunities that were reserved for mainframes or which just didn't exist before.