User-mode Linux*

Jeff Dike
jdike@karaya.com

Abstract

User-mode Linux [2](UML) is the port of the Linux[1] kernel to Linux. It implements a Linux virtual machine running on a Linux host. Its hardware is virtual, being constructed from resources provided by the host. UML can run essentially any application that can run on the host.

The design and implementation of UML has been previously described[1]. This paper will describe the work that has taken place during the last year, including changes to what was described in the previous paper. This paper will also discuss new applications of UML involving integration of the virtual and host environments, along with other possibilties such as using UML as a clustering platform.

1 Overview

User-mode Linux is a port of the Linux kernel to Linux. A rationale for doing this can follow from viewing a computer running Linux to be a "hardware platform" in the sense of considering whether it is capable of running a full-featured operating system. From this point of view, the special capabilities that would normally be provided by the processor, such as privileged instructions, special registers, and special memory regions, are instead provided by the Linux system call interface. The question that is immediately raised is whether that interface provides everything needed to implement Linux.

UML proves that the answer is "yes". It is a full-fledged port of the Linux kernel. It occupies a location in the kernel source tree[2] alongside the other ports, it implements the same interface as the other ports, and the generic (architecture-independent) kernel can't tell that there's anything different about this one. UML makes almost no changes in the generic kernel, and the changes that it does make are invisible to the other ports.

1.1 Functional overview

UML implements a virtual Linux machine running (currently) on a Linux host. All of its devices are virtual, being constructed from software resources provided by the host. These include every type of device that would be expected to be present on a typical physical machine:

- Consoles and serial lines may be attached to a variety of devices on the host, including pseudo-terminals, virtual consoles, file descriptors, and xterms.
- Block devices can be associated with anything on the host that resembles a normal file, including files and device nodes such as CD-ROMs, floppies, disk partitions, and whole disks. These normally contain a filesystem image or a swap signature and are mounted or used as swap by the virtual machine, but they could contain arbitrary data, which could be read from the raw device with a program such as `dd`.
- Network devices can be attached to most types of software network interfaces on the host,

*This paper is to be published in the Proceedings of the 5th Annual Linux Showcase & Conference, Oakland CA

[1]Linux is a trademark of Linus Torvalds

[2]UML is currently in Alan Cox's kernel tree, having been merged in the middle of April 2001. I am planning on submitting it for inclusion into Linus' tree when it has reached a level of stability and functionality that warrants calling it version 1.0.

such as TUN/TAP, Ethertap, and slip devices. There are also two mechanisms for exchanging Ethernet frames directly between virtual machines without going through the host's networking system - one involving a central daemon acting as an Ethernet switch and the other using a multicast network.

UML runs the same binary executables as the host - normal userspace code runs natively on the processor just as it does on the host; there is no instruction emulation. It can run essentially everything that the host can. The few exceptions include applications such as emulators, which are hardware-dependent, and use privileged instructions which UML doesn't emulate, and a few other things, such as installation procedures, which expect to use specific devices which don't exist in UML.

1.2 Design and implementation overview

The other virtual machine technologies available on the PC platform, VMWare[5] and Plex86[4], create a platform (the emulated machine) which is authentic enough to boot an existing operating system. In contrast, UML takes Linux itself as the platform and modifies the Linux kernel to run on it. Since the usual method of making Linux run on a new platform is to port it, UML does exactly that, and ports the Linux kernel to its own system call interface.

The task of porting Linux to itself amounted to finding ways of virtualizing all of the required hardware capabilities in terms of Linux system calls. The most important of these is a distinction between a privileged kernel mode and an unprivileged user mode. A native kernel running on hardware must have a privileged mode in which only trusted code (i.e. the kernel) can run so that it can reliably and securely arbitrate access to the hardware. UML must have an equivalent distinction which allows the kernel to have access to the host's system calls while its processes must request that access by calling into the system. This distinction is implemented using the Linux system call interception mechanism provided by `ptrace`.

A special thread is used to intercept the system calls made by all the UML process threads. This tracing thread annuls these system calls in the host and redirects the processes into the kernel, where tracing is turned off. Thus, while in user mode, processes have their system calls intercepted and virtualized, in kernel mode, they are released from tracing and their system calls run directly in the host kernel. This is the exact analog to the privileged access to hardware enjoyed by the kernel on a physical machine.

Virtual memory posed the next most significant challenge. The Linux kernel expects the hardware to provide access to a pool of physical memory which may be allocated for kernel data structures or allocated for virtual memory and mapped arbitrarily in either a process or the kernel virtual memory area. This is done by creating a file on the host which is the size of the physical memory declared on the UML command line. This file is mapped as a contiguous block into a region of the UML process address space set aside as "physical memory". The pages of memory in this region are released to the kernel memory allocator which is then able to allocate them to whatever subsystems need it. When a page is mapped into virtual memory, the low-level mapping code maps that page from the underlying file into the appropriate spot in the virtual address space. Thus, each page is mapped into the physical memory region, and arbitrarily many times into process or kernel virtual address spaces.

With this mechanism in place, providing multiple, independent, mutually inaccessible address spaces is straightforward. Each UML process has a separate address space on the host, so a context switch from one process to another automatically causes an address space switch. However, this is complicated by the fact that a process address space may be modified while it's out of context. The system may swap out some of its memory. Therefore, when a process comes back into context, the state of its address space may need to be updated since it may still map pages which have been freed and reallocated for some other purpose. So, a scan of the process page tables occur on a context switch, during which any pages which have had their protections or mappings change are updated to reflect the current state of the address space.

Those were the most significant challenges involved in the port. With those solved, the rest of the port is comparatively simple, with the remaining required mechanisms having obvious implementations in terms of Linux system calls.

Hardware faults are implemented in terms of Linux system calls - I/O device interrupts are provided by `SIGIO`, page faults by `SIGSEGV`, and timer interrupts by `SIGALRM` and `SIGVTALRM`. A normal Linux signal handler layer receives all signals, determines what the cause is, and calls into the kernel appropriately, either by calling the IRQ system for device interrupts, the page fault handler for `SIGSEGV`, or the signal subsystem for signals, such as `SIGILL`, `SIGBUS`, and `SIGTRAP`, that are simply passed along to the process.

Delivering signals to processes is apparently simple, but there are a number of ways of fouling up the implementation. In general, this is done by constructing a special frame on the process stack which contains the process register state at the time of the signal, some other process context, and a procedure context which will cause the process to call its handler for that signal when it returns to userspace. UML makes the host kernel construct the signal frame by sending a signal to the process, which is handled by a UML handler. This handler then invokes the process signal handler. The cleanup after the handler returns is triggered by the interception of the host's `sigreturn`. The old process state is restored and the process returns to userspace at the point at which it received the signal.

2 Changes during the last year

2.1 The COW block driver

The major enhancement to the UML block driver has been the contribution by Greg Lonnon of a copy-on-write (COW) layering capability. This allows UML to layer a private writable file over a shared read-only file to form a single read-write block device. The writable file contains only the blocks that have been modified. This provides the ability for multiple UMLs to share a single filesystem image and to write to it. This is important for root filesystems, since they tend to be large, and Linux doesn't deal well with read-only roots.

When the block driver is using a COW device, it writes modified blocks to the COW layer, and reads from either the COW layer or the backing layer, depending on whether or not the requested block has been modified.

The COW file contains a header which contains the following

- a magic number to distinguish a COW file from a normal filesystem image
- a version number
- the path of the read-only backing file
- the last modification time and last size of the backing file, which are used to check that the backing file hasn't been modified
- the sector size used by this file

Following the header is a bitmap describing which blocks have been modified and are valid in the COW layer. This is loaded into memory and used to decide where blocks should be read from.

Following the modified block bitmap is the actual block data. This is sparse, meaning that valid blocks are located in the same location relative to the start of the filesystem as their equivalents in the backing file and that only those blocks which have been modified have been allocated disk space.

The sparseness of the COW file greatly simplifies the driver by allowing it to read and write the same locations relative to the filesystem start, regardless of whether the I/O is happening on the COW file or the backing file.

With many UMLs booting from the same filesystem through COW devices, the disk space required is greatly reduced. In the situation where a number of virtual machines are booted from the same filesystem and they have made few changes to the data, the disk space consumption for the entire group is not much more than that of a single UML. This reduction in disk space can greatly increase the number of virtual machines that the host machine can run. It can also increase the efficiency with which it can run them. Since the vast majority of the data used by the virtual machines is shared, only one copy of it will exist in the host's caches. This effectively increases the size of the host's memory, since, previously, different virtual machines would have entire private filesystems, which would be cached separately by the host despite the fact that they are largely identical.

It is also an administrative convenience. Creating a new COW file is done automatically by UML when it is requested on the command line. This is far more convenient than copying a large filesystem image, which can take several minutes for a large filesystem.

Another advantage is that it provides a simple checkpointing facility. If the virtual machine crashes or something important was deleted, and there was no important data in the COW file, then the old filesystem state can be restored by simply deleting it and starting over with another one.

It is sometimes desirable to be able to merge the COW file changes into the backing file. This is done with the `uml_moo` utility. `uml_moo` simply traverses both the COW file and backing file, writing the current version of each block out to a third file.

2.2 The mconsole

The management console (mconsole) driver is a new low-level interface into the kernel. It provides the following capabilities:

Halting and rebooting the kernel This allows the mconsole user to perform a kernel shutdown. The system is not shut down cleanly, so the filesystems are not unmounted cleanly and `init` has no opportunity to shut down the system's services.

Plugging and unplugging devices This is the ability to insert and remove devices from the running system, which is discussed in detail in section 2.5. Not all UML drivers currently support devices being inserted and removed - this is intended to be fixed in the medium-term future.

Providing access to the SysRq driver The mconsole has a `sysrq` command which allows the user to execute the commands provided by the SysRq driver in the generic kernel.

Enabling and disabling the kernel debugger The kernel debugger is considered to be a device, so it can be 'plugged' and 'unplugged' like the other devices. This is also described in more detail in section 2.5.

These capabilities are accessed through a client which sends requests to the mconsole driver and receives replies from it through a Unix domain socket. The existing client is a simple text-mode command-line client that can communicate with a single UML.

2.2.1 Future mconsole development

The current mconsole client is the simplest possible one - it has a command-line interface, controls a single UML instance, and implements only the commands supported by the mconsole driver.

Future mconsole clients will have a number of client-side enhancements:

- the ability to communicate with and control multiple UMLs
- different interfaces, such as GUIs and HTML, and possibly more specialized interfaces such as emacs, Linuxconf, and IRC
- built-in scripting or programming to allow automated management of UMLs

In addition, there are a number of mconsole protocol enhancements planned in order to provide clients with greater abilities to control and monitor virtual machines.

Notification of UML events The ability to register for events such as panics and console output and to retrieve configuration information from the virtual machine is intended to support applications which configure, launch, and manage many UMLs. Receiving panic information and console output will allow such an application to monitor the status of the virtual machines under its care and to provide a central point for the administrator to monitor them or for the application itself to filter the console output looking for unusual events.

Ability to retrieve configuration information The ability to get configuration information will simplify these applications by not requiring them to maintain its own record of how the virtual machines are configured. It also makes it possible to manage a virtual machine which it didn't configure or launch.

Communication over IP sockets Currently, the mconsole driver communicates through a Unix domain socket. Possibly, this will be extended to allow it to use a Internet domain socket as well. This would allow the mconsole client to transparently manage virtual machines regardless of whether or not they are running on the same host as the client.

The issue that makes this uncertain is authentication. There needs to be some security so that random users can't shut down or reconfigure virtual machines that don't belong to them. Currently, this is handled by passing a set of credentials over the Unix socket. Access is allowed if the user id and group id in the credentials match the UML's user and group ids. This is not possible across Internet sockets, so some other scheme would be necessary. What would probably be done is that a name of a file containing a secret would be passed in on the UML command line or configured later with a local mconsole client. A remote mconsole client would need to present this secret to the virtual machine before being allowed access to it.

However, the secret would not be protected from snooping without the connection being encrypted. So, this would probably involve ssh somehow, and there is already a perfectly good way of using ssh to access a UML's mconsole without adding any extra mechanisms to either UML or the client. That is to run the mconsole client locally with the virtual machine and to have the central administration application communicate with it over ssh. So, unless there turns out to be some good reason not to, my intention is to recommend this scheme for anyone wanting to communicate remotely with UML's mconsole driver.

2.3 hostfs

The host filesystem (hostfs) is a virtual UML filesystem which provides access to the host filesystem. Mounting a hostfs filesystem is done in the same way as any other virtual filesystem -

```
mount -t hostfs none /home/jdike -o
/home/jdike
```

mounts the host `/home/jdike` directory on the same location inside UML. The `-o` switch specifies the host directory to mount. If none is given, the default is the host's `/`.

There is currently no ability to restrict a UML's hostfs to a given directory on the host. This is needed if hostfs is to be secure against hostile UML users. With this in place, a user inside UML would only be able to mount host directories that are within the directory specified on the command line. Effectively, this would provide a hostfs equivalent of `chroot`.

Use of hostfs in an environment which is supposed to be secure against hostile UML root users is somewhat problematic. It can be done, but care needs to be taken. To start with, hostfs should be restricted as just described. If restricted to an appropriate hierarchy on the host, UML users will not be able to access files that don't belong to them.

There is still the possibility of using hostfs to launch a Denial Of Service (DOS) attack against the host's disk space. This can be countered by putting disk quotas on the user id of the virtual machine. This will stop any DOS attacks against the host as a whole, but leaves open the possibility of a attack against the quota itself, depriving other virtual machines which share that user id of the ability to use hostfs. This problem can be solved by providing each UML with its own uid. This may be acceptable for a hosting service which is providing virtual machines for paying customers, but is less workable for a public access UML service. In that situation, it likely would be best to not provide hostfs at all.

2.3.1 hostfs design and future work

hostfs is split into a kernelspace piece and a userspace piece. The kernel portion implements the VFS interface and converts it into calls to the userspace portion, which makes libc calls on the host. The interface between them provides a set of simple file access operations.

This interface enables a number of possibilities besides simple access to the host's file systems. External resources that can be made to look like files and filesystems can be plugged into the interface, with the result that UML processes can access them as files. For example, with a suitable implementation of this interface, a database on the host could be mounted inside UML as a filesystem, with individual records possibly being represented as files.

This would allow processes such as shell scripts to browse and manipulate a database without requiring any database programming. Obviously, this is not the way to write a real database application, but it may be useful for writing a simple database browser. It would also be possible to create a special directory hierarchy within this filesystem that allows database queries to be made and which would provide access to the results. For example, if a MySQL database is mounted on `/mysql`, the command

```
% cd /mysql/query/'select * from cars where color =
"red"'
```

would change the current directory to a location where `ls` would show the records that matched the query.

Another possible use of a mountable database is to mount it jointly with a normal host filesystem. If the database and the filesystem initially contain the same information, then the userspace layer of hostfs can keep them in sync with each other by updating both whenever something is written out. The benefit of this is that searches on the filesystem can be performed by querying the database rather than doing a full search of the host filesystem. The standard Linux utilities have no ability to perform specialized database queries like this, so this would require that applications which have a particular need for this have knowledge of hostfs added. A good example would be a mail reader which knows that the user's mail directory is a joint hostfs mount of a normal directory and a database. It could implement its search capability by making a query to the database rather than the equivalent of grepping the directory.

2.4 IO memory emulation

UML has a new IO memory interface. This allows a file on the host to be mapped into UML as a separate, named region of memory. A driver would request the region and then provide some appropriate userspace access to it.

The demo driver provided with UML allows a process to `mmap` the region into its own address space. The process can then read and write the file through memory accesses in its own address space.

The purpose of this interface is to allow UML to access simulated hardware. So, on the host, there would be a process emulating a device which might be treating the file as a set of device registers, watching for changes, interpreting them as commands, simulating those commands, and writing results back into the file for the UML driver to see.

2.5 Pluggable drivers

The management console provides the ability to insert and remove devices from a running virtual machine. For example, the command

```
(mconsole) config eth0=ethertap,tap0,,192.168.0.254
```

adds the `eth0` device to the virtual machine that the mconsole is attached to. New devices are specified to the mconsole in exactly the same way that they are specified on the command line. So, in both cases, the same code in the driver parses and interprets the specification, and creates the new device.

Similarly,

```
(mconsole) remove eth0
```

removes the `eth0` device from the system.

Only devices that don't already exist may be plugged in at run-time. An attempt to create an `eth0` device on a system that already had an `eth0` would fail. Also, a device that is to be removed must be "idle" according to the driver. The definition of "idle" necessarily differs from device to device. The network driver requires that the interface be down, while the block driver requires that the device not be open in any way, either by being mounted or by being directly opened by a process such as `dd`.

The kernel debugger is considered a "device" in this context and can be started and stopped at run-time in a similar manner. At the time of writing, these three devices are the ones that can be plugged and unplugged.

The remainder of the device drivers will have this capability added in the medium future. It is a goal for version 1.0 of UML to have everything pluggable that can be.

The obvious exceptions are the console and serial line drivers. There are also some less-obvious exceptions. Memory should at least be pluggable. It's fairly straightforward to add memory to a machine at runtime. Less obvious is how to remove it. If the new memory region contains kernel data when it is to be removed, then it can't be, since kernel memory is not movable or swappable. Changing this is a problem for the generic kernel. Possibly, the new memory could be declared as being reserved for only process data or a mechanism for moving kernel data structures could be introduced.

I/O memory regions will also be pluggable. These are far easier than regular memory regions. They are owned by a single driver, and would only need to be released by that driver in order to be unplugged.

Another possibility is pluggable processors. Linux already has support for adding and removing processors, so once UML has SMP support, this will come for free. So, the mconsole will use the existing code to allow plugging and unplugging processors.

2.6 UML/ppc

There is now a second Linux port of UML. Chris Emerson ported it to Linux/ppc. As of this writing, it is nearly completely functional, except for a bug in its signal delivery which prevents it from booting up to multi-user mode. However, as long as UML stays in single-user mode, pretty much everything else seems to work fine. UML/ppc is completely merged into the UML pool, so it's completely up-to-date as far as UML features are concerned.

This has special significance as the first architecture port of UML. It has provided the information needed to cleanly separate the architecture-dependent code from the architecture-independent code. There turns out to be not very much of it. Most of the differences are due to machine details which show through the ptrace interface. These include register names and the storage of system call numbers, arguments, and return values in the register set. There are also a variety of small differences between architectures which need to be accommodated:

- there are differences in the layout and allocation of sigcontext structures
- ELF executables have different requirements on register and stack initialization on different platforms
- process address space layout is different, requiring adjustments of where UML loads itself
- some architectures don't implement `PTRACE_GETREGS` and `PTRACE_SETREGS`, requiring UML on that platform to emulate them with a loop of `PTRACE_GETREG` or `PTRACE_SETREG`.

The information gathered from this effort has been used to write a guide to future architecture ports of UML[3], which should make them largely a matter of filling in the blanks.

3 Future work

3.1 Address space reorganization

Currently, UML locates its text, data, and physical and kernel virtual memory areas in the middle of its process' address spaces. This causes some virtual address space fragmentation which could break some applications which need large contiguous areas of virtual memory.

This will be fixed by relocating all of UML's data above `TASK_SIZE`, allowing the process to use all memory below that. The UML `TASK_SIZE` will be lower than the host `TASK_SIZE` by enough to fit UML and its data between them.

However, with the standard `TASK_SIZE` on the host, this will break one specialized application of UML - honeypots. For UML to work as a honeypot, exploits that work against physical machines also have to work against virtual machines. For stack smash attacks, this requires that the stack be in the same location in the address space as on the host. So, I'm planning on allowing the process stack to occupy the very top of the process address space, with UML just below that, and the process having a large contiguous virtual memory area below that. A cleaner solution, but one which requires that the host kernel be rebuilt, is to change the userspace/kernelspace virtual address space split from the usual 3G/1G to something like 3.5G/.5G. This would allow the UML `TASK_SIZE` to be `0xc0000000`, which would leave the process stacks in the same location as they are on a normally configured host.

3.2 Block driver improvements

As mentioned in the previous paper[1], the UML block driver currently can have only one outstanding I/O request at a time. This severely hurts the kernel's attempts to do readaheads in order to have data in memory when a process is going to need it.

The obvious fix to this is to use an asynchronous I/O (AIO) mechanism to have a larger number of outstanding requests. There is an AIO mechanism in the works, and I plan to change the block driver to use it.

In addition, there is also the possibility of doing I/O to the host with `mmap` instead of `read` and `write`. This opens up some interesting possibilities for doing low-overhead I/O. The files that the driver is accessing would be mapped into the UML address space. If it can be arranged that the I/O request data buffer is the correct mapped page in that region, then I/O from UML to the host is essentially free of overhead. As soon as the data is written into that buffer in the upper levels of the kernel's I/O system, the I/O is done. If the UML process is also doing mmapped I/O, then it is zero-copy from that process through UML and into the host kernel.

This would work equally well with COW and non-COW devices. The shared pages would be mapped into UML read-only. Reads would work normally. A write would cause an access fault, which would be trapped, and the fault handler would unmap the read-only page and map in its place the appropriate read-write page from the COW layer. The write would then proceed normally, with the new data going into the private writable file.

A further advantage of doing I/O this way is that the filesystem data that's shared by multiple virtual machines will not be copied once for each of them. The mapped file will occupy memory that's shared between the UMLs. This is a potentially large advantage over the current situation, where data that's used by multiple UMLs is copied separately into each one. Allowing them to share like this reduces their memory consumption on the host, and potentially increases the capacity of the physical machine to host virtual machines.

3.3 Operating system ports

In principal, UML could be ported to operating systems other than Linux. In practice, whether this is possible depends on whether the target OS provides some of the specialized capabilities, such as Linux system call interception, that UML requires.

Currently, there are no efforts underway to do any UML ports, although there has been sporadic interest in doing a Windows port. Windows appears to be capable of running UML, although its memory mapping capabilities are not as general as those of Linux. Windows processes can only map files on 64K boundaries, while Linux requires page granularity (4K). This can be worked around by mapping an entire 64K block of memory on each page fault. This requires that the entire 64K block be mapped from a single source and that it be contiguous in that source. This is the case for almost all processes, so this won't impose any major restrictions on what UML/Windows would be able to run.

There has also been some investigation of a FreeBSD port. FreeBSD is interesting because most of the specialized facilities of Linux that UML uses have almost exact analogs in FreeBSD. The exception is system call interception. FreeBSD does have the ability for a process to load a new system call vector, which would allow UML to use a new vector make its processes call back out to userspace whenever they execute a system call.

As far as I know, there has been no investigation into porting UML to the commercial Unixes or the major non-Unix operating systems.

3.4 Shared subsystems

Multiple virtual machines running on the same host are sharing hardware resources, memory in particular. This opens up opportunities for sharing and communication between them that are impossible for physical machines without fairly exotic hardware.

One possibility is for separate UMLs to share a filesystem. This would be implemented by configuring a UML so that it consists of the filesystem code, the block layer, the UML block driver, and whatever supporting code that these systems require. It would not contain a number of things which are normally considered essential for a kernel, such as process support and virtual memory.

It would have its own memory pool, which would be mapped, together with its code, into the address spaces of a number of UMLs. This would be seen by the virtual machines as something very similar to a kernel module that implements a filesystem. They would mount it normally, and accesses to that filesystem would be handled by this shared code.

All of the virtual machines would perceive this filesystem to be local, even though it's really not owned by any of them. This would be a high-performance alternative to filesystems such as NFS when a shared read-write filesystem is required.

The code in the shared subsystem would effectively be SMP (and it would need to be compiled as such) since multiple virtual machines could be executing the code and accessing its data simultaneously. Since the virtual machines could be UP, this arrangement would create a set of hybrid UP-SMP virtual machines.

Configuring the kernel into this minimal, very specialized, role would require changes in the generic kernel. Capabilities that are normally mandatory would become optional. The kernel startup would need to be changed so that it doesn't attempt to boot a system by running init. Instead, the startup code would be executed by one of the virtual machines using it, and it would return after doing the necessary setup of its own data.

Filesystems aren't the only subsystems that could be shared in this way. For some applications, a shared block device, with either no filesystem or a distributed filesystem, would be useful. Another possibility is the network subsystem. Sharing it would effectively create a shared network device which the virtual machines could use to communicate with the outside world.

These are obviously useful for virtual machines running on the same host. They offer performance advantages and more efficient use of the host resources. These advantages would also apply to physical machines, if this arrangement can be realized in actual hardware. Devirtualizing the shared filesystem would result in a multiported disk with its own memory that is mapped into the memory of each machine that has access to it. This memory contains the code and data described above, and the native kernels would treat it as a module in exactly the same way that UML would.

This physical disk would offer the same advantages to the physical machines as the shared virtual filesystem offers to UMLs. If this happens, it would create a new area of application for UML - prototyping new devices virtually in order to see what uses they may have, and if they turn out to be useful enough, implementing them physically in hardware.

3.4.1 Security considerations

For multiple virtual machines to share a subsystem which enforces some kind of security, they must also share a security domain. In particular, for a shared filesystem, the virtual machines must also share user ids and group ids. Since users of virtual machines will commonly have root access, a set of UMLs that share a read-write filesystem must be mutually trusting. Otherwise, a root user in one machine could destroy data in the shared filesystem that was written by one of the other virtual machines.

4 Applications

There are a variety of applications for virtual machines such as UML. The main ones include

- Kernel debugging and development
- Process debugging in situations where the kernel is doing something inexplicable - stepping through the relevant portion of the kernel can quickly uncover the problem
- Sandboxing - this includes jailing potentially hostile code and untrusted services, as well as safely running new versions of the kernel, new distributions, and new services
- Education - using virtual machines instead of physical ones greatly simplifies the logistics of running courses on operating system design, system administration, network administration, and clustering
- Virtual hosting - the applications in the hosting industry are obvious; customers get root

access to their own virtual machine without having to colocate a physical server

- A Linux environment for other operating systems - once UML is ported to other operating systems, it provides a completely authentic Linux environment

It has become apparent that UML can be configured in many more ways than just a single virtual machine running on a single host. The rest of this section discusses applications that take advantage of this flexibility.

4.1 UML clustering

It is not necessary for a virtual machine to be confined to running on a single host machine. Since UML physical memory is virtual from the perspective of the host, it can be mapped into and unmapped from the UML address space. This means that it's possible to partition the physical memory of a single UML instance across a number of hosts. Memory that's present on one host would be unmapped from the others. When a virtual processor accesses memory that's not present on its host, a new low-level fault handler would request the page from the host which currently owns it. It would be unmapped from that host, copied across the net, and mapped on the host that needs to access it.

This would create an SMP virtual machine instance running on multiple hosts with at least one virtual processor on each host. Since this virtual machine has access to the combined resources of the hosts, this is effectively a single system image (SSI) cluster.

This would be useful to experiment with and fun to play with, but it would be so slow that it would be unlikely to find any kind of production use. The reason is that some kernel data structures are accessed so often by all the processors on the system that this cluster would spend all of its time copying the pages containing this data from node to node. Some data structures, spinlocks in particular, are accessed in such a pathological way that this cluster would come to a halt whenever two processors on different hosts tried to access them simultaneously.

The root cause of these problems is that this is an extreme form of Non-Uniform Memory Access (NUMA). Normal NUMA machines have a number of nodes, each containing one or more processors, with their own local memory. This local memory can be accessed quickly by the processors in that node. Accesses to the local memory of a different node is comparatively very expensive. In addition, there is some global memory which is equally accessible by all the node. Access to global memory is slower than access to a node's local memory and faster than access to a different node's local memory.

This UML cluster has no global memory, only local memory. In addition, access to a different node's local memory is particularly slow. The performance problems of this type of cluster will at least be alleviated by NUMA support in the generic kernel. This will partition some of the kernel's data between the machine's nodes so that non-local accesses are infrequent, and the only inter-node accesses are from a slow, background load balancing process.

This existence of this type of cluster will effectively put NUMA hardware in the hands of a large number of people who otherwise would have no access whatsoever to it. I hope and expect that this will attract more people to the effort of adding good NUMA support to the kernel.

However, this sort of shared-memory cluster may be so extreme a type of NUMA that good support in Linux may not be enough to get good performance. In this case, some sort of RPC interfaces will be needed so that the nodes can cooperate without needing to fault entire pages back and forth. The pure shared memory cluster would make a good starting point for that effort. For all its performance problems, it would work, so the RPC could be added incrementally, with a working cluster available at each stage. Debugging and performance analysis would be possible at all points of this process.

This work would also largely be applicable to native kernels running on physical machines. So, this process would probably speed the development of a more traditional clustering system for Linux, where the nodes are physical machines rather than virtual ones. However, the genesis of this system as a virtual cluster would leave its mark. Once Linux has physical clustering, there would be no requirement that a cluster's nodes be physical machines. A cluster could consist of a combination of physical machines and virtual nodes.

This creates the possibility of personal clusters, where an individual could cluster a number of personal machines, such as desktop and laptop machines, with virtual machines running on larger central servers. This would provide access to the hardware of the personal machines and the CPU horsepower of the servers. The virtual nodes could have access to the servers' hardware, or they could have access to nothing but virtual devices. So, these clusters could be used for everything from a sysadmin cluster, which provides centralized access to all of the servers and their hardware, to a user cluster, which provides access to personal hardware and no access to server hardware.

4.2 UML as a userspace library

So far, UML as been used solely in a standalone configuration, as a traditional virtual machine. However, the fact that it is normal userspace code makes it possible to configure it as a library which would be linked into other applications. This would require some structural changes which are also needed for it to be used in the shared subsystem configuration described in section 3.4. The kernel's initialization code would be required to initialize its data and nothing else. It would not then exec init in order to boot a system. It would behave the same as any library initialization code - initialize the library and return to the application.

The most obvious implication of this is that user-level applications could link against the kernel and gain access to the kernel's facilities, such as memory management and allocation, threads, filesystems, and networking.

The kernel's memory allocation facilities include the slab allocator, which allocates uniform sized objects, and page allocator, which allocates memory in units of pages using a buddy system which provides defragmentation.

The thread facilities offer a very efficient scheduler and a full set of spinlock and semaphore primitives.

These are all well-tested, debugged, efficient, and scalable. For these reasons alone, they make attractive replacements for their libc equivalents. In particular, the ongoing scalability work that's intended to make Linux scale to high-end SMP machines will translate directly into allowing applications to link against the UML library and gain the same scalability to a similar number of threads.

There are also a number of facilities in the kernel which do not have any equivalent in libc. The filesystems supported by Linux can be viewed as hierarchical data stores. A filesystem stored on a ramdisk is a temporary data store which will go away when the process exits. In order to make the data persistent, it would simply be stored on a device backed by a file on the host.

The network subsystem provides a complete private TCP/IP stack and network interface. This would allow an application to be a full-fledged, if somewhat specialized, network node.

Combined, these facilities offer some interesting possibilities to an application. It could store some of its data in an internal filesystem and export it to the rest of the world via NFS or another remote filesystem. External processes could mount this filesystem and gain access to this internal data. If it's read-only, this would allow transparent monitoring of the application's internal state. If it's writable as well, external monitors could change that state.

This would be useful for managing the configuration of a complex server such as Apache. It could export its configuration to an external monitor, which could tweak it as needed, without needing to change the configuration files and have Apache restart or reread those files. A sophisticated monitor could keep track of the state of the machines running Apache and change limits so that it continues to run efficiently. For example, a server on a fully-loaded server could be limited to not accepting further requests until it has reduced its backlog.

Another possibility would be to store the configuration in a filesystem outside the application and have it import it via NFS. This would enable changing the configuration of a large number of instances of this application at once, from a central location.

A different sort of application of this capability is to have an interactive application export its user interface (UI) to the host as a filesystem. External processes could then examine and manipulate the UI, allowing them to do such things as

- customizing the application as it's launched, restoring the state of the UI at the time that it was shut down or making site- or user-specific modifications
- filling in forms and dialog boxes as they are

created, restoring the most recently entered values, default values for the user or site, or values taken from another application that the process is monitoring

Allowing a process to copy data from one application's UI to another would allow it to intelligently maintain session-wide context. For example, if the user visits a friend's home page and starts composing an email message, the UI monitor could figure out whose home page that was, look up that person's email address, and fill it in as the default destination in the email composition window.

5 Conclusion

Up to this point, UML has implemented a classical virtual machine. This involves creating strict boundaries between the virtual machine, the host and everything else running on the host, including other virtual machines. The future development of UML is, in large part, going to blur, or completely eliminate these boundaries.

In a small way, this has started to happen. The hostfs filesystem allows UML to access the host filesystem as though it were its own. The boundary between the host and virtual machine are going to blur further as hostfs is used to access other host resources, such as databases, processes, and services. Other host resources will be virtualized within UML as processes within the virtual machine, allowing them to be manipulated in the same way as UML processes.

The distinction between the virtual machine and host will become even more hazy when UML becomes a library and is linked into applications. Then, the running image is both a normal application and a virtual machine running a Linux kernel. It will act as an application when it's running its own code, and as an operating system when it is making use of the capabilities of the UML library.

Boundaries between applications will become hazy as they export their UI structure and other internal state as filesystems where processes on the host can monitor and manipulate them. They'll be able to do such things as copy data from one application to another and use information taken from one to determine how to manipulate the other. As this process becomes extensive, it will be increasingly unclear what application really "owns" a particular piece of data. The data will start acquiring a existence independent of the applications that used to have sole control over it.

Similarly, the implementation of UML clusters is going to eliminate the boundary between different hosts, as spreading a single virtual machine instance across multiple hosts will allow all of the resources of all the hosts to be accessed from within a single machine. Almost as a side-effect, this will provide a convenient platform for developing SSI clustering technologies for Linux as a whole, not just for UML. In addition, the genesis of this general clustering in virtual machines will mean that Linux clusters will be able to arbitrarily mix physical and virtual nodes, providing a new level of flexibility in clustering.

Taking all of these together, it is possible to see a future where kernel code and user-level code are arbitrarily mixed together, where a process may be running on a single machine or spread over several machines, which may be both physical and virtual. It will be possible for several processes to be transparently operating on and presenting to the user the same data. Breaking down boundaries always opens up new possibilities and the most interesting ones will be ones that no one had predicted beforehand.

References

[1] Dike, Jeff. A user-mode port of the Linux kernel. In *Proceedings of the 4^{th} Annual Linux Showcase & Conference, Atlanta,* page 63, Atlanta, GA, 2000. Usenix

[2] The User-mode Linux Kernel Home Page. `http://user-mode-linux.sourceforge.net`

[3] Porting UML to a new architecture. `http://user-mode-linux.sourceforge.net/arch-port.html`

[4] Plex86 White Paper. `http://www.plex86.org/cgi-bin/cvsweb.cgi/~checkout~/plex86/docs/txt/paper-19991129a.txt`

[5] VMWare Home Page. `http://www.vmware.com`