Proceedings of the Linux Symposium

Volume One

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Linux as a Hypervisor

An Update

Jeff Dike
Intel Corp.
jeffrey.g.dike@intel.com

Abstract

Virtual machines are a relatively new workload for Linux. As with other new types of applications, Linux support was somewhat lacking at first and improved over time.

This paper describes the evolution of hypervisor support within the Linux kernel, the specific capabilities which make a difference to virtual machines, and how they have improved over time. Some of these capabilities, such as `ptrace` are very specific to virtualization. Others, such as AIO and `O_DIRECT` support help applications other than virtual machines.

We describe areas where improvements have been made and are mature, where work is ongoing, and finally, where there are currently unsolved problems.

1 Introduction

Through its history, the Linux kernel has had increasing demands placed on it as it supported new applications and new workloads. A relatively new demand is to act as a hypervisor, as virtualization has become increasingly popular. In the past, there were many weaknesses in the ability of Linux to be a hypervisor. Today, there are noticeably fewer, but they still exist.

Not all virtualization technologies stress the capabilities of the kernel in new ways. There are those, such as qemu, which are instruction emulators. These don't stress the kernel capabilities—rather they are CPU-intensive and benefit from faster CPUs rather than more capable kernels. Others employ a customized hypervisor, which is often a modified Linux kernel. This will likely be a fine hypervisor, but that doesn't benefit the Linux kernel because the modifications aren't pushed into mainline.

User-mode Linux (UML) is the only prominent example of a virtualization technology which uses the capabilities of a stock Linux kernel. As such, UML has been the main impetus for improving the ability of Linux to be a hypervisor. A number of new capabilities have resulted in part from this, some of which have been merged and some of which haven't. Many of these capabilities have utility beyond virtualization, as they have also been pushed by people who are interested in applications that are unrelated to virtualization.

`ptrace` is the mechanism for virtualizing system calls, and is the core of UML's virtualization of the kernel. As such, some changes to `ptrace` have improved (and in one case, enabled) the ability to virtualize Linux.

Changes to the I/O system have also improved the ability of Linux to support guests. These were driven by applications other than virtualization, demonstrating that what's good for virtualization is often good for other workloads as well.

From a virtualization point of view, AIO and `O_DIRECT` allow a guest to do I/O as the host kernel does—straight to the disk, with no caching between its own cache and the device. In contrast, `MADV_REMOVE` allows a guest to do something which is very difficult for a physical machine, which is to implement hotplug memory, by releasing pages from the middle of a mapped file that's backing the guest's physical memory.

FUSE (Filesystems in Userspace), another recent addition, is also interesting, this time from a manageability standpoint. This allows a guest to export its filesystem to the host, where a host administrator can perform some guest management tasks without needing to log in to the guest.

There is a new effort to add a virtualization infrastructure to the kernel. A number of projects are contributing to this effort, including OpenVZ, vserver, UML, and others who are more interested in resource control than virtualization. This holds the promise of allowing guests to achieve near-native performance by allowing guest process system calls to execute on the host rather than be intercepted and virtualized by `ptrace`.

Finally, there are a few problem areas which are important to virtualization for which there are no immediate solutions. It would be convenient to be able to create and manage address spaces separately from processes. This is part of the UML SKAS host patch, but the mechanism implemented there won't be merged into mainline. The current virtualization infrastructure effort notwithstanding, system call interception will be needed for some time to come. So, system call interception will still be an area of concern. Ingo Molnar implemented a mechanism called VCPU which effectively allows a process to intercept its own system calls. This hasn't been looked at in any detail, so it's too early to see if this is a better way for virtual machines to do system call interception.

2 The past

2.1 `ptrace`

When UML was first introduced, Linux was incapable of acting as a hypervisor[1]. `ptrace` allows one process to intercept the system calls of another both at system call entry and exit. The tracing process can examine and modify the registers of the traced child. For example, `strace` simply examines the process registers in order to print the system call, its arguments, and return value. Other tools, UML included, modify the registers in order to change the system call arguments or return value. Initially, on i386, it was impossible to change the actual system call, as the system call number had already been saved before the tracing parent was notified of the system call. UML needed this in order to nullify system calls so that they would execute in such way as to cause no effects on the host. This was done by changing the system call to `getpid`. A patch to fix this was developed soon after UML's first release, and it was fairly quickly accepted by Linus.

While this was a problem on i386, architectures differ on their handling of attempts to change system call numbers. The other architectures to which UML has been ported (x86_64, s390, and ppc) all handled this correctly, and needed

[1]on i386, which was the only platform UML ran on at the time

no changes to their system call interception in order to run UML.

Once ptrace was capable of supporting UML, attention turned to its performance, as virtualized system calls are many times slower than non-virtualized ones. An intercepted system call involves the system call itself, plus four context switches—to the parent and back on both system call entry and exit. UML, and any other tool which nullifies and emulates system calls, has no need to intercept the system call exit. So, another ptrace patch, from Laurent Vivier, added PTRACE_SYSEMU, which causes only system call entry to notify the parent. There is no notification on system call exit. This reduces the context switching due to system call interception by 50%, with a corresponding performance improvement for benchmarks that execute a system call in a tight loop. There is also a noticeable performance increase for workloads that are not system call-intensive. For example, I have measured a ~3% improvement on a kernel build.

2.2 AIO and O_DIRECT

While these ptrace enhancements were driven solely by the needs of UML, most of the other enhancements to the kernel which make it more capable as a hypervisor were driven by other applications. This is the case of the I/O enhancements, AIO and O_DIRECT, which had been desired by database vendors for quite a while.

AIO (Asynchronous IO) is the ability to issue an I/O request without having to wait for it to finish. The familiar read and write interfaces are synchronous—the caller can use them to make one I/O request and has to wait until it finishes before it can make another request. The wait can be long if the I/O requires disk access, which hurts the performance of processes which could have issued more requests or done other work in the meantime

A virtual OS is one such process. The kernel typically issues many disk I/O requests at a time, for example, in order to perform readahead or to swap out unused memory. When these requests are performed sequentially, as with read and write, there is a large performance loss compared to issuing them simultaneously. For a long time, UML handled this problem by using a separate dedicated thread for I/O. This allowed UML to do other work while an I/O request was pending, but it didn't allow multiple outstanding I/O requests.

The AIO capabilities which were introduced in the 2.6 kernel series do allow this. On a 2.6 host, UML will issue many requests at once, making it act more like a native kernel.

A related capability is O_DIRECT I/O. This allows uncached I/O—the data isn't cached in the kernel's page cache. Unlike a cached write, which is considered finished when the data is stored in the page cache, an O_DIRECT write isn't completed until the data is on disk. Similarly, an O_DIRECT read brings the data in from disk, even if it is available in the page cache. The value of this is that it allows processes to control their own caching without the kernel performing duplicate caching on its own. For a virtual machine, which comes with its own caching system, this allows it to behave like a native kernel and avoid the memory consumption caused by buffered I/O.

2.3 MADV_REMOVE

Unlike AIO and O_DIRECT, which allow a virtual kernel to act like a native kernel, MADV_REMOVE allows it implement hotplug memory, which is very much more difficult for a physical machine. UML implements its physical memory by creating a file on the host of the appropriate size and mapping pages from it into its own address space and those of its processes. I have long wanted a way to be able to free dirty pages from this file to the host as though they were clean. This would allow a simple way to manage the host's memory by moving it between virtual machines.

Removing memory from a virtual machine is done by allocating pages within it and freeing those pages to the host. Conversely, adding memory is done by freeing previously allocated pages back to the virtual machine's VM system. However, if dirty pages can't be freed on the host, there is no benefit.

I implemented one mechanism for doing this some time ago. It was a new driver, `/dev/anon`, which was based on tmpfs. UML physical memory is formed by mapping this device, which has the semantics that when a page is no longer mapped, it is freed. With `/dev/anon`, in order to pull memory from a UML instance, it is allocated from the guest VM system and the corresponding `/dev/anon` pages are unmapped. Those pages are freed on the host, and another instance can have a similar amount of memory plugged in.

This driver was never seriously considered for submission to mainline because it was a fairly dirty kludge to the tmpfs driver and because it was never fully debugged. However, the need for something equivalent remained.

Late in 2005, Badari Pulavarty from IBM proposed an `madvise` extension to do something equivalent. His motivation was that some IBM database wanted better control over its memory consumption and needed to be able to poke holes in a tmpfs file that it mapped. This is exactly what UML needed, and Hugh Dickens, who was aware of my desire for this, pointed Badari in my direction. I implemented a memory hotplug driver for UML, and he used it in order to test and debug his implementation.

`MADV_REMOVE` is now in mainline, and at this writing, the UML memory hotplug driver is in -mm and will be included in 2.6.17.

3 Present

3.1 FUSE

FUSE[2] is an interesting new addition to the kernel. It allows a filesystem to be implemented by a userspace driver and mounted like any in-kernel filesystem. It implements a device, `/dev/fuse`, which the userspace driver opens and uses to communicate with the kernel side of FUSE. It also implements a filesystem, with methods that communicate with the driver. FUSE has been used to implement things like sshfs, which allows filesystem access to a remote system over ssh, and ftpfs, which allows an ftp server to be mounted and accessed as a filesystem.

UML uses FUSE to export its filesystem to the host. It does so by translating FUSE requests from the host into calls into its own VFS. There were some mismatches between the interface provided by FUSE and the interface expected by the UML kernel. The most serious was the inability of the `/dev/fuse` device to support asynchronous operation—it didn't support `O_ASYNC` or `O_NONBLOCK`. The UML kernel, like any OS kernel, is event-driven, and works most naturally when requests and other things that require attention generate interrupts. It must also be possible to tell when a particular interrupt source is empty. For a file, this means that when it is read, it returns `-EAGAIN`

[2]http://fuse.sourceforge.net/

instead of blocking when there is no input available. /dev/fuse didn't do either, so I implemented both O_ASYNC and O_NONBLOCK support and sent the patches to Miklos Szeredi, the FUSE maintainer.

The benefit of exporting a UML filesystem to the host using FUSE is that it allows a number of UML management tasks to be performed on the host without needing to log in to the UML instance. For example, it would allow the host administrator to reset a forgotten root password. In this case, root access to the UML instance would be difficult, and would likely require shutting the instance down to single-user mode.

By chrooting to the UML filesystem mount on the host, the host admin can also examine the state of the instance. Because of the chroot, system tools such as ps and top will see the UML /proc and /sys, and will display the state of the UML instance. Obviously, this only provides read access to this state. Attempting to kill a runaway UML process from within this chroot will only affect whatever host process has that process ID.

3.2 Kernel virtualization infrastructure

There has been a recent movement to introduce a fairly generic virtualization infrastructure into the kernel. Several things seemed to have happened at about the same time in order to make this happen. Two virtualization projects, Virtuozzo and vserver, which had long maintained their kernel changes outside the mainline kernel tree, expressed an interest in getting their work merged into mainline. There was also interest in related areas, such as workload migration and resource management.

This effort is headed in the direction of introducing namespaces for all global kernel data. The concept is the same as the current filesystem namespaces—processes are in the global namespace by default, but they can place themselves in a new namespace, at which point changes that they make to the filesystem aren't visible to processes outside the new namespace. The changes in question are changed mounts, not changed files—when a process in a new namespace changes a file, that's visible outside the namespace, but when it make a mount in its namespace, that's not visible outside. For filesystems, the situation is more complicated than that because there are rules for propagating new mounts between namespaces. However, for virtualization purposes, the simplest view of namespaces works—that changes within the namespace aren't visible outside it.

When[3] finished, it will be possible to create new instantiations of all of the kernel subsystems. At this point, virtualization approaches like OpenVZ and vserver will map pretty directly onto this infrastructure.

UML will be able to put this to good use, but in a different way. It will allow UML to have its process system calls run directly on the host, without needing to intercept and emulate them itself. UML will create an virtualized instance of a subsystem, and configure it as appropriate. At that point, UML process system calls which use that subsystem can run directly on the host and will behave the same as if it had been executed within UML.

For example, virtualizing time will be a matter of introducing a time namespace which contains an offset from the host time. Any process within this namespace will see a system time that's different from the host time by the amount of this offset. The offset is changed by settimeofday, which can now be an unprivileged operation since its effects are invisible outside the time namespace.

[3] or if—some subsystems will be difficult to virtualize

gettimeofday will take the host time and add the namespace offset, if any.

With the time namespace working, UML can take advantage of it by allowing gettimeofday to run directly on the host without being intercepted. settimeofday will still need to be intercepted because it will be a privileged operation within the UML instance. In order to allow it to run on the host, user and groups IDs will need to be virtualized as well.

UML will be able to use the virtualized subsystems as they become available, and not have to wait until the infrastructure is finished. To do this, another ptrace extension will be needed. It will be necessary to selectively intercept system calls, so a system call mask will be added. This mask will specify which system calls should continue to be intercepted and which should be allowed to execute on the host.

Since some system calls will sleep when they are executed on the host, the UML kernel will need to be notified. When a process sleeps in a system call, UML will need to schedule another process to run, just as it does when a system call sleeps inside UML. Conversely, when the host system call continues running, the UML will need to be notified so that it can mark the process as runnable within its own scheduler. So, another ptrace extension, asking for notification when a child voluntarily sleeps and when it wakes up again, will be needed. As a side-benefit, this will also provide notification to the UML kernel when a process sleeps because it needs a page of memory to be read in, either because that page hadn't been loaded yet or because it had been swapped out. This will allow UML to schedule another process, letting it do some work while the first process has its page fault handled.

3.3 `remap_file_pages`

When page faults are virtualized, they are fixed by calling either mmap or mprotect[4] on the host. In the case of mapping a new page, a new vm_area_struct (VMA) will be created on the host. Normally, a VMA describes a large number of contiguous pages, such as the process text or data regions, being mapped from a file into a region of a process virtual memory.

However, when page faults are virtualized, as with UML, each host VMA covers a single page, and a large UML process can have thousands of VMAs. This is a performance problem, which Ingo Molnar solved by allowing pages to be rearranged within a VMA. This is done by introducing a new system call, remap_file_pages, which enables pages to be mapped without creating a new VMA for each one. Instead, a single large mapping of the file is created, resulting in a single VMA on the host, and remap_file_pages is used to update the process page tables to change page mappings underneath the VMA.

Paolo Giarrusso has taken this patch and is making it more acceptable for merging into mainline. This is a challenging process, as the patch is intrusive into some sensitive areas of the VM system. However, the results should be worthwhile, as remap_file_pages produces noticeable performance improvements for UML, and other mmap-intensive applications, such as some databases.

4 Future

So far, I've talked about virtualization enhancements which either already exist or which show

[4]depending on whether the fault was caused by no page being present or the page being mapped with insufficient access for the faulting operation

some promise of existing in the near future. There are a couple of areas where there are problems with no attractive solutions or a solution that needs a good deal of work in order to be possibly mergeable.

4.1 AIO enhancements

4.1.1 Buffered AIO

Currently AIO is only possible in conjunction with `O_DIRECT`. This is where the greatest benefit from AIO is seen. However, there is demand for AIO on buffered data, which is stored in the kernel buffer cache. UML has several filesystems which store data in the host filesystem, and the ability for these filesystems to perform AIO would be welcome. There is a patch to implement this, but it hasn't been merged.

4.1.2 AIO on metadata

Virtual machines would prefer to sleep in the host kernel only when they choose to, and for operations which may sleep to be performed asynchronously and deliver an event of some sort when they complete. AIO accomplishes this nicely for file data. However, operations on file metadata, such as `stat`, can still sleep while the metadata is read from disk. So, the ability to perform `stat` asynchronously would be a nice small addition to the AIO subsystem.

4.1.3 AIO `mmap`

When reading and writing buffered data, it is possible to save memory by mapping the data and modifying the data in memory rather than using `read` and `write`. When mapping a file, there is no copying of the data into the process address space. Rather, the page of data in the kernel's page cache is mapped into the address space.

Against the memory savings, there is the cost of changing the process memory mappings, which can be considerable—comparable to copying a page of data. However, on systems where memory is tight, the option of using `mmap` for guest file I/O rather than `read` and `write` would be welcome.

Currently, there is no support for doing `mmap` asynchronously. It can be simulated (which UML does) by calling `mmap` (which returns after performing the map, but without reading any data into the new page), and then doing an AIO read into the page. When the read finishes, the data is known to be in memory and the page can be accessed with high confidence[5] that the access will not cause a page fault and sleep.

This works well, but real AIO `mmap` support would have the advantage that the cost of the `mmap` and TLB flush could be hidden. If the AIO completes while another process is in context, then the address space of the process requesting the I/O can be updated for free, as a TLB flush would not be necessary.

4.2 Address spaces

UML has a real need for the ability of one process to be able to change mappings within the address space of another. In SKAS (Separate Kernel Address Space) mode, where the UML kernel is in a separate address space from its processes, this is critical, as the UML kernel needs to be able to fix page faults, COW processes address spaces during `fork`, and empty process address spaces during `execve`. In

[5]there is a small chance that the page could be swapped out between the completion of the read and the subsequent access to the data

SKAS3 mode, with the host SKAS patch applied, this is done using a special device which creates address spaces and returns file descriptors that can be used to manipulate them. In SKAS0 mode, which requires no host patches, address space changes are performed by a bit of kernel code which is mapped into the process address space.

Neither of these solutions is satisfactory, nor are any of the alternatives that I know about.

4.2.1 `/proc/mm`

`/proc/mm` is the special device used in SKAS3 mode. When it is opened, it creates a new empty address space and returns a file descriptor referring to it. This address space remains in existence for as long as the file descriptor is open. On the last close, if it is not in use by a process, the address space is freed.

Mappings within a `/proc/mm` address space are changed by writing structures to the corresponding file descriptor. This structure is a tagged union with an arm each for `mmap`, `munmap`, and `mprotect`. In addition, there is a `ptrace` extension, `PTRACE_SWITCH_MM`, which causes the traced child to switch from one address space to another.

From a practical point of view, this has been a great success. It greatly improves UML performance, is widely used, and has been stable on i386 for a long time. However, from a conceptual point of view, it is fatally flawed. The practice of writing a structure to a file descriptor in order to accomplish something is merely an ioctl in disguise. If I had realized this at the time, I would have made it an ioctl. However, the requirement for a new ioctl is usually symptomatic of a design mistake. The use of `write` (or `ioctl`) is an abuse of the interface. It would have been better to implement three new system calls.

4.2.2 New system calls

My proposal, and that of Eric Biederman, who was also thinking about this problem, was to add three new system calls that would be the same as `mmap`, `munmap`, and `mprotect`, except that they would take an extra argument, a file descriptor, which would describe the address space to be operated upon, as shown in Figure 1

This new address space would be returned by a fourth new system call which takes no arguments and returns a file descriptor referring to the address space:

```
int new_mm(void);
```

Linus didn't like this idea, because he didn't want to introduce a bunch of new system calls which are identical to existing ones, except for a new argument. Instead he proposed a new system call which would run any other system call in the context of a different address space.

4.2.3 `mm_indirect`

This new system call is shown in Figure 2.

This would switch to the address space specified by the file descriptor and run the system call described by the second and third arguments.

Initially, I thought this was a fine idea, and I implemented it, but now I have a number of objections to it.

- It is unstructured—there is no type-checking on the system call arguments. This is generally considered undesirable in the system call interface as it makes it impossible for the compiler to detect many errors.

```
int fmmap(int address_space, void *start, size_t length,
          int prot, int flags, int fd, off_t offset);
int fmunmap(int addresss_space, void *start, size_t length);
int fmprotect(int address_space, const void *addr, size_t len,
              int prot);
```

Figure 1: Extended `mmap`, `munmap`, and `mprotect`

```
int mm_indirect(int fd, unsigned long syscall,
                unsigned long *args);
```

Figure 2: `mm_indirect`

- It is too general—it makes sense to invoke relatively few system calls under `mm_indirect`. For UML, I care only about `mmap`, `mprotect`, and `munmap`[6]. The other system calls for which this might make sense are those which take pointers into the process address space as either arguments or output values, but there is currently no demand for executing those in a different address space.
- It has strange corner cases—the implementation of `mm_indirect` has to be careful with address space reference counts. Several system calls change this reference count and `mm_indirect` would need to be aware of these. For example, both `exit` and `execve` dereference the current address space. `mm_indirect` has to take a reference on the new address space for the duration of the system call in order to prevent it disappearing. However, if the indirected system call is exit, it will never return, and that reference will never be dropped. This can be fixed, but the presence of behavior like this suggests that it is a bad idea. Also, the kernel stack could be attacked by nesting `mm_indirect`. The best way to deal with these problems is probably just to disallow running the problematic system calls under `mm_indirect`.
- There are odd implementation problems—for performance reasons, it is desirable not to do an address space switch to the new address space when it's not necessary, which it shouldn't be when changing mappings. However, `mmap` can sleep, and some systems (like SMP x86_64) get very upset when a process sleeps with `current->mm != current->active_mm`.

For these reasons, I now think that `mm_indirect` is really a bad idea.

These are all of the reasonable alternatives that I am aware of, and there are objections to all of them. So, with the exception of having these ideas aired, we have really made no progress on this front in the last few years.

4.3 VCPU

An idea which has independently come up several times is to do virtualization by introduc-

[6]and `modify_ldt` on i386 and x86_64

ing the idea of another context within a process. Currently, processes run in what would be called the privileged context. The idea is to add an unprivileged context which is entered using a new system call. The unprivileged context can't run system calls or receive signals. If it tries to execute a system call or a signal is delivered to it, then the original privileged context is resumed by the "enter unprivileged context" system call returning. The privileged context then decides how to handle the event before resuming the unprivileged context again.

In this scheme, the privileged context would be the UML kernel, and the unprivileged context would be a UML process. This idea has the promise of greatly reducing the overhead of address space switching and system call interception.

In 2004, Ingo Molnar implemented this, but didn't tell anyone until KS 2005. I haven't yet taken a good look at the patch, and it may turn out that it is unneeded given the virtualization infrastructure that is in progress.

5 Conclusion

In the past few years, Linux has greatly improved its ability to host virtual machines. The `ptrace` enhancements have been specifically aimed at virtualization. Other enhancements, such as the I/O changes, have broader application, and were pushed for reasons other than virtualization.

This progress notwithstanding, there are areas where virtualization support could improve. The kernel virtualization infrastructure project holds the promise of greatly reducing the overhead imposed on guests, but these are early days and it remains to be seen how this will play out. If, for some reason, it goes nowhere, Ingo Molnar's VCPU patch is still a possibility.

There are still some unresolved problems, notably manipulating remote address spaces. This aside, all major problems with Linux hosting virtual machines have at least proposed solutions, if they haven't yet been actually solved.