Making Linux Safe for Virtual Machines

Jeff Dike (jdike@karaya.com)

Abstract

User-mode Linux (UML)[1] is the port of Linux to Linux. It has demonstrated that the Linux system call interface is sufficiently powerful to virtualize itself. However, the virtualization isn't perfect, in terms of individual UML performance, aggregate performance of a number of UML instances sharing a host, or, in one way, functionality.

This paper discusses the current weaknesses in the ability of Linux to host virtual machines and proposes some ways of correcting those shortcomings.

1 Introduction

User-mode Linux (UML) is a port of the Linux kernel to the Linux system call interface. Since this had not been done before, it is impressive that, except for one trivial patch, Linux already provided the functionality needed to virtualize itself.

However, as development of UML has continued, some weaknesses in this support have become evident. Although UML has been successfully implemented using the existing system call interface, in some respects, it is highly non-optimal. This results in poor performance in some areas of UML and poor code in others.

The principal area which hurts UML performance is the Linux `ptrace` interface, which is used to virtualize system calls. Every kernel entry and kernel exit in UML requires two full context switches on the host. This makes system calls, in particular, far more expensive than on the host, but it also hurts the performance of interrupts, which also require four host context switches to complete.

For better context switching performance, each UML process has an associated host process. The host processes are created to gain access to their address spaces. The fact that threads also need to be created wastes host kernel memory and complicates UML context switching. To fix this problem, this paper proposes that Linux address spaces be made independent of threads and that they be created, populated, switched between, and destroyed as separate Linux objects.

Finally, there is the issue of maximizing the capacity of a given server to host virtual machines. Dealing with the issues described above will certainly help, but when considering the hosting throughput of a server, new problems arise. The largest one is memory management. The host and the virtual machines sharing it all have independent virtual memory systems which likely will have completely different pictures of how scarce memory is. Overall, this will lead to inefficient use of the host's memory because some virtual machines will perceive a memory shortage when there isn't one and free memory too aggressively. Similarly, others will not feel any memory pressure when there is some, and will consume memory that could more productively be used elsewhere. So, there will need to be new mechanisms for the host to communicate the true ex-

[1] http://user-mode-linux.sourceforge.net

tent of memory pressure to the UMLs that it's hosting and for the UMLs to respond appropriately to that information.

2 Debugging interface enhancements

2.1 Background

2.1.1 System call virtualization

UML runs the same executables as the host. Since those executables invoke system calls by trapping into the host kernel, UML needs some way to intercept and cancel them, and then execute them in its own context.

This is done through the `ptrace` mechanism. `ptrace` allows one process to intercept the system calls of another, letting it read them out and modify them. UML virtualizes the system calls of its processes by having a master thread, the tracing thread, use `ptrace` to trace the system calls of all UML processes. It annuls them in the host kernel by changing them into `getpid()`.

This works well, but it's slow, since each UML system call involves four context switches, to the tracing thread and back at the start of each system call and again at the end.

2.1.2 Kernel debugging

`ptrace` is exclusive in the sense that a given process can be traced by only one other process at a time. This is inconvenient for UML because the use of `ptrace` by the tracing thread precludes `gdb` from attaching to UML threads, making it harder to use as a UML debugger.

This was solved by having the tracing thread also trace `gdb`, intercepting its system calls, and manipulating the `ptrace` calls in order to fake `gdb` into believing that it's attached to UML.

This works well, but it's inconvenient, and has some unpleasant side-effects. Since `ptrace` reparents the process to whatever has attached to it, the original parent can change its behavior as a result. This generally isn't a problem, since UML usually runs `gdb` itself, or gdb is run under a shell or `emacs`, which don't notice the reparenting. However, a prominent exception is `ddd`, which spawns `gdb`, and calls `wait()` on it periodically. When UML attaches to that `gdb`, `wait` starts behaving differently for `ddd`, resulting in it not working at all.

2.2 A new debugging interface

To solve the performance problems of `ptrace`, UML needs a way for a process to intercept its own system calls. This could be done by delivering a signal whenever it performs a system call and having the signal handler nullify it in the host kernel and execute it inside UML.

Another possibility, which is more elegant, is to introduce a notion of two contexts within a single process. One context would trace the other, gaining control whenever it entered the kernel. This would be implemented by the debugging interface saving the master context state while the traced context runs. When the traced context makes a system call or receives a signal, the master context would be restored. It would have the state of the traced context available, and it could modify it however it saw fit.

This would reduce the cost of a UML system call from four host context switches to about two host system calls.

To allow `gdb` to debug UML, it would also be necessary to simultaneously allow other pro-

cesses to trace it. This doesn't pose any problems as long as their needs don't interfere with each other or with UML's own system call tracing.

These requirements are sufficiently different from what `ptrace` provides that a new interface is called for. David Howells of Red Hat is working on a `ptrace` replacement. It doesn't allow threads to trace themselves, but that looks like it can be added, and it satisfies all of the other requirements that UML has.

3 Address spaces

3.1 Background

3.1.1 UML address space switching

Currently, each UML process gets a process on the host. This is to provide each UML process with a separate host address space, which makes context switching faster. The alternative, all UML processes sharing a host address space, requires that that address be completely remapped on each context switch. So, if UML switches from a bash to an Apache, that address space would need to be changed from a bash address space to an Apache address space. In fact, this is how UML was first implemented. The change to giving each process a different host address space was done in order to avoid the overhead of walking the address space and remapping it on every context switch.

This optimization nearly converts a UML context switch into a host context switch. The exception is when a process has had its address space changed while it was not running. Most commonly, this is the result of the process being swapped. It can also happen when a threaded process forks. The thread that's out of context will have its address space COWed. When the process is next run, its address space needs to be updated to reflect these changes.

This is done with the help of two architecture-specific flag bits in the pte, `_PROT_NEWPAGE` and `_PROT_NEWPROT`. In a process that's not running, when a page is unmapped or a new page is mapped, `_PROT_NEWPAGE` is set in the page's pte. Similarly, when a page's protection is changed, `_PROT_NEWPROT` is set. When that process is next run, the page tables are scanned for these bits, and the appropriate action (`mmap`, `munmap`, or `mprotect`) is taken on the host in order to bring those pages up to date.

A second complication with bringing an address space up to date during a context switch is the kernel virtual memory area. The kernel, including its VM, is mapped into each process address space. When a process causes a change in the kernel virtual mappings, by adding or removing swap areas or by loading a module, those mappings need to be updated in each process that subsequently runs. The `_PROT_NEWPAGE` and `_PROT_NEWPROT` bits can't be used in this case because the kernel page tables, which store those mappings, are shared by all processes. So, there's no way for a single pte bit to indicate which processes are up to date and which aren't.

This problem is handled by using a counter which increments each time a change is made in the kernel VM area. Each UML process holds in its thread structure the value of that counter when it last went out of context. If the counter hasn't changed, then the process has an up to date kernel VM area. If it has, then it scans the kernel page tables in order to bring its address space up to date.

3.1.2 Execution context switching

The fact that every UML process has an associated host process has implications for switching execution contexts. The obvious way of doing this is for the outgoing process to do the following

1. send the incoming process `SIGCONT`
2. send itself `SIGSTOP`

However, this is unfixably racy. To see this, consider this sequence of events (process A is the outgoing process, and B is the incoming process).

1. A sends B `SIGCONT`
2. B is now runnable on the host, so the host scheduler switches to it, putting A to sleep temporarily
3. B finishes its work before its UML quantum expires, so it switches back to A
4. A now runs on the host and finishes its original UML context switch by sending itself `SIGSTOP`

Now, all UML processes are asleep and will never wake up, effectively hanging the machine.

This was the first implementation of UML's context switching. When this race was discovered, it was eliminated by having the tracing thread mediate context switches. The outgoing process would send a message to the tracing thread asking it to start the incoming process. The tracing thread would do that, leaving the outgoing process stopped. Using the tracing thread as a synchronization point eliminated the race.

In order to eliminate the role of the tracing thread in context switching, two further designs were tried. The first avoided the race by the outgoing process stopping itself with a signal, but blocking that signal and using `sigsuspend` to atomically sleep and enable it:

1. A blocks `SIGTERM`
2. A sends B `SIGTERM`
3. A calls sigsuspend, simultaneously sleeping and enabling `SIGTERM`

In this case, if B runs and switches back to A before it sleeps, then the `SIGTERM` won't be delivered until the `sigsuspend` call, avoiding the race.

However, implementing this involved some non-obvious signal manipulation, so a simpler method was implemented, and it is the current context switching mechanism.

Now, each UML process creates a pipe which is used by other processes to bring it into context. A context switch now works as follows:

1. A writes a character to B's pipe
2. B, which has been blocked in a read on that pipe, returns and continues running
3. A calls read on its own pipe

This avoids races in a similar, but simpler, way to the `SIGTERM` design.

3.2 The solution

So, while assigning a host process to each UML process provides reasonable context switching performance, it has a number of problems of its own:

- Changes to address spaces of other processes can't be effective immediately because one process can't change the address space of another. So, whenever a process is switched in, it must bring its address space up to date if necessary.
- Context switching is complicated by the need to avoid races when one host process continues another and stops itself

The proposed solution is to allow Linux address spaces to be created, manipulated, and destroyed just as processes are. In effect, this would turn address spaces into objects in their own right, separating them from threads. The capabilities that are needed are

- Creation of a new address space
- Changing the mappings in an arbitrary address space
- Switching between address spaces
- Destruction of address spaces

The implementation of this is fairly straightforward. Address spaces are already represented by a separate structure within the kernel, the `mm_struct`. Access to an `mm_struct` can be provided by a new driver which provides userspace access to it through a file descriptor. So, a handle to a process' address space may be obtained by opening `/proc/`*`pid`*`/mm` and a new, empty address space may be created by opening `/proc/mm`. Creating a new handle to an `mm_struct` would increment its reference count, and closing it would decrement it. So, an address space would not disappear as long as there are processes running in it or there are processes which have handles to it.

Allowing a process to change the mappings in another address space would be done with an extension to `mmap`:

```
void *mmap2(void *start,
size_t length, int
prot, int flags, int
src_fd, off_t offset,
int dest_fd);
```

The new argument, `dest_fd`, specifies the address space within which the new mapping is to take place. A value of -1 would specify the current address space, making this interface a superset of the existing `mmap`.

`munmap` and `mprotect` would need to be similarly extended.

3.3 Moving the UML kernel to a separate address space

With the ability to arbitrarily create new address spaces, and the debugging interface described in section 2.2, it is possible to move the UML kernel out of its process address spaces. What's needed is for the debugging interface to switch to the kernel address space when it restores the tracing context.

This would hurt performance somewhat by adding a memory switch to each kernel entry and exit, but would have some large compensating advantages.

The primary gain from doing this would be that it would make UML's data completely inaccessible to its processes. Currently, UML text and data occupy the top .5G of its process' address spaces. By default, this memory is not write-protected when userspace code is running.

This is a problem for applications of UML such as jails and honeypots that need to confine a hostile root user. There is a jail mode

in UML which write-protects kernel memory while userspace code is executing by using mprotect to enable write permission on kernel entry and to disable it on kernel exit.

However, there is a severe performance penalty doing this. Implementing jail mode by locating the kernel in a different address space would replace the calls to mprotect with two memory context switches. This is likely to be much faster, and if it's enough faster, it could become the default for UML.

4 AIO

5 Memory management primitives

6 Cooperative memory management between host and guest

6.1 Background

The changes discussed elsewhere in this paper are focussed on improving the performance of an individual UML instance. However, in some applications, such as virtual hosting, the aggregate performance of UML is of equal or greater importance. The aggregate performance is the performance of a set of UML instances running on the same host. Improving this at the expense of the individual UMLs can improve the economics of a virtual hosting installation if the capacity of the host improves enough to increase its overall throughput.

The consumption and use of host memory by UML is a crucial aspect of the server's hosting capacity. There are currently a number of aspects of Linux and UML which cause it to use more host memory than is necessary and to use it less efficiently than it should.

6.1.1 Unused memory is wasted memory

A basic assumption of the Linux VM system is that memory should be used for something, and if memory is plentiful, it doesn't matter what the excess is used for, because it might prove useful in the future. So, data is not thrown out of memory until there is a shortage.

This is fine for a physical machine which contains memory which can't possibly be used by anything else, but this policy hurts the UML hosting capacity of Linux. UML inherits this from the generic, architecture-independent kernel and thus won't free memory until it is feeling a shortage.

The problem is that the host may be short of memory without any of the UMLs it's hosting being short. So, they will hang on to their own data even though they could increase the host's performance by giving up some of it. If the host is swapping enough, they could even improve their own performance by giving up enough of their data to stop the host from needing to swap.

6.1.2 UML memory isn't shared

UML memory is instantiated by creating a temporary file on the host and mapping that into the UML address space. When some of this memory is used for file data, the UML block driver requests, via the `read()` system call, that the data be copied from the host's page cache into its own page cache. There are now two copies of that data in the host's memory. If ten UMLs each boot from separate copies of the same filesystem and copy the data into their own page caches, there will be twenty copies of that data on the host, ten in the host page cache because it loaded ten identical filesystems, and one for each UML. Clearly, this is a waste of memory.

This waste can be alleviated by having them boot from the same filesystem image with separate, private COW files. This will reduce the number of copies of shared data from twenty to eleven. Since they are sharing the same underlying filesystem, the number of copies in the host page cache is now one. However, there is still one copy per UML. This is still a waste of memory.

The problem is that there is currently no mechanism for reducing this any further. Clearly, the copy count could be reduced to one by having UML map data directly from the host page cache into its own page cache.

This would require a different I/O model in the generic kernel. Currently, Linux considers that it has a fixed amount of memory available to is, and when it reads data from disk, it has to allocate memory for that data and copy it from the disk. A UML instance mapping file data directly from the host memory is akin to having that memory, with the data already in it, materialize from nowhere.

6.1.3 VM information isn't shared

A further problem is that the host and the UML instances running on it all have completely independent VM systems which likely will have completely different ideas of how scarce memory is at any given moment. This is a problem because they are all sharing the same memory, and there is only one true picture of how scarce it is, and it is held in the host's VM system.

Somehow, this information needs to be communicated to the UMLs in a way that they can respond to. There are a number of requirements that need to be met if this is to happen:

- UML instances need to be able to free memory to the host. This can be done by unmapping unneeded memory or by calling `madvise(..., MADV_DONTNEED)`. The basic mechanisms are available on the host, but the generic kernel provides no clean way of using them.

- The host needs to be able to demand that a UML free a certain amount of memory of a certain type, i.e. dirty or clean pages, in a given time. This is needed in order for UML instances to feel memory pressure when the host does and to respond appropriately to it.

- UML instances need to be able to tell the host that memory which appears dirty, because it has been changed and not written to its backing store, is really clean, because it is in sync with its backing store from the UML point of view. This would happen when the UML instance had itself swapped the data to its own swap area. In this case, the host could treat the memory as clean and reallocate it without swapping it out.

6.2 Improving memory management cooperation

In contrast to the other problems identified in this paper, this is not a single problem that has a single fix as much as it is a set of problems which will require a set of improvements. This will likely require thought and work for some time to come in order to develop solutions that work reasonably well.

So, I will outline a set of possible partial solutions rather than the single fixes that have been described so far. Some of these may turn out to be of limited value, while other ideas, not described here, may be the ones that solve major parts of the problem.

6.2.1 Passing memory pressure from the host to UML

If there is to be any cooperation at all between the host and its UML instances in memory management, the host needs to be able to communicate memory pressure and its severity to UML. There are currently no mechanisms at all for doing this in Linux.

The host would need to be able to communicate the following information to the UML instances that it is hosting:

- the existence of memory pressure
- the amount of memory that a given instance should release
- the type of memory, i.e. dirty or clean pages, that should be released
- a deadline

The amount of memory that the host asks for may be more a guess than a calculation. It may be more useful for the host to decide what it will do to a UML instance if it doesn't get enough memory released from all sources. This would likely be some number of pages of that UML's memory that will be swapped out. Then, the UML instance can decide what it will do in order to try to avoid that fate. It would likely have a better idea of what memory it can do without than the host, so it could release pages that it thinks it needs less than what the host would choose to swap out.

Whether the host wants clean or dirty pages released depends on whether the host is also I/O-bound and on the timeframe within which it wants the memory. If it is I/O-bound, then freeing dirty pages isn't going to be useful because they would need to be written to swap before they could be freed, adding to the I/O congestion. Similarly, if the need for memory is immediate, then releasing dirty pages won't help because there will be a significant time lag between their release by UML and their availability to the system.

A UML instance will have some pages which it considers to be clean, but which the host considers to be dirty. A mechanism to inform the host that these should be considered clean would increase the proportion of clean pages available for release.

Another possible mechanism for reclaiming memory is for the host to just take clean pages from a UML without it explicitly releasing them. This would require that the host signal the UML instance when it next accesses such a page. It would have to refill the page with its original data before it could continue to be reused. This has the disadvantage that the UML has no choice in what pages are taken, so there is no guarantee that the host will choose pages that the UML instance can do without. The host could make reasonable guesses by looking at the hardware access bit, but it will still not have information about access patterns within the UML that may be relevant.

6.2.2 Freeing memory to the host

Currently, a UML instance is assigned a certain amount of memory which is considered to be its physical memory. It is not allocated on the host immediately. Rather, the host allocates it as it is used. However, once it is used, it is never given back. So an instance with a large amount of memory that has not been used recently will hold onto it even if there are other instances which have a much greater need for it.

As described above, there are mechanisms for a process to free memory back to the host. How-

ever, there is no way in the generic Linux kernel to give up memory. With some hooks in the page allocation and page release code, it is possible for the architecture to do some things.

Given a hook in the page release path, the architecture could release a page to the host by unmapping it or calling `madvise` appropriately. It also has a choice between freeing it to the page allocator or not. This decision would be based on whether the UML instance is to reduce the amount of memory it has at its disposal. If the page is made available to the page allocator, it will very likely be reallocated and reused. Thus, the host will need to reallocate the page soon after having it freed. This will limit the benefits to the host of freeing the page in the first place.

So, there would appear to be benefits to not freeing pages to the page allocator if the host is under memory pressure. However, if this is done, there would need to be some way of getting those pages back when the memory pressure on the host has abated.

Given a mechanism of the sort described in section 6.2.1, there should also be an obvious way of indicating that the memory pressure has diminished, and that the UML instance can start reclaiming the memory that it gave up. In both cases, there would need to be some indication from the host of how much memory should be given up and how much can be reclaimed. This would prevent undershooting and overshooting in both cases and make it more likely that the host will have a reasonable amount of free memory.

7 Conclusion

Fixing the problems described above will greatly increase both the performance of individual UML instances and the hosting capacity of a given server.

A more efficient system call interception interface will greatly increase the performance of system-intensive applications running inside UML. Compute-bound processes typically run as fast inside UML as on the host, but other, more system-intensive workloads can run two to three times slower. Allowing a thread to intercept its own system calls would bring the performance of these processes much closer to their performance on the host.

Similarly, using AIO to allow many outstanding I/O requests would help I/O-intensive workloads. Without AIO, UML is limited to one outstanding I/O request at a time. As a result, I/O-intensive processes can spend much of their time idle, waiting for their data to be read from the host. This isn't a waste of the host's processing power like the use of `ptrace` is, so it may not hurt the host's capacity, but it does noticeably hurt the performance of individual UML instances.

Allowing host address spaces to be manipulated as objects separate from threads would help UML's context switching performance. It would also greatly simplify the low-level context-switching code. There would be no need to traverse the address spaces of processes coming into context in order to bring them up to date with whatever changes were made while they were out of context. There would also be no need for the optimizations that have been made in order to make the existing algorithms faster. A further benefit to the code would be that the execution context switch would be completely race-free, since it would no longer be switching between host processes. This code has been significantly simplified over time, but it would become trivial once a single thread could switch between address spaces arbitrarily.

This enhancement, in conjunction with a single-thread system call interception capabil-

ity, would also allow the UML kernel to be located in a completely different address space than its processes. This would be particularly beneficial to jailing applications, which currently suffer from the poor performance of the current mechanism of protecting UML kernel data from userspace.

Finally, by improving the ability of the host to communicate memory pressure to the UML instances running on it and improving their ability to respond would noticeably improve the hosting capacity of a given server. In some cases, this would also improve the individual performance of the UML instances.

This area is also interesting because this is of much more general use than the others. `ptrace`, AIO, and address spaces are of fairly limited use to most applications. In contrast, memory management is of concern to practically all processes. So, implementing methods of cooperative memory management between the host and UML instances would provide those mechanism to other processes as well. This would open the way to this sort of cooperation being common, presumably with the result that the system performs better than it would otherwise.

This relatively small list summarizes the problems that Linux has as a virtual machine hosting environment. They mostly appear to have fairly straight-forward fixes, and some of these fixes are in progress already. The most complicated area is the cooperative memory management. That is a set of related problems that will require a set of measure to deal with them, rather than a single problem with a single fix. In contrast to the others, it will likely be the subject of study and work for some time to come.

Linux is currently quite viable as a virtual machine platform for a number of applications. Once these problems are fixed, Linux will become even more attractive for hosting virtual machines.

Proceedings of the
Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Wild Open Source, Inc.*