UML and the Intel VT extensions

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

Abstract

Intel has added virtualization extensions (VT) to the x86 architecture. It adds a new set of rings, guest rings 0 through 3, to the traditional rings, which are now called the host rings.

User-mode Linux (UML) is in the process of being enhanced to make use of these extensions for greater performance. It will run in guest ring 0, gaining the ability to directly receive software interrupts. This will allow it to handle process system calls without needing assistance from the host kernel, which will let UML handle system calls at hardware speed.

In spite of running in a ring 0, UML will appear to remain in userspace, making system calls to the host kernel and receiving signals from it. So, it will retain its current manageability, while getting a performance boost from its use of the hardware.

1 Introduction

Intel's new Vanderpool Technology[1] (VT) adds virtualization extensions to the IA architecture which enable hardware support for virtual machines. A full set of "guest" rings are added to the current rings, which are now called "host" rings. The guest OS will run in the guest ring 0 without perceiving any difference from running in the host rings 0 (or on a non-VT system). The guest is controlled by the host regaining control whenever one of a set of events happens within the guest.

The architecture is fully virtualized within the guest rings, so the guest can be an unmodified OS. However, there is also support for paravirtualization in the form of a *VMCALL* instruction which may be executed by the guest, which transers control to the host OS or hypervisor.

The hypervisor has fine-grained control over when the guest traps out to it (a *VMEXIT* event to the host) and over the state of the guest when it is restarted. The hypervisor can cause the guest to be re-entered at an arbitrary point, with arbitrary state.

The paravirtualization support is key to supporting environments other than unmodified kernels. User-mode Linux (UML) is one such environment. It is a userspace port of the Linux kernel, and, as such, would be considered a "modified" guest. It is heavily paravirtualized, as it contains a complete reimplementation, in terms of Linux system calls, of the architecture-specific layer of the kernel.

The reason to consider making UML use this support, when it is not obvious that it is useful, is that there are performance benefits to be

[1] AMD subsequently introduced a compatible technology code-named Pacifica.

realized by doing so. A sore spot in UML performance is its system call speed. Currently, UML must rely on ptrace in order to intercept and handle its process system calls. The context switching between the UML process and the UML kernel and the host kernel entries and exits when the process executes a system call imposes an order of magnitude greater overhead than a system call executing directly on the host. As will be described later, the VT architecture allows a guest to receive software interrupts directly, without involving the host kernel or hypervisor. This will allow UML/VT to handle process system calls at hardware speed.

2 Overview of UML/VT support

The VT paravirtualization support can be used to allow UML to run in a guest ring. For various reasons that will be discussed later, UML will be made to run as a real kernel, in guest ring 0. This would seem to contradict the "user-mode" part of UML's name, but as we shall see, the basic character of UML will remain the same, and the fact that it's running in a ring 0 can be considered an implementation detail.

The essential characteristics of UML are

- It makes system calls to the host kernel.
- It receives signals from the host kernel.
- It resides in a normal, swappable, process address space.

We are going to preserve the first by using the VT paravirtualization support to make system calls to the host kernel from the guest ring 0. Signals from the host will be injected into the guest by the host manipulating the guest state appropriately, and VMENTERing the guest. The third will be preserved as a side-effect of the rest of the design. UML/VT will start in a process address space, and the host will see page faults in the form of VMEXITs whenever the guest causes an access violation. Thus, the normal page fault mechanism will be used to populate the UML/VT kernel address space, and the normal swapping mechanism can be used to swap it out if necessary.

The fact that UML will be running in kernel mode means that it can't make system calls in the normal way, by calling the glibc system call wrappers, which execute *int 0x80* or *sysenter* instructions. Since we can't use glibc for system calls any more, we must implement our own system call layer in terms of *VMCALL*. glibc is UMLs interface to the host Linux kernel, so replacing that with a different interface to the underlying OS can be considered a port of UML to a different OS. Another way of looking at it is to observe that UML will now be a true kernel, in the sense of running in ring 0, and must be ported to that environment, making this a kernel-mode port of UML.

There must be something in the host kernel to receive those VMCALLs, interpret them as system calls, and invoke the normal system call mechanism. A *VMCALL* instruction invokes the VMEXIT handler in the host kernel, as does any event which causes a trap out of the guest to the host. The VMEXIT handler will see all such events, be they hardware interrupts, processor exceptions caused by the guest, or an explicit VMCALL.

3 Porting UML to VT

The first step in porting UML to VT is to make UML itself portable between host operating systems. To date, UML has run only on Linux, so it is strongly tied to the Linux system call interface. To fix this, we must first abstract out the Linux-specific code and put it under an interface which is somewhat OS-independent. Total OS-independence is not possible with only two examples which are very similar to each other, and is a more of a process than a goal in any case. What we are aiming for is an interface which supports both Linux and VT, and can be made to support other operating systems with modest changes.

To this end, we are moving all of the Linux-specific code to its own directory within the UML architecture (arch/um/os-Linux) and exposing a somewhat OS-independent interface to it. This task is simplified to some extent by the fact that glibc-dependent code had to be separated from kernel-dependent code anyway. The reason is that the former needs to include glibc headers and the latter needs to include kernel headers. The two sets of headers are very incompatible with each other—including both glibc and kernel headers into the same file will produce something that has no chance of compiling. So, from the beginning, UML has been structured such that glibc code and kernel code have been in separate files.

So, to some extent, this part of the port has involved simply moving those files from the main UML source, where they are intermingled with kernel source files, to the os-Linux directory. There are functions which are neither glibc- or kernel-dependent, so these need to be recognized and moved to a kernel file.

Once this code movement has happened, and the resulting interface has been cleaned up and minimized, the next step is to actually implement the interface in terms of VT, using *VMCALL*. So, we will create a new directory, possibly arch/um/os-vt, and implement this interface there. To actually build a VT-enabled UML, we will need to tell the kernel build process (kbuild) to use the os-vt directory rather than the os-Linux one. This is currently determined at runtime by setting a make variable to the output of `uname -s`, and forming the OS directory from that. We can override this variable on the command line by adding `OS=vt` to it, forcing kbuild to use the OS interface implementation in os-vt rather than os-Linux.

4 Host kernel support

As previously mentioned, there will need to be support added to the host kernel in order for it to run UML as a VT guest. Linux currently has no real support for being a hypervisor, and this is what is needed for this project.

The host kernel will need to do the following new things:

- Handle VMEXITs caused by the guest explicitly executing *VMCALL* instructions in order to make system calls.
- Handle hardware interrupts that happen while the guest is running, but which the guest doesn't need to deal with.
- Handle processor faults caused by the guest.
- Force the guest to handle whatever signals it receives from elsewhere on the host.
- Launch the guest and handle its exit.

The design for this calls for a kernel thread in the host to be created when a UML/VT instance is launched. This thread will do the VT-specific work in order to create the guest context and to start UML within it.

Once the UML instance is launched and running, this thread will become the VMEXIT handler for the instance. It will be invoked whenever the CPU transfers control from the guest to the host for any of a number of reasons.

VMCALL The guest will invoke the *VMCALL* whenever it wants to make a system call to the host. The handler will need to interpret the guest state in order to determine what system call is requested and what its arguments are. Then it will invoke the normal system call mechanism. When the system call returns, it will write the return value into the guest state and resume it. The VT-specific system call layer within the guest will retrieve the return value and pass it back to its caller within UML.

Hardware interrupts Whenever a hardware interrupt, such as a timer tick or a device interrupt, happens while the UML guest is running, the host kernel will need to handle it. So, the VMEXIT handler will need to recognize that this was the cause of the transfer back to the host and invoke the IRQ system in the host.

Processor faults The guest will cause CPU faults in the normal course of operation. Most commonly, these will be page faults on its own text and data due to the guest either not having been fully faulted in or having been swapped out. These interrupts will be handled in the same way as hardware interrupts—they will be passed to the normal host interrupt mechanism for processing.

This thread will be the guest's representative within the host kernel. As such, it will be the target of any signals intended for the guest, and it must ensure that these signals are passed to the UML, or not, as appropriate.

In order to see that there is a signal that needs handling, the thread must explicitly check for pending signals queued against it. When a signal is queued to a process, that process is make runnable, and scheduled. So, if the signal arrives while the guest is not sleeping, then the thread will see the signal as soon as it has been scheduled, and deliver it at that point. If the signal is queued while the guest is running, then delivery will wait until the next time the thread regains control, which will be a hardware timer interrupt, at the latest. This is exactly the same as a signal being delivered to a normal process, except that the wakeup and delivery mechanisms are somewhat different.

If the signal is to be handled by the UML instance, as with a timer or I/O interrupt, then the thread must cause the signal to be delivered to the guest. This is very similar to normal process signal delivery. The existing guest CPU state must be saved, and that state must be modified (by changing the IP and SP, among others) so that when the guest resumes, it is executing the registered handler for that signal. When the handler returns, there will be another exit to the host kernel, analogous to sigreturn, at which point the thread will restore the state it had previously saved and resume the guest at the point at which the signal arrived.

If the signal is fatal, as when a `SIGKILL` is sent to the guest, the thread will shut the guest down. It will destroy the VT context associated with the guest and then call `exit()` on its own behalf. The first step will release any VT-specific resources held by the guest, and the second will release any host kernel resources held by the thread.

This is the same process that will happen on a normal UML shutdown, when the UML instance is halted, and it calls `exit()` after performing its own cleanup.

The final thing that the thread must do is check for rescheduling. Since it's in the kernel, it must do this explicitly. If the guest's quantum has expired, or a higher priority task can run, then a flag will be set in the thread's task structure indicating that it must call `schedule()`. The thread must check this periodically and schedule whenever the flag is set.

5 Guest setup

When it is launched, a UML/VT guest must do some setup which is hardware-dependent since it is running in ring 0. There are two principal things which must be initialized, system call handling and kernel memory protection.

System call handling As mentioned earlier, this is the area where we expect the greatest performance benefit from using VT. Before launching the guest, the host has specified to the hardware that it does not want a VMEXIT whenever a process within the guest causes a soft interrupt, as happens whenever it makes a system call. The guest will handle these directly, and the guest IDT must be initialized so that the guest's system call handler is invoked.

This will cause UML process system calls to be handled by the guest kernel without any involvement by the host. The host involvement (through `ptrace`) is what currently makes UML system calls so much slower than host system calls. This VT support will make UML process system calls run at hardware speed.

Kernel memory protection Another benefit of running in ring 0 is that UML gets to use the same hardware mechanisms as the host to protect itself from it processes. This is not available to processes—they cannot have two protection domains with the higher one being inaccessible by something running in the lower one. However, by initializing the guest GDT appropriately, UML/VT can install itself as the kernel within the guest domain.

6 Current status

The port of UML to VT is ongoing, as a project within Intel. All of the actual work is being done by two Intel engineers in Moscow, Gennady Sharapov and Mikhail Kharitonov. At this writing, they have finished the OS abstraction work, and I have that as patches in my development tree. These patches have started to be included in the mainline kernel.

The VT-specific work is now in progress. They are making VT system calls to the host and making the guest handle signals sent from the host. The next steps are the hardware initialization to handle system calls and to enable the protection of the kernel.

Following that will be the actual port. The OS abstraction work will be hooked up to the VT system calls in the os-vt layer. The host kernel thread will need to be fleshed out to handle all of the events it will see. Once this is done, it will be possible to start booting UML on VT and to start debugging it.

7 Conclusion

This paper has described the changes needed to make UML work in guest ring 0 with the VT extensions. However, a great deal won't change, and will continue to work exactly as it does today.

The UML address space will still be a completely normal process address space, under the full control of the host kernel. In the host, the address space will be associated with the kernel thread that is standing in for the VT guest. It will be swappable and demand paged just like any other process address space.

Because of this, and because UML will create its own processes as it does today, UML's copy-user mechanisms will work just as they do currently.

Resource accounting will similarly work exactly as it does today. UML/VT will use the address space occupied by its host kernel thread, and its memory consumption will show up in `/proc` as usual. Similarly, when the guest is running, its kernel thread will be shown as running, and it will accrue time. Thus, CPU accounting, scheduling priority, and other things which depend on process CPU time will continue to work normally.

In spite of being run as a kernel, in a ring 0, UML/VT will continue to maintain the characteristics of a process running within the host kernel. So, it will gain the performance advantages of using the hardware support provided by VT, while retaining all of the benefits of being a process.

Proceedings of the Linux Symposium

Volume One

July 20nd–23th, 2005
Ottawa, Ontario
Canada