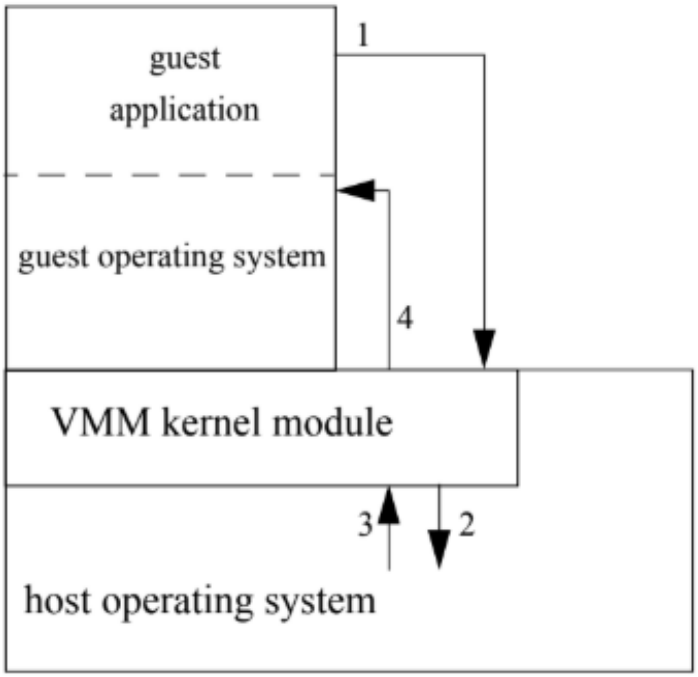
# Bottlenecks Of Type-II VMM UMLinux and Their Solutions

1.)The first bottleneck is UMLinux has a system structure with two separate host pro-cesses ,the Guest machine process and the VMM process. The VMM process serves two purposes: it redirects to the guest operating system signals and system-calls that would otherwise go to the host operating system, and it restricts the set of system calls allowed by the guest operating system. The VMM process uses **ptrace** to intercept events(signals and system calls) executed by the guest machine process and thus mediate access between the guest-machine process and the host Operating System. So a system call from the guest application causes an inordinate number of context switches on the host to get to the guest kernel to handle the system call.

***Solution:*** Most of these context switches can be eliminated by moving the VMM process's functionality into the host kernel. We encapsulate the bulk of the VMM process functionality in a VMM loadable kernel module. Moving the VMM process's functionality into the host kernel drastically reduces the number of context switches in UMLinux. For example, transferring control to the guest kernel on a guest system call can be done in just two context switches. It also simplifies the system conceptually, because

the VMM kernel module has more control over the guest-machine process than is provided by **ptrace**.




- guest application issues system call; intercepted by VMM kernel module
- VMM kernel module calls mmap to allow access to guest kernel data
- Mmap returns to VMM kernel module
- VMM kernel module sends SIGUSR1 to guest SIGUSR1 handler

2.)The second issue is protecting guest kernel space from guest application processes. The guest-machine process switches frequently between guest user mode and guest kernel mode. The guest kernel is invoked to service guest system calls and other exceptions issued by a guest application process and to service signals initiated by virtual I/O devices.

Each time the guest-machine process switches from guest kernel mode to guest user mode, it must first prevent access to the guest kernel's portion of the address space [0x70000000, 0xc0000000). Similarly, each time the guest-machine process switches from guest user mode to guest kernel mode, it must first enable access to the guest kernel's

portion of the address space. The guest-machine process performs these address space manipulations by making the host system calls **mmap, munmap, and mprotect.**

Unfortunately, calling mmap, munmap, or mprotect on large address ranges incurs significant overhead, especially if the guest kernel accesses many pages in its address space.

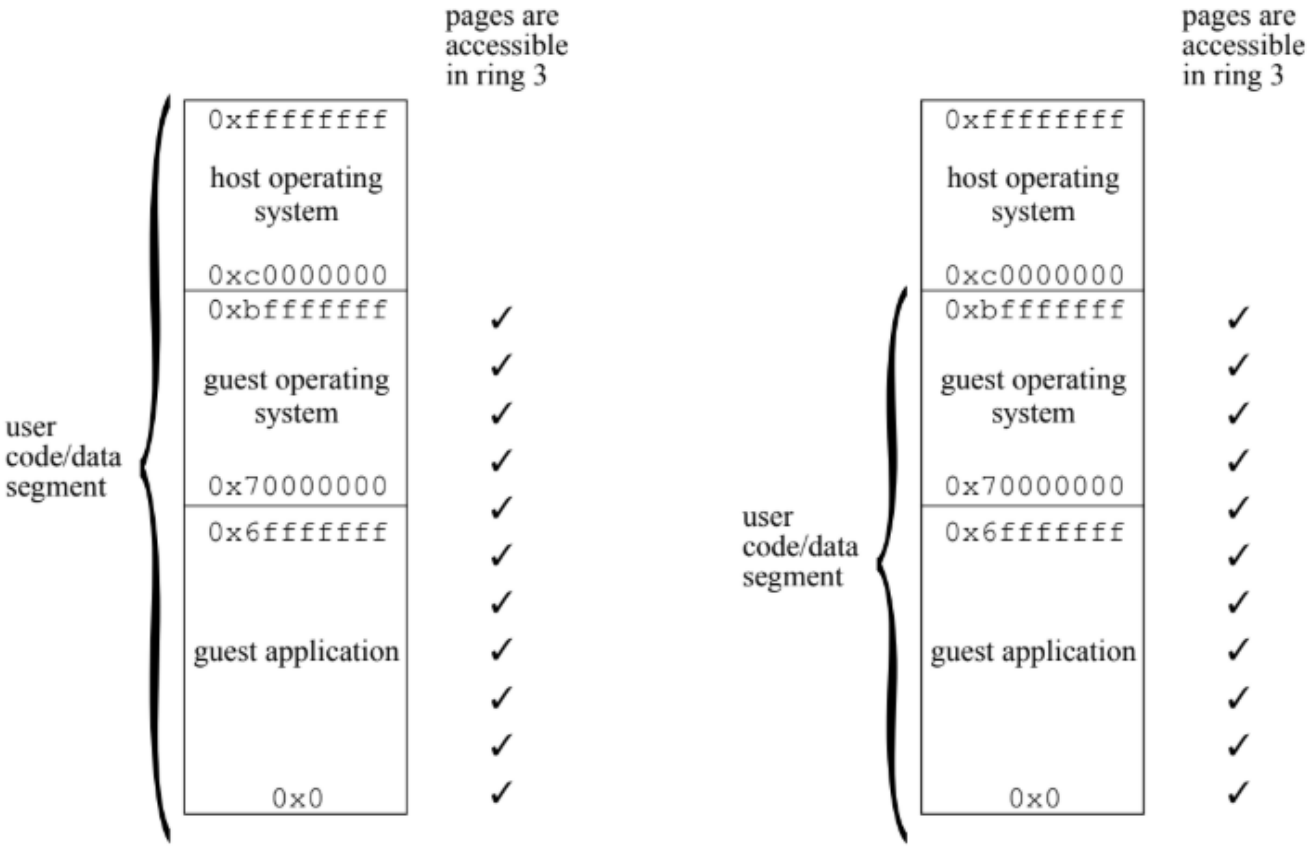
***Solution:*** Our first solution protects the guest kernel space from guest user code by changing the bound on the user code and data segments .When the guest-machine process is running in guest user mode, the VMM kernel module shrinks the user code and data segments to span only [0x0, 0x70000000). When the guest-machine process is running in guest kernel mode, the VMM kernel module grows the user code and data segments to its normal range of [0x0, 0xffffffff].

One limitation of the first solution is that it assumes the guest kernel space occupies a contiguous region directly below the host kernel space. Our second solution allows the guest kernel space to occupy arbi-trary ranges of the address space within [0x0,0xc0000000) by using the page table's supervisor-only bit to distinguish between guest kernel mode and guest user mode (Figure 8). In this solution, the VMM

kernel module marks the guest kernel's pages as accessi-ble only by supervisor code (ring 0-2), then runs the guest-machine process in ring 1 while in guest kernel mode. When running in ring 1, the CPU can access pages marked as supervisor in the page table, but it cannot execute privileged

instructions (such as changing the segment descriptor). To prevent the guest-machine pro-cess from accessing host kernel space, the VMM kernel module shrinks the user code and data segment to span only [0x0, 0xc0000000). The guest-machine process runs in ring 3 while in guest user mode, which prevents guest user code from accessing the guest kernel's data.

This allows the VMM kernel module to protect arbitrary pages in [0x0, 0xc0000000) from guest user mode by setting the supervisor-only bit on those pages.



3.) A third bottleneck in a Type II VMM occurs when switching address spaces between guest application processes. Changing guest address spaces means changing

the current mapping between guest virtual pages and the page in the virtual machine's "physical" memory file.

Changing this mapping is done by calling munmap for the outgoing guest application process's virtual address space, then calling mmap for each resident virtual page in the incoming guest application process. UMLinux minimizes the calls to mmap by doing it on demand, i.e.as the incoming guest application process faults in its address space. Even with this optimization, however, UMLinux generates a large number of calls to mmap, especially when the working sets of the guest application processes are large.

***Solution:*** To improve the speed of guest context switches, we enhance the host OS to allow a single process to maintain several address space definitions. Each address space is defined by a separate set of page tables, and the guest-machine processes switches between address space definitions via a new host system call switch guest. To switch address space definitions, switch-guest needs only to change the pointer to the current

first-level page table. This task is much faster than mmap'ing each virtual page of the incoming guest application process. We modify the guest kernel to use switchguest when context switching from one guest application process to another. We reuse initialized address space definitions to minimize the overhead of creating guest application processes. We take

care to prevent the guest-machine process from abusing switchguest by limiting it to 1024 different address spaces and checking all parameters carefully.