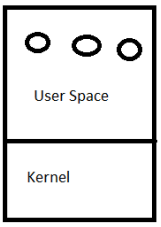
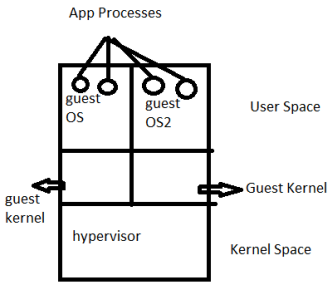
Operating Systems

Lab 2A

3) Review of Virtualization

In the given scenario, when the guest OS is not executing any privileged instructions there is no overhead. Since there are no privileged instructions, there are no system calls and the instructions are getting executed in the user space. So the virtualization should not make any difference and there is no overhead. The instructions should just get executed in the user space.





As can be seen from the diagram above any app process which is not doing any system calls and not executing any privileged intructions run in the user space and there is no difference between a system where virtualization has not been implemented.

Consider the assembly language program given below which subtracts 2 numbers

pushl %ebp

pushl %esp, %ebp

movl 4(%esp), %eax

movl 8(%esp), %ebx

subl %ebx, %eax

popl %ebp

ret

These above instructions no system calls are made and these just involve loading of the registers and then pushing and popping the values. Here no privileged intructions are getting executed, so there is no overhead.

Now when we are executing simple system calls like getpid(), this should also not make much a difference and there should not be any overhead. Consider a non-virtualized machine, when a process does a getpid it calls the kernel and retrieves the pid from the process table. This does not involve executing any complex intructions and is pretty straight forward. Now even in a Fully virtualized machine, when the getpid system call takes place, the app asks its kernel which is running in the user space. But this is not a complex instruction or a privileged instruction, the guest OS just has to retrieve the pid value and return it to the process. And hence there is no overhead involved in this case.

2)

Consider the case of issuing interrupts. In a Full virtualized machine, suppose one of the guest OS process issues a clear interrupt (cli) call. Now this is a privileged instruction and hence the guest OS will make a system call to the guest OS. But since the guest OS itself is running in the user space, it will not be allowed to execute the clear interrupt signal and disable all the interrupts for the entire system. This call is passed on to the hypervisor which then has to disable the interrupts for only that particular guest OS whose process issued the cli call. In this way the hypervisor must build firewalls between the guest OS's. If not then the interrupts will be disabled for all the virtual machines, which is not correct. This is where overhead is also added for the virtual machine.

So when executing the privileged intructions, the guest app process, calls to its guest OS. Since the guest OS is running in the user space, the hypervisor does not allow it to execute these privileged calls, thereby building a sandbox between the guest apps and the guest kernels. The hypervisor itself executes these privileged intructions and makes sure the instructions will not effect other guest OS's and thus build a firewall among guest OS's

3)

Sensitive instructions are POPF like intructions. And these were introduced in the Intel x86. These are neither privileged intructions nor non-priveleged intructions. And they depend on which mode, it is being run i.e the user mode or the kernel mode. During a contect switch, the contents of the current registers needs to be saved. There is a register called EFLAGs register which has 2 important bits the IF bit and the IOPL. IF is to enable/disable interrupts. The other important field is the IOPL (it has 2 bits – 00 for kernel mode is default). This IOPL is used to compare against CPL, and only when CPL<=IOPL, the privilege instructions can be executed. This is one of the field whose state is saved during the context switch. POPF puts the saved contents back and continues to execute. The EFLAGS are stored in the user space. Since, it is in user space, there is a probability that some rogue program or any other process might overwrite it and cause unusual behavior. They may set the IF flag to 0 to disable the interrupts or set the IOPL to 11 to execute the privileged instructions. But since POPF is a sensitive instruction, it will not cause the interrupt like a privileged instruction but it won't allow the user to change the EFLAG. In full virtualization environment this is a problem

because the guest kernel is also in the user mode. POPF loses its actual behavior as it is supposed to act in the native OS and it makes the guest kernel buggy. To solve this problem, VMWARE came up with just in time translation(Jit). It is an interpreter which does not precompile the code but compiles the code at runtime and it is kept in cache. Once this is done and the binary is put in cache, any other code revisiting this part of code looks up in the cache.

4)

1)

The 'P' is interleaved. This shows Xinu implements static scheduling as the priority of these processes were not changed.

2)

When the INITPRIO was set to 35, the main process was the highest priority process with priority 35. As a result, it got executed first and printed all the P and then the other processes with priority 20 were printed

3)

The first three have same priority as the main while the fourth process has the priority of 50. Due to this all the three will run and the moment the 4^{th} one is created, it will be run as it has the highest priority. Once this completes the rest of the other processes execute.

4)

When we execute with sleepms(1), the process sleeps for 1 ms and NULL process runs. With sleepms(0) it shows that NULL process is always running as it has priority of 0. When process is completed executing then NULL process starts executing until a process of higher priority comes. With sleepms(0) there is no interleave and it prints N. With sleepms(1) there is interleave.

5)

The CPU time for a process is implemented the following way. :

Declared another attribute for a process in struct procent called the prcputime.

Update the cputime in the clkhandler every millisecond when it runs. The cputime taken by an individual process is kept in millisecond. When the process is killed its prcputime is set to 0.