Lecture 5: February 3

Lecturer: Prashant Shenoy *Scribe: Jun Wang*

5.1 Case studies for multiprocessor and distributed scheduling Contd.

5.1.1 CONDOR

Condor is designed to make use of idle cycles on workstations, or to schedule jobs on a cluster of servers. Condor takes ideas from Sprite implements in LAN environment. An idle machine contacts the Condor coordinator and requests for a job, the coordinator sends a job back. The system executes the assigned job, and if the owner of the machine comes back, the job is suspended. It has support job migration (and thus machines in the cluster should be homogeneous) or restarting the job depending on which policy is in place. Condor has a flexible scheduling policy. When on a system of servers, jobs are submitted to a centralized queue, and the Condor coordinator takes a job and assigns it to a server in the pool. The Sun Grid engine has similar features, and is used to assign jobs on clusters.

5.2 Virtualization

Virtualization is a technique that extends or replaces an existing interface to mimic the behavior of another system. It takes an interface and uses it to emulate another interface.There are different types of interfaces,ranging from the basic instruction sets offered by the CPU to application programming interfaces. Virtualization was first designed by IBM in the late 1970s. It allowed legacy software to run on expensive mainframe hardware. In other words, the virtualization layer is like a portability layer that lets software compatible with an old hardware architecture to run seamlessly on the new hardware. However, these ported application cannot use the new features provided by the new hardware.

5.2.1 Types of Interfaces

Virtualization can be implemented at different levels of the hardware and software stack, depending on what one is trying to mimic. Computer systems offer four different types of interfaces, at four different levels:
1. An interface between the hardware and software,consisting of *machine instructions* that can be invoked by any program.
2. An interface between the hardware and software,consisting of machine instructions that can be invoked only privileged programs,such as an operating system.
3. An interface consisting of *system calls* as offered by an operating system.
4. An interface consisting of library calls, forming the *application programming interface(API).*

5.2.2 Types of Virtualization

- Hardware level virtualization
  - **(Full) Emulation** is where software is used to simulate hardware for a guest operating system to run in. As a result you can emulate any machine on another with different CPU architectures. For example, when Mac's were PowerPC based, Microsoft used to have a virtualization layer called VirtualPC that emulated an entire x86 PC on a PowerPC architecture. Inside VirtualPC you could run Windows, or Linux, or any other software that was designed to run on a x86 PC unmodified. The advantage is that any hardware can be emulated as the entire functionality of hardware can be implemented in software. The disadvantage is that the speed is slow. Each assembly instruction will be replaced by one or more instructions in the native environment. Each instruction has to be dynamically translated and executed. So the speed can sometimes be an order of magnitude slower.
  - **Full/Native Virtualization**: In this case, one interface is being used to emulate an underlying interface of the same type. E.g. emulating Intel x86 on Intel x86. This is mainly done, for example, to isolate failures. One could run Linux on Windows, or Windows on Linux. The advantage is that a full translation is not needed because the hardware family is the same. Assembly instructions map one-to-one as they are all, say, x86 instruction. So, the overhead is lower in this case. Examples include VMware, VirtualBox.
  - **Para-virtualization**: Here the VM does not simulate hardware but it requires modifications to the OS so as to on the virtual machine. Instead of executing sensitive instructions , the modified guest OS makes hypervisor calls.In effect the guest operating system is acting like a user program making system calls to the operating system(hypervisor) E.g Xen virtual machine monitor.

    In this approach, guest operating systems require modification to enable hypercalls from the virtual machine to the hypervisor. Instead of the hypervisor having to translate a potentially unsafe instruction from a guest operating system to guarantee system state integrity, a structured hypercall is made from the guest to the hypervisor to manage the system state changes.
- OS Level Virtualization: In this case, one OS emulates the interface of another OS. For example, by using OS-level virtualization an application that runs on linux kernel 2.x can continue to run on linux 3.x, where the old system calls are emulated by the new system calls. OS-level virtualization can also be used to provide resource isolation. It provides a notion of a smaller virtual server/resource container which has its own copy of the OS. The process running inside one of these containers just sees an ordinary OS, same as the native OS, but cannot see the processes in other containers. It is important to note that an entire copy of the OS is not running in the container. This is only a way to isolate the container, and allocate to it a part of the resources. For instance, the disk can be partitioned such that every container has access only to its own partition. OS level virtualization also lets you partition resources such as CPU such that one container can occupy only a certain percentage. In essence, resource isolation using OS level virtualization provides multiple system call interfaces in each container for the same OS. E.g. Solaris Containers and BSD Jails.
- Application Level Virtualization: It lets you emulate one application level interface on another. When Mac machines switched from PowerPC to Intel, the applications that users had purchased to run on the older Macs would not run on the newer versions. Rosetta was a solution that allowed emulation at the application level. If an application compiled for PowerPC was run on the new Mac, the OS would recognize that this as a PowerPC compiled binary. It would then dynamically attach a library to that binary. The attached library would intercept every PowerPC instruction, and on the fly translate and execute it natively. In a sense, this did what emulation does, except it did not actually expose an entire processor, but attached a library which was taking a call and translating it. Other examples include JVM as discussed before and WINE that lets you run windows application on linux or MAC by emulating the Win32 interface.

5.2.3 Types of Hypervisors

In this section, we introduce two approaches to virtualization.

- Type 1: In this case, the hypervisor (or virtual machine monitor) runs on bare metal, and the system is booted on the hypervisor. In reality, it is the only program running in kernel mode. Its job is to support multiple copies of the actual hardware, called virtual machines. On top of the hypervisor, multiple virtual machines with same hardware environment, but with different OSs. This type of hypervisor is typically only seen in server environments.
- Type 2: The type 2 hypervisor runs as an application on top of the OS. This hypervisor emulates the same underlying hardware and exposes the same hardware interface. There you can create a virtual machine and run a second OS, which could be different from the first OS. E.g. Windows could be run on a Mac OS X. To the host OS, the entire virtual machine looks like a single process. It has no knowledge of the OS and processes running inside the virtual machine.

5.2.4 How Virtualization Works

In relation to hardware architecture and processors, there is a notion of protection rings. Rings define levels of protection in terms of what privileges a process has when it runs at that level. The OS kernel runs at ring 0, and has full control over things happening in the machine. User processes run at ring 3 i.e. they can only run some restricted set of instructions, not including certain special instructions that only the kernel can run. These rings offer protection from user processes that can corrupt components and cause the system to come down. Instruction sets on any CPU can be partitioned into instructions that only kernel can run, and those that any process can run. The instructions that only the kernel can run are called *sensitive instructions*, e.g. instructions that can change page table settings or instructions that trigger I/O. If user processes try to execute these sensitive instructions, their request will be denied, as only the kernel has the privileges to execute them. Any assembly instruction that causes a trap or an interrupt is known as a *privileged instruction*. Example include segmentation faults while running programs, divide by zero, etc.

Claim: It is possible to implement type 1 virtualization only if *sensitive instructions* are a subset of *privileged instructions*. In other words type 1 virtualization can be implemented if the sensitive instructions trigger a trap.

5.2.5 Type 1 Hypervisor

In a type 1 hypervisor, the hypervisor should run in kernel mode as the hyperviosr schedules the VM's, allocates CPU and so on. The user processes that run on the VM will run in user mode. The guest OS that runs in the VM should also run in user mode yet should implement instructions that the kernel needs to execute. This can be done by implementing traps, where the guest OS asks the underlying hypervisor to implement the sensitive instructions on its behalf. Eg. In Intel 386 sensitive instructions executed in user mode are ignored. Therefore, there no way to implement type 1 virtualization for these systems. However, today's Intel CPUs have extra support called VT(virtualization tchnology). If VT support is turned on, on an Intel processor, a bitmap will list the instructions that should trigger a trap, if they are executed in user mode. For type 1 hypervisors, a good way to design the CPU is to have multiple rings, where ring 0 as before is the most trusted mode that the hypervisor runs in. Ring 3 is the least trusted mode that the user processes in the guest OS would run, and the guest OS could run in one of the middle modes. This way sensitive instructions from the guest OS will be executed but sensitive instructions from the user process on the guest OS will not.

5.2.6 Type 2 Hypervisor

In a type 2 setting, there is already an OS running on the machine and the hypervisor runs as a user process. The host OS kernel is trusted and it runs in kernel mode, everything else runs in user mode. Hence, any sensitive instruction that is executed by the guest OS kernel will not be allowed to execute. To solve this problem type 2 hypervisors scan guest OS instructions as they are about to execute, and whenever there is a kernel instruction it is replaced with a function call to the hypervisor. The hypervisor, in turn, triggers a system call and requests the host OS to execute that instruction. The host OS then executes that instruction for the hypervisor. Hence, type 2 hypervisors require binary translation where guest OS kernel instructions are changed on-the-fly, and are replaced with hypervisor instructions. The disadvantage is that the systems takes a performance hit and runs more slowly. However, the advantage is that no hardware support is needed for virtualization. So, you can use type 2 hypervisors on older systems as well (e.g. on Intel processors without VT support). All sensitive instructions are replaced by procedures that emulate them.

5.2.7 Paravirtualization

Paravirtualization is a technique to run a guest OS on top of a type 1 hypervisor, without requiring hardware support. Up until now, the assumption was that the OS is unmodified. However, in paravirtualization, the assumption is that the OS can be modified. In effect, a programmer modifies the OS by taking each kernel mode instruction and replacing it with a function call to the hypervisor. So, in some sense, it is similar to what was being done in type 2 hypervisors, except now it is not done on-the-fly, but instead, it is done beforehand in the source code, by the programmer. The programmer looks at kernel code, and every time the kernel makes a system call, it is replaced with a function call to the hypervisor. These calls to the hypervisor are called 'hypercalls'. So the programmer replaces all sensitive instructions with hypercalls. Paravirtualization can be implemented without hardware support, as there are no kernel instructions left in the kernel. All kernel instructions are replaced by hypercalls to a type 1 hypervisor, which runs in kernel mode and can execute those instructions on behalf of the OS.