CMPSCI 677 Operating Systems **Spring 2013**

Lecture 5: February 6

Lecturer: Prashant Shenoy *Scribe: Vani Gupta*

5.1 Recap

Last time distributed scheduling was discussed. Distributed scheduling is most useful when the system is moderately utilized. When the load is on a system is light, jobs can be executed wherever they are submitted. When the load on the system is high, there is nothing much that can be done by moving load around. Sender-initiated, receiver-initiated, and adaptive policies were discussed.

5.2 Systems that implement Distributed Scheduling

Examples of systems that implement distributed scheduling, including two research-based, and two real-world systems, are described next.

5.2.1 Case Study 1: V-System (Stanford)

V-System was a distributed operating system designed in the late nineties. It was one of the two research projects that came up with the notion of distributed scheduling, but did not become a commercial project. It assumed a state-change driven information policy, in which if a system saw any significant changes in load (memory utilization or CPU utilization), it broadcast its load information to all nodes. So every node had an insight into the load of all systems. Nodes sent these broadcast messages asynchronously. 'M' least loaded system were receivers, and all the other nodes were senders. The distributed scheduling policy was sender-initiated. Given that the load information may be stale, the location policy involved polling a random receiver from the list of receivers, and confirming whether it was still a receiver. Load information could have been stale because a node that was a receiver in the past may have received jobs and become much more loaded. V-Systems had no process migration, so only new tasks were send to other systems.

5.2.2 Case Study 2: Sprite (Berkeley)

Sprite was also a distributed operating system. It was developed at Berkeley, around the same time as V-System, and it also had distributed scheduling mechanisms built into it. It was built for an environment where every user had a powerful workstation. It was based on the assumption that the load on these workstations was variable - some were heavily loaded, while others were idle. The idea was to take advantage of systems that are lightly loaded or idle. The system determined which workstations were idle by tracking keyboard/mouse activity. If there was no activity for a certain time period (in this case 30 secs), and the load on the machine was below a certain threshold, the machine was considered idle. Sprite tried to ensure that the owner of the workstation was not impacted by the foreign jobs that were moved to their idle system. So, if a user came back to his/her idle system, it was important that their system not be unresponsive. This was achieved by stopping all background jobs and sending them elsewhere as soon as the workstation became active. Sprite implemented a centralized policy, in which there was a coordinator that kept track of which systems were idle and which were loaded. The coordinator would allow senders to send jobs to idle machines. Sprite was state-change driven, so as soon as a workstation became idle, it sent a message to the coordinator that it was willing to accept jobs. The location policy required the sender to query the coordinator, and the coordinator would tell the sender where to send jobs. Process migration was a new feature that Sprite added. If the user of a workstation became active, then the foreign jobs executing on that workstation could be migrated to another workstation. In case of Sprite, the file system was distributed and available to all workstations. In order to migrate a process, the state of a suspended process was saved to disk, a new process was created on the receiving machine, the state of the suspended process was loaded from disk to memory on the receiver, and the execution of the migrated process was resumed from where it was suspended. References to open files could be handled fairly easily, given all files were stored on a network server or a distributed file system. The new workstation where the process was migrated had a new IP address, so issues like open sockets were harder to deal with. There were one of two ways this could be handled. A restriction could be put saying process migration would only work for processes that were not doing network I/O, or tuneling could be used to forward packets to the new IP address associated with the migrated process. For now, the assumption is that all the workstations were homogenous systems.

5.2.3 Case Study 3: Volunteer Computing

Volunteer computing is in use today and is also called Internet scale operating system (ISOS). Users that have idle machines can volunteer their machine's cycles for a good cause. In volunteer computing, when a user as a very large computationally intensive task that requires a large cluster that they do not have, they ask users to donate their systems free cycles. These tasks are generally based on a good cause, which provides an incentive for other users to donate free cycles. Users that donate free cycles download software that runs as a screenscaver on their system. When the system becomes idle, and the screensaver is activated, it contacts a central server asking for a job. When the user comes back and starts using the system, it deactivates the screensaver and the task stops running. The user does not have to do anything special to make this happen. Volunteer compting can be used for embarrassingly parallel tasks like search for extra terrestrial intelligence (SETI@Home) where signals from outer space are scanned for patterns. SETI was the first to do this but there are other systems like BOINC etc. that also use volunteer computing. Application designers looking for free cycles can contribute tasks, contact the coordinator of BOINC, and get free cycles. These application designers could for example be working on drugs for cancer. Users interested in helping these designers in their cause, can volunteer their cycles. Loose form of distributed scheduling where there is a coordinator assigning tasks, which are scheduled on different machines, results are sent back, and machines come and go. Google SETI@Home and BOINC to learn more.

5.2.4 Case Study 4: 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 or restarting the job depending on which policy is in place. Condor has a flexible scheduling policy. It can run in a workstation environment or on system of servers. In case of a system of servers, jobs are submitted to a centralized queue, the Condor coordinator takes a job assigns it to a system in the pool. The Sun Grid engine has similar features, and is used to assign jobs on clusters. It is used on the 'Swarm' cluster to assign jobs. Users submit jobs on Swarm, and the distributed scheduler running on the Sun Grid engine decides which of the 100 cores will run a job.

5.3 Virtualization

Virtualization means that you extend or replace an existing interface to mimic the behavior of another system. It takes an interface and uses it to emulate another interface. E.g. Suppose a system is running Windows. If there is software running on this Windows system that exposes, say Linux API. Then, because of that software you could technically run a process that runs on Linux, on a Windows machine. This is an example of virtualization. Virtualization was first designed by IBM in the late 60s. Virtualization enabled IBM customers to run their old code on newer systems, by giving them the option of exposing the old mainframe interface on newer machines.

5.3.1 Types of Interfaces

There are many types of interfaces. Virtualization can be implemented at different levels of the hardware and software stack, depending on what one is trying to mimic.

- Hardware level virtualization: This is the lowest level of virtualization. Here one processor is used to mimic another processor.
- OS level virtualization: Here one system call interface is used to mimic another system call interface E.g Linux running on Windows
- Library level virtualization: Use one library interface to mimic another library interface. E.g. the JVM uses the C library to expose a Java interface.

5.3.2 Types of Virtualization

- Emulation or Hardware level virtualization: This is used to emulate one hardware on another. The virtual machine emulates a complete piece of hardware using software. As a result, an unmodified guest OS for a different PC can be run on a machine. E.g. 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 PC. 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 of 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 on Intel. 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.
- Para-virtualization: Here the VM does not simulate hardware. In paravirtualization, the virtualization layer is called the hypervisor, which traps calls from the operating system and executes them. E.g Xen.
- OS-Level Virtualization: In this case, the hardware and OS are the same, and it is used for performance or security isolation. E.g. Solaris Containers and BSD Jails. Untrusted code can be run inside the container without impacting other applications. The process running inside the container sees just an ordinary OS, same as the native OS. Even if the applications running inside the container are malicious,

buggy, or crash, the apps running outside are unaffected. Also, if the code in the container gets stuck in an infinite loop, the cycles available to process outside the container are unaffected. A certain amount of resources like a fixed amount of RAM and a given fraction of the CPU can be allocated to the container. 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. The main OS continues to service all the requests of the container, and the applications in the container see the native OS. In Solaris you can expose a certain version of the OS to the container. Solaris also provides this feature for compatibility with older applications that run on older versions of the OS.

- Application Level Virtualization: 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 it and execute it natively. In a sense, this did what emulation does, except it did not actually expose an entire processor, it attached a library which was taking a call and translating it.

5.3.3 Types of Hypervisors

Hypervisors that do native virtualization:

- Type 1: In this case, the hypervisor or virtualization software runs on bare metal, and the system is booted with the hypervisor, not with an OS. On top of the hypervisor a virtual machine that emulates the same hardware environment can be run. So, multiple virtual machines can be running with same hardware environment, but with different OSs. This type of hypervisors are typically only seen in server environments.
- Type 2: In this case, the machine runs some native OS. 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. In there you can create a virtual machine and run a second OS, called the Guest OS, which could be different from the first OS. E.g. Windows could be run on a Mac OS X. VMWare fusion was demonstrated as an example of a type 2 hypervisor. In the example, the VM was a Software PC with Linux, and there was a Firefox application running in the VM. So, we could run Linux on a Mac OS X. To the host Mac OS X, the entire virtual machine looks like a single process. It has no knowledge of the OS and processes (like Firefox) running inside the virtual machine.

5.3.4 How Virtualization Works

In relation to hardware architecture and processors, there is a notion of 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 (including the kernel). 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 processes try to execute these sensitive instructions, their request will be denied, as only the kernel has the privileges to execute them. Now, any assembly instruction that causes a trap or an interrupt is known as a *privileged instruction.*

Claim: It is possible to implement type 1 virtualization only if *sensitive instructions* are a subset of *privileged instructions.*

In a type 1 hypervisor when a process in the virtual machine wants to do I/O (say open a file), the guest OS tries to execute the instructions as if it is directly running on hardware. So, say it tries to open a file in the underlying disk, just like it would normally do. Now, this guest OS is running in a virtual machine, and is in ring 3. So, it will not be able to access the hard disk. The hypervisor is trusted and runs in ring 0. So, when sensitive instructions are executed in user space, somehow the hypervisor needs to be notified to execute those instructions on behalf of the guest OS. Therefore, we would like sensitive instructions to cause a trap and use that as a way to let the hypervisor know that the guest OS needs to execute a sensitive instruction.

E.g. 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. 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.