CMPSCI 377 Operating Systems **Fall 2013**

Lecture 2: September 9

Lecturer: Prashant Shenoy *Scribe: Armand Halbert*

2.1 OS & Computer Architecture

The operating system is the interface between a user and the underlying architecture of her computer. This lecture discusses the important features of modern operating system and explains how many of these are dependent on support from the hardware architecture underneath the OS.

2.1.1 Important OS Functionality

The operating system provides concurrency by allowing multiple users and applications to run simultaneously without interference. Similarly, the OS is responsible for allowing multiple applications to simultaneously access a single hardware device. The OS achieves these goals by supporting threads and processes. A single CPU can only process one thread at a time, but the OS gives the impression of concurrency by splitting a CPU's time between multiple threads or processes.

The operating system must also manage I/O devices both to support concurrency within a single device, and to allow one application to utilize the CPU while others are blocked waiting for I/O requests to be processed. The operating system controls I/O devices by through code in the kernel for each device (device drivers). User applications can utilize I/O devices through the kernel which in turn uses the device drivers.

Each process is also granted memory by the OS. The OS must protect the memory of each application to prevent malicious or erroneous applications from modifying their memory data. The OS also provides mechanisms to allow data to be transferred from disk to memory.

The OS takes raw disk devices and presents users and applications with the abstraction of files and directories with the use of file systems. The OS must manage the disk space used by files and must ensure protection between users by managing which users have access to which files.

Applications increasingly rely on network connectivity. Since the OS is responsible for managing I/O devices, it must control the network card and allow multiple applications to use it for communication.

Modern operating systems provide all of this functionality, but how it is implemented varies between systems. How the operating systems provide this functionality has evolved over time, and changes both as new software techniques are developed and due to changes in computer architecture.

2.1.2 Generic Computer Architecture

A general model of a modern computer has a CPU, I/O devices, memory, and the system bus. The CPU (Central Processing Unit) is the processor that executes all of the instructions. This can be a single core, but more common today is multiple cores where each core can simultaneously perform computations. Having multiple cores tends to complicate the OS so we will generally assume a single core in this class. I/O devices are everything from a terminal or video card to a printer or network card. An I/O device can be anything that the user uses to interact with the machine. Memory is the RAM (Random Access Memory) that contains all of the data and program code that is used by the CPU. The CPU, I/O devices, and memory all communicate by the system bus.

2.2 Protection Boundaries and System Calls

An important feature provided by the OS is protection. Protection is provided in part by dividing the set of instructions supported by the hardware into ordinary and privileged instructions. While many instructions can be safely run by an application without concern of corrupting the system, instructions that perform I/O directly, control memory control state such as page table pointers, manage interrupts, or halt the machine must be carefully managed by the OS itself. The OS enforces this by requiring that these privileged instructions be run within *kernel mode*. All normal processes are run within *user mode*; only the trusted OS kernel code itself is run in kernel mode.

If a process attempts to run a privileged instruction while in user mode, a trap is triggered that switches the system back into kernel mode. This allows the OS to stop the process from performing the illegal operation. To achieve this, the OS relies on support from the hardware in the form of a kernel mode bit which can be checked to determine what mode the system is currently running in. By ensuring that only the OS itself can modify this bit, this hardware feature can be used to provide protection for privileged instructions.

To completely prevent processes from ever using privileged instructions would greatly limit their functionality. In order to allow processes to safely perform privileged actions such as opening a file, a process must ask the OS to perform the action on its behalf using a *system call*. System calls allow the kernel to check the requests being made by the process and determine whether to handle the request or return an error. Each operating system provides a different set of system calls, but in general they cover process management, file I/O, device management, access to system information, and communication mechanisms.

2.2.1 Traps

When a user process makes a system call it triggers a *trap* in the hardware. A trap is a CPU event that is triggered by a program, similar to a software level exception. Traps are sometimes called software interrupts. They can be deliberately triggered by a special instruction, or they may be triggered by an illegal instruction or an attempt to access a restricted resource. In either case, the trap causes the running user process to be paused and the system switched to kernel mode. The kernel's *trap handler* then processes the event as needed. The trap handler saves the caller's state (Program Counter, mode bit, etc.) so that it can be restored after the system call. In the case of a system call, the kernel can determine which call the process was attempting to run and will process it on its behalf. The kernel will then place the result of the call into memory accessible by the process and unpause it. The process then resumes and can immediately read the result of the system call.

Traps are implemented within the OS with the help of a trap vector that maintains a list of memory addresses that correspond to different triggers. This table is a hardware feature and is used by the OS so that when a trap occurs, the relevant kernel routine can be called by loading the code at the specified memory region. The trap table is really just an optimization–the same goal could be achieved by instead adding additional instructions within the user process itself.

2.2.2 Memory Protection

A second form of protection that the OS must provide is the isolation of the memory used by each application. Additionally, the OS must protect its own memory from being read or modified by user programs. The OS again relies on hardware features to support this feature. The hardware maintains a pair of *base* and *limit* registers that are loaded by the OS before each process gains access to the CPU. These registers are set to the start and end regions of RAM which the process is supposed to have access to. Any subsequent memory accesses made by the application are checked by the hardware to ensure that they are between the base and limit registers. If a process attempts to reach memory outside of its allowable region, then a trap is triggered that allows the kernel to prevent the illegal memory access.

2.3 Memory Hierarchy

The memory in a machine is typically divided up into a hierarchy. The hierarchy allows a tradeoff between the amount of storage available and the time required to read or write data. The fastest component in the hierarchy is the set of machine registers. Data stored in a register can be accessed in a single CPU cycle. Below this, a CPU may have several caches such as an L1 and L2 cache. These caches can hold significantly more entries, but can take multiple cycles to access. Next is the machine's main memory which can have a latency of around 100 cycles, but can hold gigabytes of data. When RAM is not enough, applications can store data on disk, providing substantially larger storage areas at the expense of access times that are several orders of magnitude longer. Finally, the network itself can be viewed as part of the memory hierarchy, allowing data to be stored on different machines, but further increasing the access latency. As an application runs, parts of its code and data may be in different areas of the memory hierarchy.

2.3.1 Registers

Registers are specialized pieces of memory in the CPU, and can be seen as dedicated names for specific *words* of memory. Some of them are of general purpose (such as AX, BX, CX, which on a x86 can be used for addition, multiplication, etc). Other registers have special purposes: the SP (Stack Pointer), for instance, is used to point to one of the ends of the stack; the PC (Program Counter) is used to point to the current instruction being executed; and so on.

Since there are so few registers, they are not divided up between processes. Instead, each process is given access to all of the registers. When the kernel switches between processes, the state of each register is saved to main memory and the new processes' registers are reloaded. This procedure is known as a *context switch* and must be performed every time the kernel switches between the different processes running in the system.

2.3.2 Cache memory

While registers allow for incredibly fast access times, they are expensive and only a small number can be included with each CPU. Caches are used as a performance optimization between registers and main memory. The role of the cache memory is to be a faster-than-RAM memory, but at a more affordable cost than registers. Since cache memory is not as big as the main memory, the OS needs smart ways to manage it. Usually, caches hold **recently-accessed data or instructions**. The crucial assumption that justifies this approach is that data recently accessed might be needed again in a near future, and that data *near* the one just accessed might also be needed soon.

2.4 I/O Devices

Most I/O devices have a hardware component called a controller, for example a disk controller or usb controller. The controller contains a small processor which can be issued commands from the host operating system through the system bus. When the controller completes a request, it triggers an interrupt which propagates back to the operating system. In turn, the OS uses this interrupt to return the relevant data back to the user level process which initiated the request.

I/O requests can be either synchronous or asynchronous. In a *synchronous* or "blocking", I/O request, the calling process waits while the request is handled by the I/O device. Alternatively an *asynchronous* I/O request can be used to allow the CPU to continue working while the I/O controller processes the request. This can lead to improved system performance since the process can continue to run while the I/O request finishes. Upon completion, the I/O device triggers an interrupt which returns the data back to the process. This form of interrupt is processed similar to a system call. In practice, synchronous I/O is often preferred since asynchronous I/O requests can require more complicated programming.

Memory-mapped I/O is an I/O scheme where the device's own on-board memory is mapped into the processor's address space. Data to be written to the device is copied by the driver to the device memory, and data read in by the device is available in the shared memory for copying back into the system memory. Memory-mapped I/O is frequently used by network and video devices. Many adapters offer a combination of programmed I/O and memory-mapped modes, where the data buffers are mapped into the processor's memory space and the internal device registers that provide status are accessed through the I/O space. The adapter's memory is mapped into system address space through the PCI BIOS, a software setup program, or by setting jumpers on the device. Before the kernel can access the adapter's memory, it must map the adapter's entire physical address range into the kernel's virtual address space using the functions supplied by the driver interface.

2.5 Timers

Clocks and timers are additional hardware features utilized by operating systems. Timers are particularly useful for triggering the context switches that transfer control of the CPU between different processes. To give the impression of many processes running simultaneously on a single CPU, the OS runs each process for a small time quantum, for example 10 milliseconds. When a timer expires after this interval, the context switch is called to save the current process's state and load the registers for the next process.

Timers are implemented using hardware clocks and a hardware interrupt table. A timer can be set by saving a timer value into a special register which is then decremented every millisecond. When the timer runs out, an additional register is checked to determine what code should be triggered, for example the context switch code or an update to the system clock.

2.6 Synchronization

Synchronization is important so that the OS can coordinate state among cooperating processes. Synchronization is particularly important since interrupts, for example from asynchronous I/O requests, can arrive at any time and disrupt running code. Similarly, multiple processes may need to synchronize shared data. Since processes can be scheduled at unpredictable times and can be interrupted at will, it can be difficult to safely synchronize this data without hardware support. We will discuss synchronization mechanisms in more detail during subsequent lectures; here we discuss two hardware features that are used by the OS to support these mechanisms.

To prevent interrupts from disrupting kernel functions, the hardware supports the idea of *atomic sections*. An atomic section is a region of code which must be run without interruption. This is achieved by disabling interrupts prior to starting an atomic section and only enabling them after the code segment is complete. During this time, any incoming interrupts must be queued by the hardware so they can be processed after the section ends.

Recent computer hardware also supports the *test and set* instruction to help support synchronization between processes. This acts as a single instruction that is able of testing if a variable is true, and if not setting it to be true. While this sounds simple, it is a powerful instruction that can be used as a primitive to create more complex synchronization mechanisms to coordinate multiple processes. The design of these mechanisms will be discussed in subsequent lectures.

2.7 Virtual Memory

Virtual memory is an abstraction that gives processes the impression that a system has an infinite amount of memory available to it. While a laptop or server may only have 2GB of RAM, virtual memory can be used to allow the operating system to start up processes which utilize more memory than this physical limit. To achieve this, the OS only loads a portion of each process into memory at once. The remaining memory data for a process is actually stored on disk while not in use. The OS then *swaps* data between memory and disk in order to keep the required data loaded into RAM at all times. While this gives the illusion of infinite memory, loading data to and from disk can significantly decrease performance of any running applications. In order to implement the virtual memory abstraction, the underlying hardware must support additional registers and lookup tables. These are used by the OS to determine how memory addresses for a process map to either regions currently in RAM or data stored on disk.