CSE 325: Operating Systems

Lecture 1: Introduction

1.1 Components of Computer System

We begin the lecture by the discussion of operating system's role in the overall computer system. A computer system is roughly divided into four (4) components including users, applications, operating system and hardware. We can also view a computer system with software, hardware and data. There are two (2) major types of software- application software and system software. Application software provides services to the users. It helps users to solve problems or carry out given tasks. A word processor (MS Office) and Web browsers are examples of application software. System software coordinates the activities and functions of hardware and software, and it controls the operations of computer hardware. A computer's operating system is an example of system software. System software also includes utility software, device drivers and firmware. Utility software is a part of OS and helps to manage, maintain and control computer resources. Device drivers act as a translator between the operating system of the computer and the device connected to it. Firmware is software that is embedded in a piece of hardware and provides the low-level control for that device's specific hardware. The hardware part of OS consists micro architecture, device controller and the physical device. Micro architecture describes the processor components, interconnectivity, and interoperation to implement predefined instruction set architecture (ISA). The microinstruction set, execution model and formulation including processor registers and data implementation are described by ISA and micro programmed inside programmable Logic Array (PLA)* device or Read-only Memory (ROM)* or Electrically Erasable Programmable ROM (EEPROM)*.

*PLA
*ROM The details of those circuits, you will learn digital logic design course.
*EEPROM

1.2 Definition of Operating System

Now the question arises- what is operating system?

Operating system is a program that acts as an intermediary between a user & hardware. It takes specific commands from user(s) or application program(s), operates/executes them by controlling existing hardware and software resources. It plays a role of coordinator to coordinates between user and hardware. The fundamental goal of OS is to execute user programs and to make solving user problems easier. This directly reflects OS as a resource allocator to operate, to manage and to utilize all the available resources and takes decision between conflicting requests in a well-organized and fairly manner. In addition of resource allocation OS acts as program controller to control program execution to prevent errors and improper use of the computer resources as example memory protection during multitasking.

OS needs to be convenient. Thus, it services user requests by abstracting the implementation or processing details. Let's take an example to visualize the OS abstraction. Reading/Writing a file from hard disk, user just needs to place a command like read/write to their program and the disk path location. User/user program does not require knowing the internal hard disk structure including cluster, sector, track to define the particular file. OS helps to do that for the user. In addition, the abstraction helps not to change the code for another hard disk vendor specification.

1.3 Operating System Structure and Organization

Lets first introduces the full computer system then we will discuss the operation mechanism of the system. Modern computer system consists one or more CPUs and a number of device controllers* (chip/circuit with registers, memory buffer etc.) connected through a system bus and that provides access to the shared memory. Each device controller is in charge of a specific type of device (for example, disk drive, audio device or video displays). The CPU and device controllers can execute parallel, competing for memory access. To ensure orderly access to the shared memory, a memory controller synchronizers access to memory. When a computer starts running or powered up or rebooted- it needs to have an initial program to run. This initial program is called bootstrap program. Typically, it is stored in ROM or EEPROM, known by the general term firmware. It initialized all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the OS (kernel) and how to start executing that system. Once OS is loaded and executing, it can start providing services to the system and the users.

OS consists of two major parts- kernel and shell. Kernel is the heart and soul of an OS. Useful applications and utilities are added over the kernel, and then the complete package becomes an OS. Kernel directly controls the computer hardware by performing OS services using specific system calls or requests or hardware interrupts. Linux is a kernel as it does not include applications- file-system utilities, windowing systems and graphical desktops, system administrator commands, text editors, compilers etc. Various companies added these kinds of applications over Linux kernel and provided their operating system including ubuntu, centOS, redHat etc. There are three (3) types of kernel – monolithic kernel, layered kernel and micro kernel. Monolithic kernel is a single large process running entirely in a single address space. This is the most common OS architecture where all parts of entire OS including scheduler, file system, memory, device drivers etc. are working in kernel as separate modules, and the modules are loaded or unloaded at runtime dynamically. Inter module communication is done by signals or sockets. Monolithic kernel is faster in execution; however addition a new module needs to recompile the whole kernel and crashing a single module needs to reinstall the whole kernel. Kernel size is very large thus porting OS modules to another location is inflexible. Examples of operating systems that use a monolithic kernel are - Linux, BSDs (FreeBSD, OpenBSD, NetBSD), Solaris, OS-9, AIX, HP-UX, DOS, Microsoft Windows (95,98,Me), OpenVMS, XTS-400 etc. Layered kernel was built by E.W. Dijkstra and his students and had six layers, each one constructed upon the one below. Layer0 deals processor allocation, process switching during interrupts or timers expired and thus, it provides the basic multiprogramming of the CPU. Layer1 does the memory management and drum (magnetic data storage device) management. It allocates space for processes in main memory and on a 512K word drum is used to hold surplus parts of the process (pages). Processes do not have to worry about whether they are in memory or on the drum. The layer1 software take care of making sure pages are brought into memory whenever they are needed. Layer2 handles communication between each process and its own operator console. Layers3 manages I/O devices and buffering the information to and from them. Layer4 holds the user programs. They do not have to worry about process, memory, console, or I/O management. The system operator process is located in layer5. MULTICS have the layering concept. In microkernel, the kernel is broken down into separate modules, only one of which is microkernel that runs in kernel space others (including device drivers, file systems) are in user space. As the modules are separated, bugs in a module can crash that particular module only but cannot crash the entire system. Kernel size is relatively small and thus porting kernel to another location is flexible. In addition, adding a new feature, do not need to recompile the kernel. Inter module communication is done by message passing and thus micro kernel execution is 2-4% slower than monolithic kernel. Examples of operating systems that use a microkernel are - QNX, Integrity, PikeOS, Symbian, L4Linux, Singularity, K42, Mac OS X, HURD, Minix, and Coyotos.

Shell is another important part of OS. It serves as the interface between user and kernel. User enters commands via shell in order to use the kernel services. Shell is represented into two (2) approaches- command line interface (CLI) and graphical interface (GUI).

The design concept of operating system contains- the main core kernel process is at the core. On top of the kernel has separate resource management modules including process management, memory management, storage management, I/O management, deadlock management etc.

1.4 OS Operations

1.4.1 How does a user/user program communicate with Kernel?

Application/user program does not have access to execute some of the kernel's privilege codes/modules. Operating system (kernel) wants to execute those with special monitoring. In order to ensure the proper execution of the operating system, we must be able to distinguish between the execution of kernel code and user defined code. The approach taken by most computer systems is to provide hardware support that allows us to differentiate among various modes of execution including user mode and kernel mode. A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we can distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill the request.

During a program execution phase, the program needs to run its own application instruction as well as kernel's privilege codes to communicate with I/O device modules or kernel's special services. Kernel names all kind of services by numbers. Thus, application program first asks services to the kernel by initiating software interrupt assembly language instruction with a specific service number (argument to the interrupt instruction). Kernel traces the service number, passes it to the index of an array in the OS kernel with pointers to the functions, executes that function and provides the requested service. When CPU executes kernel instruction then it will be in kernel mode otherwise it will be in user mode. System call provides an interface to the services made available by an operating system. System call is a request from an application program/user to the operating system (kernel) to perform some hardware action on behalf of the application. When a system call is executed, it is typically treated by the system as software interrupt. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to kernel mode. The system-call service routine/ specific device driver routine is a part of the operating system and starts execution.

1.4.2 How does kernel communicate with I/O devices?

The control of devices connected to the computer is a major concern of operating-system designers. Because I/O devices vary so widely in their function and speed (consider a mouse, a hard disk, and a tape robot), varied methods are needed to control them. These methods form the I/O subsystem of the kernel, which separates the rest of the kernel from the complexities of managing I/O devices.

I/O-device technology exhibits two conflicting trends. On the one hand, we see increasing standardization of software and hardware interfaces. This trend helps us to incorporate improved device generations into existing computers and operating systems. On the other hand, we see an increasingly broad variety of I/O devices. Some new devices are so unlike previous devices that it is a challenge to incorporate them into our computers and operating systems. This challenge is met by a combination of hardware and software techniques. The basic I/O hardware elements, such as ports, buses, and device controllers, accommodate a wide variety of I/O devices. To encapsulate the details and oddities of different devices, the kernel of an operating system is structured to use device-driver modules. The **device drivers** present a uniform device access interface to the I/O subsystem, much as system calls provide a standard interface between the application and the operating system.

1.4.2.1 I/O Hardware

Computers operate a great many kinds of devices. Most fit into the general categories of storage devices (disks, tapes), transmission devices (network connections, Bluetooth), and human-interface devices (screen, keyboard, mouse, audio in and out). Other devices are more specialized, such as those involved in the steering of a jet. In these aircraft, a human gives input to the flight computer via a joystick and foot pedals, and the computer sends output commands that cause motors to move rudders and flaps and fuels to the engines. Despite the incredible variety of I/O devices, though, we need only a few concepts to understand how the devices are attached and how the software can control the hardware.

A device communicates with a computer system by sending signals over a cable or even through the air. The device communicates with the machine via a connection point, or **port**—for example, a serial port. If devices share a common set of wires, the connection is called a bus. A **bus** is a set of wires and a rigidly defined protocol that specifies a set of messages that can be sent on the wires. In terms of the electronics, the messages are conveyed by patterns of electrical voltages applied to the wires with defined timings. When device *A* has a cable that plugs into device *B,* and device *B* has a cable that plugs into device *C,* and device *C* plugs into a port on the computer, this arrangement is called a **daisy chain**. A daisy chain usually operates as a bus. Buses are used widely in computer architecture and vary in their signaling methods, speed, throughput, and connection methods. A typical PC bus structure appears in Figure 13.1. In the figure, a **PCI bus** (the common PC system bus) connects the processor–memory subsystem to fast devices, and an **expansion bus** connects relatively slow devices, such as the keyboard and serial and USB ports. In the upper-right portion of the figure, four disks are connected together on a **Small Computer System Interface (SCSI)** bus plugged into a SCSI controller. Other common buses used to interconnect main parts of a computer include **PCI Express (PCIe)**, with throughput of up to 16 GB per second, and **Hyper Transport**, with throughput of up to 25 GB per second.

A **controller** is a collection of electronics that can operate a port, a bus, or a device. A serial-port controller is a simple device controller. It is a single chip (or portion of a chip) in the computer that controls the signals on the wires of a serial port. By contrast, a SCSI bus controller is not simple. Because the SCSI protocol is complex, the SCSI bus controller is often implemented as a separate circuit board (or a **host adapter**) that plugs into the computer. It typically contains a processor, microcode, and some private memory to enable it to process the SCSI protocol messages. Some devices have their own built-in controllers. If you look at a disk drive, you will see a circuit board attached to one side. This board is the disk controller. It implements the disk side of the protocol for some kind of connection—SCSI or **Serial Advanced Technology Attachment (SATA)**, for instance. It has microcode and a processor to do many tasks, such as bad-sector mapping, prefetching, buffering, and caching.

How can the processor give commands and data to a controller to accomplish an I/O transfer? The short answer is that the controller has one or more registers for data and control signals. The processor communicates with the controller by reading and writing bit patterns in these registers. One way in which this communication can occur is through the use of special I/O instructions that specify the transfer of a byte or word to an I/O port address. The I/O instruction triggers bus lines to select the proper device and to move bits into or out of a device register. Alternatively, the device controller can support **memory-mapped I/O**. In this case, the device-control registers are mapped into the address space of the processor. The CPU executes I/O requests using the standard data-transfer instructions to read and write the device-control registers at their mapped locations in physical memory. Some systems use both techniques. For instance, PCs use I/O instructions to control some devices and memory-mapped I/O to control others.

The graphics controller has I/O ports for basic control operations, but the controller has a large memory mapped region to hold screen contents. The process sends output to the screen by writing data into the memory-mapped region. The controller generates the screen image based on the contents of this memory. This technique is simple to use. Moreover, writing millions of bytes to the graphics memory is faster than issuing millions of I/O instructions. But the ease of writing to a memory-mapped I/O controller is offset by a disadvantage. Because a common type of software fault is a write through an incorrect pointer to an unintended region of memory, a memory-mapped device register is vulnerable to accidental modification. Of course, protected memory helps to reduce this risk. An I/O port typically consists of four registers, called the status, control, data-in, and data-out registers.

- The **data-in register** is read by the host to get input.
- The **data-out register** is written by the host to send output.
- The **status register** contains bits that can be read by the host. These bits indicate states, such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether a device error has occurred.
- The **control register** can be written by the host to start a command or to change the mode of a device. For instance, a certain bit in the control register of a serial port chooses between full-duplex and half-duplex communication, another bit enables parity checking, a third bit sets the word length to 7 or 8 bits, and other bits select one of the speeds supported by the serial port.

- The data registers are typically 1 to 4 bytes in size. Some controllers have FIFO chips that can hold several bytes of input or output data to expand the capacity of the controller beyond the size of the data register. A FIFO chip can hold a small burst of data until the device or host is able to receive those data.

Polling:
The complete protocol for interaction between the host and a controller can be intricate, but the basic handshaking notion is simple. We explain handshaking with an example. Assume that 2 bits are used to coordinate the producer–consumer relationship between the controller and the host. The controller indicates its state through the busy bit in the status register. (Recall that to ***set*** a bit means to write a 1 into the bit and to ***clear*** a bit means to write a 0 into it.) The controller sets the busy bit when it is busy working and clears the busy bit when it is ready to accept the next command. The host signals its wishes via the command-ready bit in the command register. The host sets the command-ready bit when a command is available for the controller to execute.

For this example, the host writes output through a port, coordinating with the controller by handshaking as follows.

1. The host repeatedly reads the busy bit until that bit becomes clear.
2. The host sets the write bit in the command register and writes a byte into the data-out register.
3. The host sets the command-ready bit.
4. When the controller notices that the command-ready bit is set, it sets the busy bit.
5. The controller reads the command register and sees the write command. It reads the data-out register to get the byte and does the I/O to the device.
6. The controller clears the command-ready bit, clears the error bit in the status register to indicate that the device I/O succeeded, and clears the busy bit to indicate that it is finished.
   This loop is repeated for each byte.

In step 1, the host is **busy-waiting** or **polling**: it is in a loop, reading the status register over and over until the busy bit becomes clear. If the controller and device are fast, this method is a reasonable one. But if the wait may be long, the host should probably switch to another task. How, then, does the host know when the controller has become idle? For some devices, the host must service the device quickly, or data will be lost. For instance, when data are streaming in on a serial port or from a keyboard, the small buffer on the controller will overflow and data will be lost if the host waits too long before returning to read the bytes.

In many computer architectures, three CPU-instruction cycles are sufficient to poll a device: read a device register, logical--and to extract a status bit, and branch if not zero. Clearly, the basic polling operation is efficient. But polling becomes inefficient when it is attempted repeatedly yet rarely finds a device ready for service, while other useful CPU processing remains undone. In such instances, it may be more efficient to arrange for the hardware controller to notify the CPU when the device becomes ready for service, rather than to require the CPU to poll repeatedly for an I/O completion. The hardware mechanism that enables a device to notify the CPU is called an **interrupt**.

Interrupts:

The basic interrupt mechanism works as follows. The CPU hardware has a wire called the **interrupt request line** that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt-request line, the CPU performs a state save and jumps to the **interrupt-handler routine** at a fixed address in memory. The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a return from interrupt instruction to return the CPU to the execution state prior to the interrupt. We say that the device controller ***raises*** an interrupt by asserting a signal on the interrupt request line, the CPU ***catches*** the interrupt and ***dispatches*** it to the interrupt handler, and the handler ***clears*** the interrupt by servicing the device.

We stress interrupt management in this chapter because even single-user modern systems manage hundreds of interrupts per second and servers hundreds of thousands per second.
The basic interrupt mechanism just described enables the CPU to respond to an asynchronous event, as when a device controller becomes ready for service. In a modern operating system, however, we need more sophisticated interrupt-handling features.

1. We need the ability to defer interrupt handling during critical processing.
2. We need an efficient way to dispatch to the proper interrupt handler for a device without first polling all the devices to see which one raised the interrupt.
3. We need multilevel interrupts, so that the operating system can distinguish between high- and low-priority interrupts and can respond with the appropriate degree of urgency. In modern computer hardware, these three features are provided by the CPU and by the **interrupt-controller hardware**. Most CPUs have two interrupt request lines. One is the **nonmaskable interrupt**, which is reserved for events such as unrecoverable memory errors.

The second interrupt line is **maskable**: it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. The maskable interrupt is used by device controllers to request service. The interrupt mechanism accepts an **address**—a number that selects a specific interrupt-handling routine from a small set. In most architecture, this address is an offset in a table called the **interrupt vector**. This vector contains the memory addresses of specialized interrupt handlers. The purpose of a vectored interrupt mechanism is to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service. In practice, however, computers have more devices (and, hence, interrupt handlers) than they have address elements in the interrupt vector. A common way to solve this problem is to use **interrupt chaining**, in which each element in the interrupt vector points to the head of a list of interrupt handlers. When an interrupt is raised, the handlers on the corresponding list are called one by one, until one is found that can service the request. This structure is a compromise between the overhead of a huge interrupt table and the inefficiency of dispatching to a single interrupt handler. Figure 13.4 illustrates the design of the interrupt vector for the Intel Pentium processor. The events from 0 to 31, which are nonmaskable, are used to signal various error conditions. The events from 32 to 255, which are maskable, are used for purposes such as device-generated interrupts.

The interrupt mechanism also implements a system of **interrupt priority levels**. These levels enable the CPU to defer the handling of low-priority interrupts without masking all interrupts and makes it possible for a high priority interrupt to preempt the execution of a low-priority interrupt.

A modern operating system interacts with the interrupt mechanism in several ways. At boot time, the operating system probes the hardware buses to determine what devices are present and installs the corresponding interrupt handlers into the interrupt vector. During I/O, the various device controllers raise interrupts when they are ready for service. These interrupts signify that output has completed, or that input data are available, or that a failure has been detected. The interrupt mechanism is also used to handle a wide variety of **exceptions**, such as dividing by 0, accessing a protected or nonexistent memory address, or attempting to execute a privileged instruction from user mode. The events that trigger interrupts have a common property: they are occurrences that induce the operating system to execute an urgent, self-contained routine.

An operating system has other good uses for an efficient hardware and software mechanism that saves a small amount of processor state and then calls a privileged routine in the kernel. For example, many operating systems use the interrupt mechanism for virtual memory paging. A page fault is an exception that raises an interrupt. The interrupt suspends the current process and jumps to the page-fault handler in the kernel. This handler saves the state of the process, moves the process to the wait queue, performs page-cache management, schedules an I/O operation to fetch the page, schedules another process to resume execution, and then returns from the interrupt. Another example is found in the implementation of system calls. Usually, a program uses library calls to issue system calls. The library routines check the arguments given by the application, build a data structure to convey the arguments to the kernel, and then execute a special instruction called a **software interrupt**, or **trap**. This instruction has an operand that identifies the desired kernel service. When a process executes the trap instruction, the interrupt hardware saves the state of the user code, switches to kernel mode, and dispatches to the kernel routine that implements the requested service. The trap is given a relatively low interrupt priority compared with those assigned to device interrupts—executing a system call on behalf of an application is less urgent than servicing a device controller before its FIFO queue overflows and loses data. Interrupts can also be used to manage the flow of control within the kernel. For example, consider one example of the processing required to complete a disk read. One step is to copy data from kernel space to the user buffer. This copying is time consuming but not urgent—it should not block other high-priority interrupt handling. Another step is to start the next pending I/O for that disk drive. This step has higher priority. If the disks are to be used efficiently, we need to start the next I/O as soon as the previous one completes.

Consequently, a pair of interrupt handlers implements the kernel code that completes a disk read. The high-priority handler records the I/O status, clears the device interrupt, starts the next pending I/O, and raises a low-priority interrupt to complete the work. Later, when the CPU is not occupied with high priority work, the low-priority interrupt will be dispatched. The corresponding handler completes the user-level I/O by copying data from kernel buffers to the application space and then calling the scheduler to place the application on the ready queue. A threaded kernel architecture is well suited to implement multiple interrupt priorities and to enforce the precedence of interrupt handling over background processing in kernel and application routines. We illustrate this point with the Solaris kernel. In Solaris, interrupt handlers are executed as kernel threads. A range of high priorities is reserved for these threads. These priorities give interrupt handlers precedence over application code and kernel housekeeping and implement the priority relationships among interrupt handlers. The priorities cause the Solaris thread scheduler to preempt low priority interrupt handlers in favor of higher-priority ones, and the threaded implementation enables multiprocessor hardware to run several interrupt handlers concurrently.

In summary, interrupts are used throughout modern operating systems to handle asynchronous events and to trap to supervisor-mode routines in the kernel. To enable the most urgent work to be done first, modern computers use a system of interrupt priorities. Device controllers, hardware faults, and system calls all raise interrupts to trigger kernel routines. Because interrupts are used so heavily for time-sensitive processing, efficient interrupt handling is required for good system performance.

Direct Memory Access

For a device that does large transfers, such as a disk drive, it seems wasteful to use an expensive general-purpose processor to watch status bits and to feed data into a controller register one byte at a time—a process termed **programmed I/O (PIO)**. Many computers avoid burdening the main CPU with PIO by offloading some of this work to a special-purpose processor called a **direct-memory access (DMA)** controller. To initiate a DMA transfer, the host writes a DMA command block into memory. This block contains a pointer to the source of a transfer, a pointer to the destination of the transfer, and a count of the number of bytes to be transferred. The CPU writes the address of this command block to the DMA controller, then goes on with other work. The DMA controller proceeds to operate the memory bus directly, placing addresses on the bus to perform transfers without the help of the main CPU. A simple DMA controller is a standard component in all modern computers, from smart phones to mainframes.

Handshaking between the DMA controller and the device controller is performed via a pair of wires called **DMA-request** and **DMA-acknowledge**. The device controller places a signal on the DMA request wire when a word of data is available for transfer. This signal causes the DMA controller to seize the memory bus, place the desired address on the memory-address wires, and place a signal on the DMA-acknowledge wire. When the device controller receives the

DMA-acknowledge signal, it transfers the word of data to memory and removes the DMA-request signal. When the entire transfer is finished, the DMA controller interrupts the CPU. This process is depicted in Figure 13.5. When the DMA controller seizes the memory bus, the CPU is momentarily prevented from accessing main memory, although it can still access data items in its primary and secondary caches.

Although this **cycle stealing** can slow down the CPU computation, offloading the data-transfer work to a DMA controller generally improves the total system performance. Some computer architectures use physical memory addresses for DMA, but others perform **direct virtual memory access (DVMA)**, using virtual addresses that undergo translation to physical addresses. DVMA can perform a transfer between two memory-mapped devices without the intervention of the CPU or the use of main memory. On protected-mode kernels, the operating system generally prevents processes from issuing device commands directly. This discipline protects data from access-control violations and also protects the system from erroneous use of device controllers that could cause a system crash. Instead, the operating system exports functions that a sufficiently privileged process can use to access low-level operations on the underlying hardware. On kernels without memory protection, processes can access device controllers directly. This direct access can be used to achieve high performance, since it can avoid kernel communication, context switches, and layers of kernel software. Unfortunately, it interferes with system security and stability. The trend in general-purpose operating systems is to protect memory and devices so that the system can try to guard against erroneous or malicious applications.

CSE 325: Operating Systems

Lecture 2: OS Evaluation

2.1 Bare Machine

Bare machines, do not have any OS support, were locked away in large, specially air-conditioned computer rooms, with staffs of professional operators to run them. Only large corporations or major government agencies or universities could afford the multimillion-dollar price tag. To run a job (i.e., a program or set of programs), a programmer would first write the program on paper (in FORTRAN or assembler), then punch it on cards. He would then bring the card deck down to the input room and hand it to one of the operators and go drink coffee until the output was ready.

When the computer finished whatever job it was currently running, an operator would go over to the printer and tear off the output and carry it over to the output room, so that the programmer could collect it later. Then he would take one of the card decks that had been brought from the input room and read it in. If the FORTRAN compiler was needed, the operator would have to get it from a file cabinet and read it in. Much computer time was wasted while operators were walking around the machine room.

2.2 Batch Processing

Given the high cost of the equipment, it is not surprising that people quickly looked for ways to reduce the wasted time. The solution generally adopted was the batch system. The idea behind it was to collect a tray full of jobs in the input room and then read them onto a magnetic tape using a small (relatively) inexpensive computer, such as the IBM 1401, which was quite good at reading cards, copying tapes, and printing output, but not at all good at numerical calculations. Other, much more expensive machines, such as the IBM 7094, were used for the real computing.

After collecting a batch of jobs, nearly takes about an hour, the cards were read onto a magnetic tape, which was carried into the machine room, where it was mounted on a tape drive. The operator then loaded a special program (the ancestor of today's operating system), which read the first job from tape and ran it. The output was written onto a second tape, instead of being printed. After each job finished, the operating system automatically read the next job from the tape and began running it. When the whole batch was done, the operator removed the input and output tapes, replaced the input tape with the next batch, and brought the output tape to a 1401 for printing off line (i.e., not connected to the main computer).

The system is a offline procedure and it took long time up to 2 days to run several programs, debugging was nearly impossible. It maintained low CPU utilization as CPU was idle during the input condition. Input, processing and output operations were sequentially done. No overlap between their executions.

To make the system more efficient and increase the CPU utilization, peripheral devices were connected, did online processing and supported Spooling (*simultaneous peripheral operations on-line*) technique. Magnetic tape was removed; random access devices (Hard disk) were arrived and pooled (job-queue) input directly to Hard disk, use it as a large storage device for reading as many input files as possible, processing unit or CPU took those inputs from disk, processed them and there after stored (in disk) output files until output devices were ready to accept them without manual interference. OS program was located in memory, and its task was simple and transferred control from one job to another and increased CPU utilization. The system allowed overlapping between I/O and CPU processing.

2.3 Multiprogramming

One of the most important aspects of operating systems is the ability to multiprogram. A single program cannot, in general, keep either the CPU or the I/O devices busy at all times. Single users frequently have multiple programs running. Multiprogramming increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute. The idea is as follows: The operating system keeps several jobs in memory simultaneously (Figure 1.9). Since, in general, main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the job pool. This pool consists of all processes residing on disk awaiting allocation of main memory.

The set of jobs in memory can be a subset of the jobs kept in the job pool. The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete. In a non-multiprogrammed system, the CPU would sit idle. In a multiprogrammed system, the operating system simply switches to, and executes, another job. When that job needs to wait, the CPU switches to another job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle.

The system dramatically reduced the processing time and built debugging opportunity. CPU processing was much faster than input processing and put CPU in idleness during input processing. A faster memory RAM was added nearby to CPU. During the CPU processing, inputted job was placed from disk to RAM so that CPU can collect next job from RAM rather than disk. It reduced several times. Multiple jobs were place in RAM with memory protection techniques for further faster processing. However, CPU used FCFS (First Come First Serve) scheduling to dispatch jobs from RAM and it creates convoy effect. Due to the convoy effect, the resource utilization was tremendously reduced.

2.4 Timesharing

Multiprogrammed systems provide an environment in which the various system resources (for example, CPU, memory, and peripheral devices) are utilized effectively, but they do not provide for user interaction with the computer system. Time sharing (or multitasking) is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.

Time sharing requires an interactive computer system, which provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly, using a input device such as a keyboard, mouse, touch pad, or touch screen, and waits for immediate results on an output device. Accordingly, the response time should be short—typically less than one second.

A time-shared operating system allows many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use, even though it is being shared among many users.

2.5 Virtualization