Operating System
CS604

Delivered by

Dr. Syed Mansoor Sarwar

Virtual University of Pakistan

Knowledge beyond the boundaries

TABLE OF CONTENTS

Operating Systems
Lecture No. 1

Reading Material
- Operating Systems Concepts, Chapter 1
- PowerPoint Slides for Lecture 1

Summary
- Introduction and purpose of the course
- Organization of a computer system
- Purpose of a computer system
- Requirements for achieving the purpose – Setting the stage for OS concepts and principles
- Outline of topics to be discussed
- What is an Operating System?

Organization of a Computer System
As shown in Figure 1.1, the major high-level components of a computer system are:

1. **Hardware**, which provides basic computing resources (CPU, memory, I/O devices).
2. **Operating system**, which manages the use of the hardware among the various application programs for the various users and provides the user a relatively simple machine to use.
3. **Applications programs** that define the ways in which system resources are used to solve the computing problems of the users (compilers, database systems, video games, business programs).
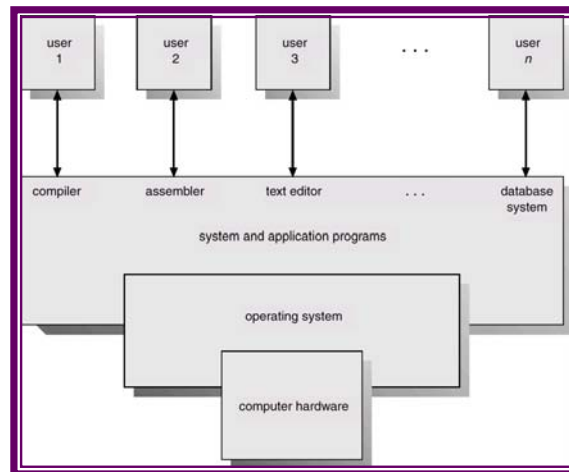4. **Users,** which include people, machines, other computers.

Figure 1.1. High-level components of a computer system

Purpose of a Computer—Setting the Stage for OS Concepts and Principles

Computer systems consist of software and hardware that are combined to provide a tool to implement solutions for specific problems in an efficient manner and to execute programs. Figure 1.2 shows the general organization of a contemporary computer system and how various system components are interconnected.
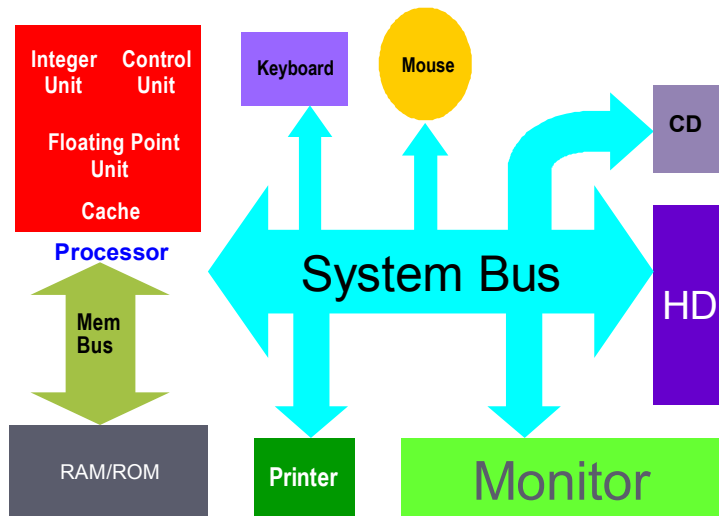
Figure 1.2. Organization of a Computer System

Viewing things closely will reveal that the primary purpose of a computer system is to generate executable programs and execute them. The following are some of the main issues involved in performing these tasks.

1. Storing an executable on a secondary storage device such as hard disk
2. Loading executable from disk into the main memory
3. Setting the CPU state appropriately so that program execution could begin
4. Creating multiple cooperating processes, synchronizing their access to shared data, and allowing them to communicate with each other

The above issues require the operating system to provide the following services and much more:

- Manage secondary storage devices
  - ➢ Allocate appropriate amount of disk space when files are created
  - ➢ Deallocate space when files are removing
  - ➢ Insure that a new file does not overwrite an existing file
  - ➢ Schedule disk requests
- Manage primary storage
  - ➢ Allocate appropriate amount of memory space when programs are to be loaded into the memory for executing
  - ➢ Deallocate space when processes terminate
  - ➢ Insure that a new process is not loaded on top of an existing process
  - ➢ Insure that a process does not access memory space that does not belong to it
  - ➢ Minimize the amount of unused memory space
  - ➢ Allow execution of programs larger in size than the available main memory
- Manage processes

- ➢ Allow simultaneous execution of processes by scheduling the CPU(s)
- ➢ Prevent deadlocks between processes
- ➢ Insure integrity of shared data
- ➢ Synchronize executions of cooperating processes
- ▪ Allow a user to manage his/her files and directories properly
  - ➢ User view of directory structure
  - ➢ Provide a mechanism that allows users to protect their files and directories

In this course, we will discuss in detail these operating system services (and more), with a particular emphasis on the UNIX and Linux operating systems. See the course outline for details of topics and lecture schedule.

What is an Operating System?

There are two views about this. The top-down view is that it is a program that acts as an intermediary between a user of a computer and the computer hardware, and makes the computer system convenient to use. It is because of the operating system that users of a computer system don't have to deal with computer's hardware to get their work done. Users can use simple commands to perform various tasks and let the operating system do the difficult work of interacting with computer hardware. Thus, you can use a command like `copy file1 file2` to copy 'file1' to 'file2' and let the operating system communicate with the controller(s) of the disk that contain(s) the two files.

A computer system has many hardware and software resources that may be required to solve a problem: CPU time, memory space, file storage space, I/O devices etc. The operating system acts as the manager of these resources, facing numerous and possibly conflicting requests for resources, the operating system must decide how (and when) to allocate (and deallocate) them to specific programs and users so that it can operate the computer system efficiently, fairly, and securely. So, the bottom-up view is that operating system is a resource manager who manages the hardware and software resources in the computer system.

A slightly different view of an operating system emphasizes the need to control the various I/O devices and programs. An operating system is a control program that manages the execution of user programs to prevent errors and improper use of a computer.

Operating Systems
Lecture No. 2

Reading Material
- Operating Systems Concepts, Chapter 1
- PowerPoint Slides for Lecture 2

Summary
- Single-user systems
- Batch systems
- Multi programmed systems
- Time-sharing systems
- Real time systems
- Interrupts, traps and software interrupts (UNIX signals)
- Hardware protection

Single-user systems

A computer system that allows only one user to use the computer at a given time is known as a **single-user system**. The goals of such systems are maximizing user convenience and responsiveness, instead of maximizing the utilization of the CPU and peripheral devices. Single-user systems use I/O devices such as keyboards, mice, display screens, scanners, and small printers. They can adopt technology developed for larger operating systems. Often individuals have sole use of computer and do not need advanced CPU utilization and hardware protection features. They may run different types of operating systems, including DOS, Windows, and MacOS. Linux and UNIX operating systems can also be run in single-user mode.

Batch Systems

Early computers were large machines run from a console with card readers and tape drives as input devices and line printers, tape drives, and card punches as output devices. The user did not interact directly with the system; instead the user prepared a job, (which consisted of the program, data, and some control information about the nature of the job in the form of control cards) and submitted this to the computer operator. The job was in the form of punch cards, and at some later time the output was generated by the system—user didn't get to interact with his/her job. The output consisted of the result of the program, as well as a dump of the final memory and register contents for debugging.

To speed up processing, operators batched together jobs with similar needs, and ran them through the computer as a group. For example, all FORTRAN programs were complied one after the other. The major task of such an operating system was to transfer control automatically from one job to the next. In this execution environment, the CPU is often idle because the speeds of the mechanical I/O devices such as a tape drive are slower than that of electronic devices. Such systems in which the user does not get to

interact with his/her jobs and jobs with similar needs are executed in a "batch", one after the other, are known as **batch systems**. Digital Equipment Corporation's VMS is an example of a batch operating system.

Figure 2.1 shows the memory layout of a typical computer system, with the **system space** containing operating system code and data currently in use and the **user space** containing user programs (processes). In case of a batch system, the user space contains one process at a time because only one process is executing at a given time.
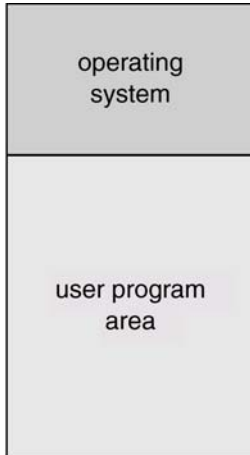
Figure 2.1 Memory partitioned into user and system spaces

Multi-programmed Systems

Multi-programming increases CPU utilization by organizing jobs so that the CPU always has one to execute. The operating system keeps several jobs in memory simultaneously, as shown in Figure 2.2. This set of jobs is a subset of the jobs on the disk which are ready to run but cannot be loaded into memory due to lack of space. Since the number of jobs that can be kept simultaneously in memory is usually much smaller than the number of jobs that can be in the job pool; the operating system picks and executes one of the jobs in the memory. Eventually the job has to wait for some task such as an I/O operation to complete. In a non multi-programmed system, the CPU would sit idle. In a multi-programmed system, the operating system simply switches to, and executes another job. When that job needs to wait, the CPU simply switches to another job and so on.

Figure 2.2 Memory layout for a multi-programmed batch system

Figure 2.3 illustrates the concept of multiprogramming by using an example system with two processes, P1 and P2. The CPU is switched from P1 to P2 when P1 finishes its CPU burst and needs to wait for an event, and vice versa when P2 finishes it CPU burst and has to wait for an event. This means that when one process is using the CPU, the other is waiting for an event (such as I/O to complete). This increases the utilization of the CPU and I/O devices as well as throughput of the system. In our example below, P1 and P2 would finish their execution in 10 time units if no multiprogramming is used and in six time units if multiprogramming is used.
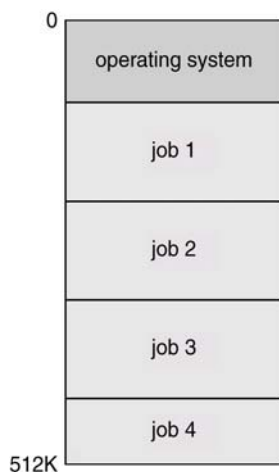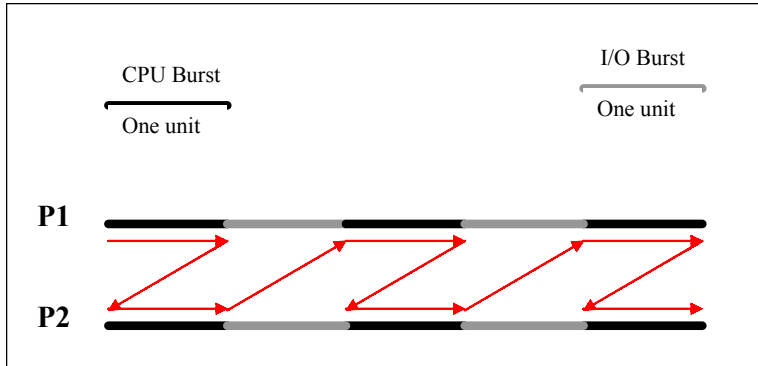

Figure 2.3 Illustration of the multiprogramming concept

All jobs that enter the system are kept in the **job pool**. This pool consists of all processes residing on disk awaiting allocation of main memory. If several jobs are ready to be brought into memory, and there is not enough room for all of them, then the system must choose among them. This decision is called *job scheduling*. In addition if several jobs are ready to run at the same time, the system must choose among them. We will discuss CPU scheduling in Chapter 6.

Time-sharing systems

A **time-sharing system** is multi-user, multi-process, and interactive system. This means that it allows multiple users to use the computer simultaneously. A user can run one or more processes at the same time and interact with his/her processes. A time-shared system uses multiprogramming and CPU scheduling to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory. To obtain a reasonable response time, jobs may have to be swapped in and out of main memory. UNIX, Linux, Widows NT server, and Windows 2000 server are time-sharing systems. We will discuss various elements of time-sharing systems throughout the course.

Real time systems

Real time systems are used when rigid time requirements are placed on the operation of a processor or the flow of data; thus it is often used as a control device in a dedicated application. Examples are systems that control scientific experiments, medical imaging systems, industrial control systems and certain display systems.

A **real time system** has well defined, fixed time constraints, and if the system does not produce output for an input within the time constraints, the system will fail. For instance, it would not do for a robot arm to be instructed to halt after it had smashed into the car it was building.

Real time systems come in two flavors: hard and soft. A **hard real time system** guarantees that critical tasks be completed on time. This goal requires that all delays in the system be completed on time. This goal requires that all delays in the system be bounded, from the retrieval of stored data to the time it takes the operating system to finish any request made of it. Secondary storage of any sort is usually limited or missing, with data instead being stored in short-term memory or in read only memory. Most advanced operating system features are absent too, since they tend to separate the user from the hardware, and that separation results in uncertainty about the amount of time an operation will take.

A less restrictive type of real time system is a **soft real time system**, where a critical real-time task gets priority over other tasks, and retains that priority until it completes. As in hard real time systems, the operating system kernel delays need to be bounded. Soft real time is an achievable goal that can be mixed with other types of systems, whereas hard real time systems conflict with the operation of other systems such as time-sharing systems, and the two cannot be mixed.

Interrupts, traps and software interrupts

An **interrupt** is a signal generated by a hardware device (usually an I/O device) to get CPU's attention. Interrupt transfers control to the **interrupt service routine** (ISR), generally through the **interrupt vector table**, which contains the addresses of all the service routines. The interrupt service routine executes; on completion the CPU resumes the interrupted computation. Interrupt architecture must save the address of the interrupted instruction. Incoming interrupts are disabled while another interrupt is being processed to prevent a *lost interrupt*. An operating system is an interrupt driven software.

A **trap** (or an *exception)* is a software-generated interrupt caused either by an error (division by zero or invalid memory access) or by a user request for an operating system service.

A **signal** is an event generated to get attention of a process. An example of a signal is the event that is generated when you run a program and then press `<Ctrl-C>`. The signal generated in this case is called SIGINT (Interrupt signal). Three actions are possible on a signal:

1. Kernel-defined default action—which usually results in process termination and, in some cases, generation of a 'core' file that can be used the programmer/user to know the state of the process at the time of its termination.
2. Process can intercept the signal and ignore it.
3. Process can intercept the signal and take a programmer-defined action.

We will discuss signals in detail in some of the subsequent lectures.

Hardware Protection

Multi-programming put several programs in memory at the same time; while this increased system utilization it also increased problems. With sharing, many processes

could be adversely affected by a bug in one program. One erroneous program could also modify the program or data of another program or even the resident part of the operating system. A file may overwrite another file or folder on disk. A process may get the CPU and never relinquish it. So the issues of hardware protection are: I/O protection, memory protection, and CPU protection. We will discuss them one by one, but first we talk about the dual-mode operation of a CPU.

a) Dual Mode Operation

To ensure proper operation, we must protect the operating system and all other programs and their data from any malfunctioning program. Protection is needed for any shared resources. Instruction set of a modern CPU has two kinds of instructions, privileged instructions and non-privileged instructions. Privileged instructions can be used to perform hardware operations that a normal user process should not be able to perform, such as communicating with I/O devices. If a user process tries to execute a privileged instruction, a trap should be generated and process should be terminated prematurely. At the same time, a piece of operating system code should be allowed to execute privileged instructions. In order for the CPU to be able to differentiate between a user process and an operating system code, we need two separate modes of operation: user mode and monitor mode (also called supervisor mode, system mode, or privileged mode). A bit, called the **mode bit**, is added to the hardware of the computer to indicate the current mode: **monitor mode** (0) or **user mode** (1). With the mode bit we are able to distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user.

The concept of privileged instructions also provides us with the means for the user to interact with the operating system by asking it to perform some designated tasks that only the operating system should do. A user process can request the operating system to perform such tasks for it by executing a **system call**. Whenever a system call is made or an interrupt, trap, or signal is generated, CPU mode is switched to system mode before the relevant kernel code executes. The CPU mode is switched back to user mode before the control is transferred back to the user process. This is illustrated by the diagram in Figure 2.4.
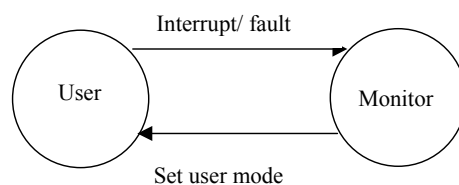
Figure 2.4 The dual-mode operation of the CPU

b) I/O Protection

A user process may disrupt the normal operation of the system by issuing illegal I/O instructions, by accessing memory locations within the operating system itself, or by

. We can use various mechanisms to ensure that such disruptions cannot take place in the system.

To prevent users from performing illegal I/O, we define all I/O instructions to be privileged instructions. Thus users cannot issue I/O instructions directly; they must do it through the operating system. For I/O protection to be complete, we must be sure that a user program can never gain control of the computer in monitor mode. If it could, I/O protection could be compromised.

Consider a computer executing in user mode. It will switch to monitor mode whenever an interrupt or trap occurs, jumping to the address determined from the interrupt from the interrupt vector. If a user program, as part of its execution, stores a new address in the interrupt vector, this new address could overwrite the previous address with an address in the user program. Then, when a corresponding trap or interrupt occurred, the hardware would switch to monitor mode and transfer control through the modified interrupt vector table to a user program, causing it to gain control of the computer in monitor mode. Hence we need all I/O instructions and instructions for changing the contents of the system space in memory to be protected. A user process could request a privileged operation by executing a system call such as read (for reading a file).

Operating Systems
Lecture No. 3

Reading Material
  ▪ Computer System Structures, Chapter 2
  ▪ Operating Systems Structures, Chapter 3
  ▪ PowerPoint Slides for Lecture 3

Summary
  ▪ Memory and CPU protection
  ▪ Operating system components and services
  ▪ System calls
  ▪ Operating system structures

Memory Protection

The region in the memory that a process is allowed to access is known as **process address space**. To ensure correct operation of a computer system, we need to ensure that a process cannot access memory outside its address space. If we don't do this then a process may, accidentally or deliberately, overwrite the address space of another process or memory space belonging to the operating system (e.g., for the interrupt vector table).

Using two CPU registers, specifically designed for this purpose, can provide memory protection. These registered are:
  ▪ Base register – it holds the smallest legal physical memory address for a process
  ▪ Limit register – it contains the size of the process

When a process is loaded into memory, the base register is initialized with the starting address of the process and the limit register is initialized with its size. Memory outside the defined range is protected because the CPU checks that every address generated by the process falls within the memory range defined by the values stored in the base and limit registers, as shown in Figure 3.1.
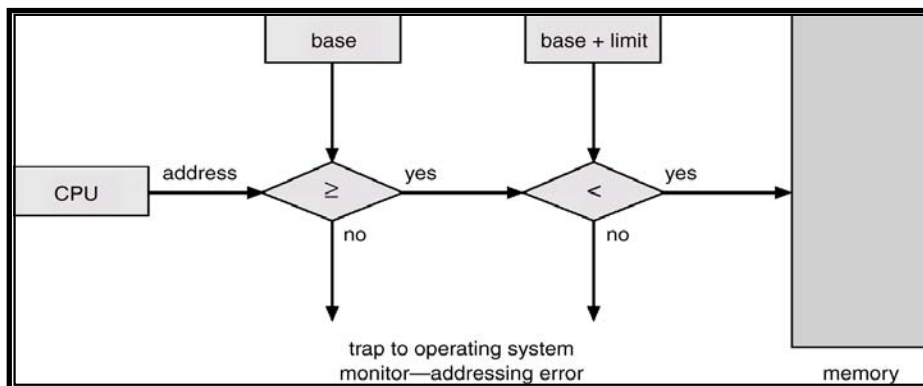

Figure 3.1 Hardware address protection with base and limit registers

In Figure 3.2, we use an example to illustrate how the concept outlined above works. The base and limit registers are initialized to define the address space of a process. The process starts at memory location 300040 and its size is 120900 bytes (assuming that memory is byte addressable). During the execution of this process, the CPU insures (by using the logic outlined in Figure 3.1) that all the addresses generated by this process are greater than or equal to 300040 and less than (300040+120900), thereby preventing this process to access any memory area outside its address space. Loading the base and limit registers are privileged instructions.
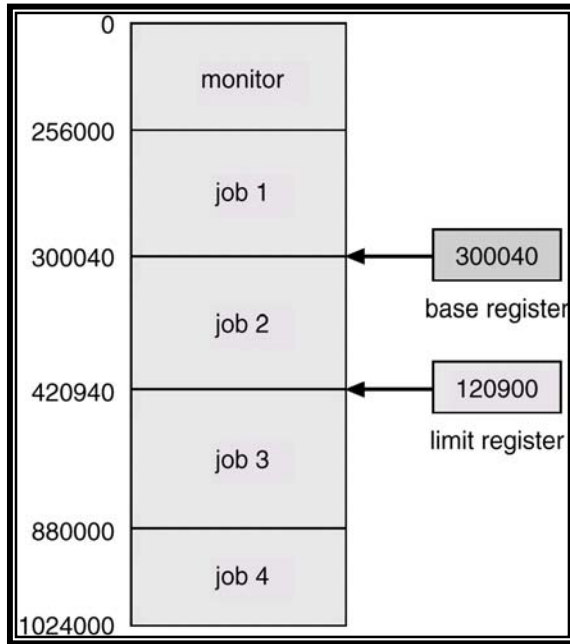
Figure 3.2 Use of Base and Limit Register

CPU Protection

In addition to protecting I/O and memory, we must ensure that the operating system maintains control. We must prevent the user program from getting stuck in an infinite loop or not calling system services and never returning control to the CPU. To accomplish this we can use a **timer**, which interrupts the CPU after specified period to ensure that the operating system maintains control. The timer period may be variable or fixed. A *fixed-rate clock* and a *counter* are used to implement a variable timer. The OS initializes the counter with a positive value. The counter is decremented every clock tick by the clock interrupt service routine. When the counter reaches the value 0, a timer interrupt is generated that transfers control from the current process to the next scheduled process. Thus we can use the timer to prevent a program from running too long. In the most straight forward case, the timer could be set to interrupt every N milliseconds, where N is the **time slice** that each process is allowed to execute before the next process gets control of the CPU. The OS is invoked at the end of each time slice to perform various housekeeping tasks. This issue is discussed in detail under CPU scheduling in Chapter 7.

Another use of the timer is to compute the current time. A timer interrupt signals the passage of some period, allowing the OS to compute the current time in reference to some initial time. Load-timer is a privileged instruction.

OS Components

An operating system has many components that manage all the resources in a computer system, insuring proper execution of programs. We briefly describe these components in this section.

■ Process management

A process can be thought of as a program in execution. It needs certain resources, including CPU time, memory, files and I/O devices to accomplish its tasks. The operating system is responsible for:

- Creating and terminating both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

■ Main memory management

Main memory is a large array of words or bytes (called memory locations), ranging in size from hundreds of thousands to billions. Every word or byte has its own address. Main memory is a repository of quickly accessible data shared by the CPU and I/O devices. It contains the code, data, stack, and other parts of a process. The central processor reads instructions of a process from main memory during the machine cycle—fetch-decode-execute.

The OS is responsible for the following activities in connection with memory management:

- Keeping track of free memory space
- Keeping track of which parts of memory are currently being used and by whom
- Deciding which processes are to be loaded into memory when memory space becomes available
- Deciding how much memory is to be allocated to a process
- Allocating and deallocating memory space as needed
- Insuring that a process is not overwritten on top of another

■ Secondary storage management

The main purpose of a computer system is to execute programs. The programs, along with the data they access, must be in the main memory or **primary storage** during their execution. Since main memory is too small to accommodate all data and programs, and because the data it holds are lost when the power is lost, the computer system must provide **secondary storage** to backup main memory. Most programs are stored on a disk until loaded into the memory and then use disk as both the source and destination of their processing. Like all other resources in a computer system, proper management of disk storage is important.

The operating system is responsible for the following activities in connection with disk management:

- Free-space management

- Storage allocation and deallocation
- Disk scheduling

■ I/O system management

The I/O subsystem consists of:
- A memory management component that includes buffering, caching and spooling
- A general device-driver interface
- Drivers for specific hardware devices

■ File management

Computers can store information on several types of physical media, e.g. magnetic tape, magnetic disk and optical disk. The OS maps files onto physical media and accesses these media via the storage devices.

The OS is responsible for the following activities with respect to file management:
- Creating and deleting files
- Creating and deleting directories
- Supporting primitives (operations) for manipulating files and directories
- Mapping files onto the secondary storage
- Backing up files on stable (nonvolatile) storage media

■ Protection system

If a computer system has multiple users and allows concurrent execution of multiple processes then the various processes must be protected from each other's activities.

Protection is any mechanism for controlling the access of programs, processes or users to the resources defined by a computer system.

■ Networking

A **distributed system** is a collection of processors that do not share memory, peripheral devices or a clock. Instead, each processor has it own local memory and clock, and the processors communicate with each other through various communication lines, such as high- speed buses or networks.

The processors in a communication system are connected through a **communication network**. The communication network design must consider message routing and connection strategies and the problems of contention and security.

A distributed system collects physically separate, possibly heterogeneous, systems into a single coherent system, providing the user with access to the various resources that the system maintains.

■ Command-line interpreter (shells)

One of the most important system programs for an operating system is the **command interpreter**, which is the interface between the user and operating system. Its purpose is to read user commands and try to execute them. Some operating systems include the command interpreter in the kernel. Other operating systems (e.g. UNIX, Linux, and DOS) treat it as a special program that runs when a job is initiated or when a user first logs on (on time sharing systems). This program is sometimes called the **command-line interpreter** and is often known as the **shell**. Its function is simple: to get the next command statement and execute it. Some of the famous shells for UNIX and Linux are

Operating System Services

An operating system provides the environment within which programs are executed. It provides certain services to programs and users of those programs, which vary from operating system to operating system. Some of the common ones are:

- **Program execution:** The system must be able to load a program into memory and to run that programs. The program must be able to end its execution.
- **I/O Operations:** A running program may require I/O, which may involve a file or an I/O device. For efficiency and protection user usually cannot control I/O devices directly. The OS provides a means to do I/O.
- **File System Manipulation:** Programs need to read, write files. Also they should be able to create and delete files by name.
- **Communications:** There are cases in which one program needs to exchange information with another process. This can occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network. Communication may be implemented via shared **memory** or **message passing**.
- **Error detection:** The OS constantly needs to be aware of possible errors. Error may occur in the CPU and memory hardware, in I/O devices and in the user program. For each type of error, the OS should take appropriate action to ensure correct and consistent computing.

  In order to assist the efficient operation of the system itself, the system provides the following functions:

- **Resource allocation:** When multiple users are logged on the system or multiple jobs are running at the same time, resources must be allocated to each of them. There are various routines to schedule jobs, allocate plotters, modems and other peripheral devices.
- **Accounting:** We want to keep track of which users use how many and which kinds of computer resources. This record keeping may be used for accounting or simply for accumulating usage statistics.
- **Protection:** The owners of information stored in a multi user computer system may want to control use of that information. When several disjointed processes execute concurrently it should not b possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled.

Entry Points into Kernel

As shown in Figure 3.3, there are four events that cause execution of a piece of code in the kernel. These events are: interrupt, trap, system call, and signal. In case of all of these events, some kernel code is executed to service the corresponding event. You have

discussed interrupts and traps in the computer organization or computer architecture course. We will discuss system calls execution in this lecture and signals subsequent lectures. We will talk about many UNIX and Linux system calls and signals throughout the course.
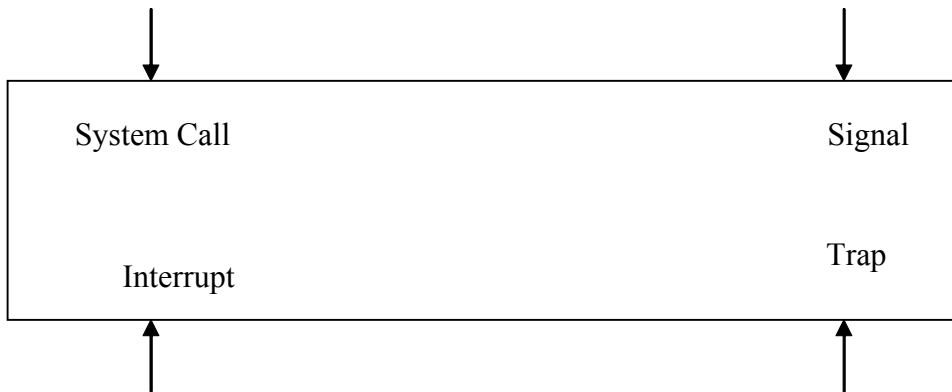
Figure 3.3 Entry points into the operating system kernel

System Calls

System calls provide the interface between a process and the OS. These calls are generally available as assembly language instructions. The system call interface layer contains entry point in the kernel code; because all system resources are managed by the kernel any user or application request that involves access to any system resource must be handled by the kernel code, but user process must not be given open access to the kernel code for security reasons. So that user processes can invoke the execution of kernel code, several openings into the kernel code, also called *system calls*, are provided. System calls allow processes and users to manipulate system resources such as files and processes.

System calls can be categorized into the following groups:
- Process Control
- File Management
- Device Management
- Information maintenance
- Communications

Semantics of System Call Execution

The following sequence of events takes place when a process invokes a system call:
- The user process makes a call to a library function
- The library routine puts appropriate parameters at a well-known place, like a register or on the stack. These parameters include arguments for the system call, return address, and call number. Three general methods are used to pass parameters between a running program and the operating system.
  - Pass parameters in *registers*.
  - Store the parameters in a table in the main memory and the table address is passed as a parameter in a register.
  - *Push* (store) the parameters onto the *stack* by the program, and *pop* off the stack by operating system.

- A `trap` instruction is executed to change mode from <mark>user to kernel</mark> and give control to operating system.
- The operating system then determines which system call is to be carried out by examining one of the parameters (the call number) passed to it by library routine.
- The kernel uses call number to index a kernel table (the **dispatch table)** which contains pointers to service routines for all system calls.
- The service routine is executed and control given back to user program via return from trap instruction; the instruction also changes mode from system to user.
- The library function executes the instruction following trap; interprets the return values from the kernel and returns to the user process.

Figure 3.4 gives a pictorial view of the above steps.
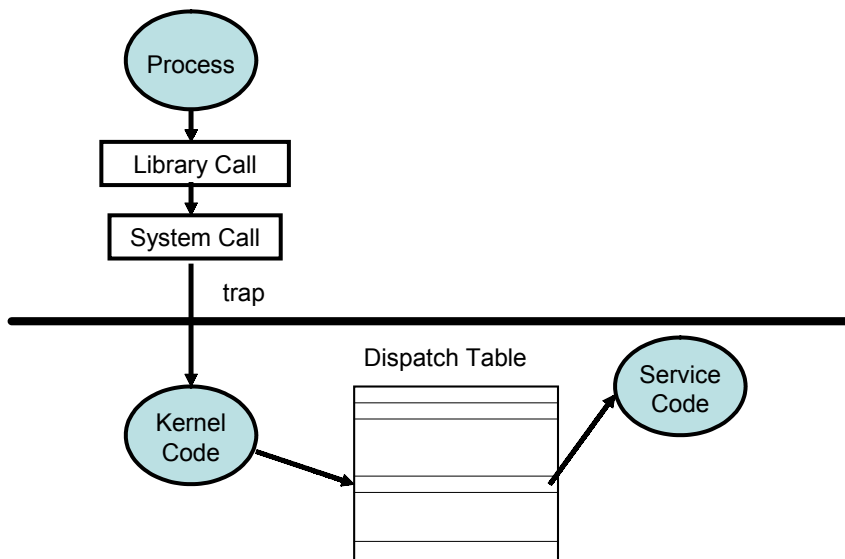
Figure 3.4 Pictorial view of the steps needed for execution of a system call

Operating Systems Structures

Just like any other software, the operating system code can be structured in different ways. The following are some of the commonly used structures.

- <mark>Simple/Monolithic Structure</mark>

<mark>In this case, the operating system code has not structure</mark>. It is written for functionality and efficiency (in terms of time and space). DOS and UNIX are examples of such systems, as shown in Figures 3.5 and 3.6. UNIX consists of two separable parts, the kernel and the system programs. The kernel is further separated into a series of interfaces and devices drivers, which were added and expanded over the years. Every thing below the system call interface and above the physical hardware is the kernel, which provides the file system, CPU scheduling, memory management and other OS functions through system calls. Since this is an enormous amount of functionality combined in one level, UNIX is difficult to enhance as changes in one section could adversely affect other areas. We will discuss the various components of the UNIX kernel throughout the course.
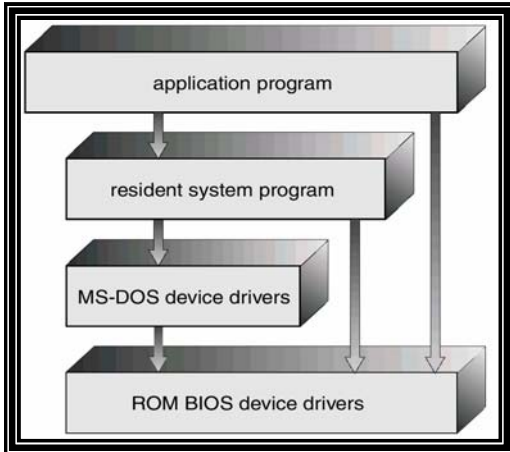
Figure 3.5 Logical structure of DOS
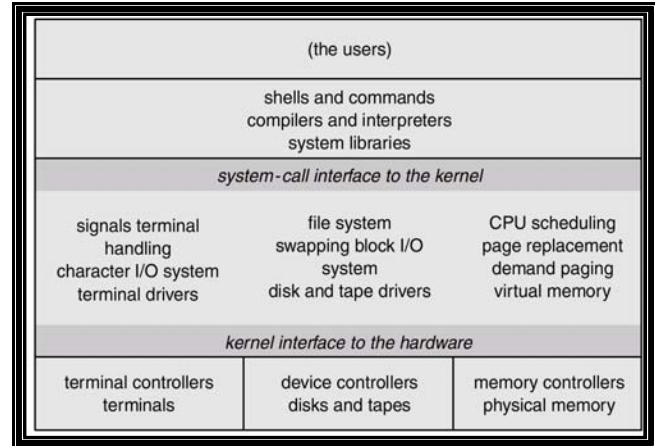
Figure 3.6 Logical structure of UNIX

Operating Systems
Lecture No. 4

Reading Material
- Operating Systems Structures, Chapter 3
- PowerPoint Slides for Lecture 3

Summary
- Operating system structures
- Operating system design and implementation
- UNIX/Linux directory structure
- Browsing UNIX/Linux directory structure

Operating Systems Structures (continued)

■ Layered Approach

The modularization of a system can be done in many ways. As shown in Figure 4.1, in the layered approach the OS is broken up into a number of layers or levels each built on top of lower layer. The bottom layer is the hardware; the highest layer (layer N) is the user interface. A typical OS layer (layer-M) consists of data structures and a set of routines that can be invoked by higher-level layers. Layer M in turn can invoke operations on lower level layers.
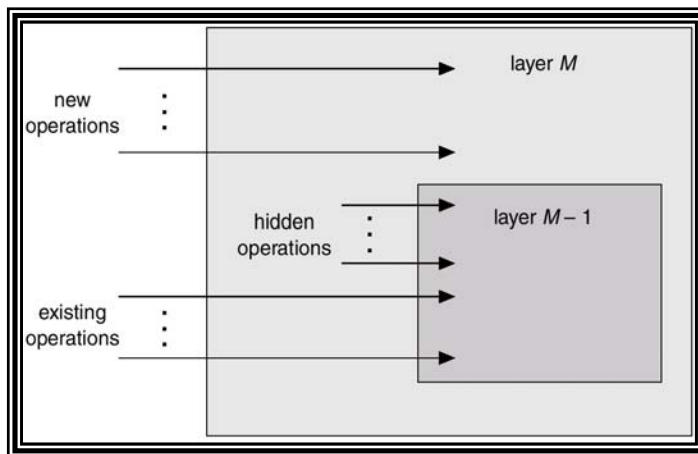
Figure 4.1 The layered structure

The main advantage of the layered approach is modularity. The layers are selected such that each uses functions and services of only lower layers. This approach simplifies debugging and system verification.

The major difficulty with layered approach is careful definition of layers, because a layer can only use the layers below it. Also it tends to be less efficient than other approaches. Each layer adds overhead to a system call (which is trapped when the

program executes a I/O operation, for instance). This results in a system call that takes longer than does one on a non-layered system. THE operating system by Dijkstra and IBM's OS/2 are examples of layered operating systems.

■ Micro kernels

This method structures the operating system by removing all non-essential components from the kernel and implementing as system and user level programs. The result is a smaller kernel. Micro kernels typically provide minimum process and memory management in addition to a communication facility. The main function of the micro kernel is to provide a communication facility between the client program and the various services that are also running in the user space.

The benefits of the micro kernel approach include the ease of extending the OS. All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer because the micro kernel is a smaller kernel. The resulting OS is easier to port from one hard ware design to another. It also provides more security and reliability since most services are running as user rather than kernel processes. Mach, MacOS X Server, QNX, OS/2, and Windows NT are examples of microkernel based operating systems. As shown in Figure 4.2, various types of services can be run on top of the Windows NT microkernel, thereby allowing applications developed for different platforms to run under Windows NT.
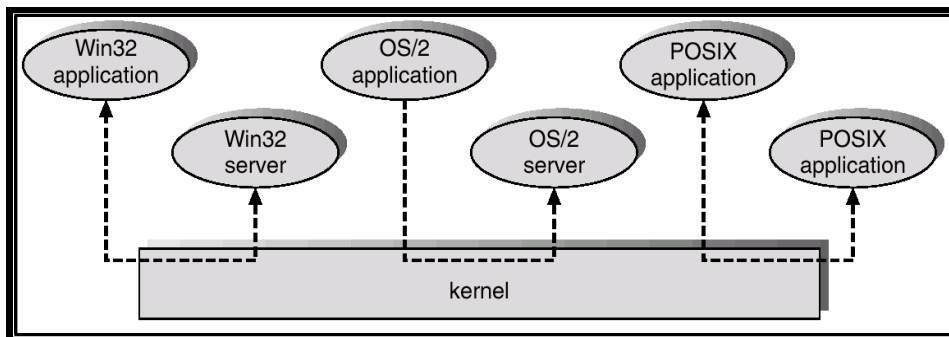
Figure 4.2 Windows NT client-server structure

■ Virtual Machines

Conceptually a computer system is made up of layers. The hardware is the lowest level in all such systems. The kernel running at the next level uses the hardware instructions to create a set of system call for use by outer layers. The system programs above the kernel are therefore able to use either system calls or hardware instructions and in some ways these programs do not differentiate between these two. System programs in turn treat the hardware and the system calls as though they were both at the same level. In some systems the application programs can call the system programs. The application programs view everything under them in the hierarchy as though the latter were part of the machine itself. This layered approach is taken to its logical conclusion in the concept of a **virtual machine** (VM). The VM operating system for IBM systems is the best example of VM concept.

By using CPU scheduling and virtual memory techniques an operating system can create the illusion that a process has its own memory with its own (virtual) memory. The

virtual machine approach on the other hand does not provide any additional functionality but rather provides an interface that is identical to the underlying bare hardware. Each process is provided with a virtual copy of the underlying computer. The physical computer shares resources to create the virtual machines. Figure 4.3 illustrates the concepts of virtual machines by a diagram.
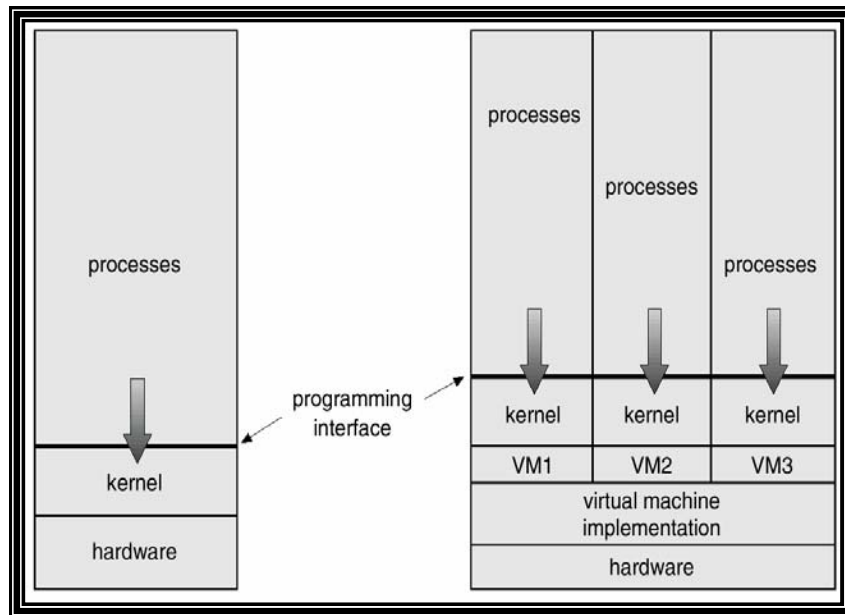
Non Virtual Machine   Virtual Machine

Figure 4.3 Illustration of virtual and non-virtual machines

Although the virtual machine concept is useful it is difficult to implement. There are two primary advantages to using virtual machines: first by completely protecting system resources the virtual machine provides a robust level of security. Second the virtual machine allows system development to be done without disrupting normal system operation.

Java Virtual Machine (**JVM**) loads, verifies, and executes programs that have been translated into Java Bytecode, as shown in Figure 4.4. **VMWare** can be run on a Windows platform to create a virtual machine on which you can install an operating of your choice, such as Linux. We have shown a couple of snapshots of VMWare on a Windows platform in the lecture slides. **Virtual PC** software works in a similar fashion.
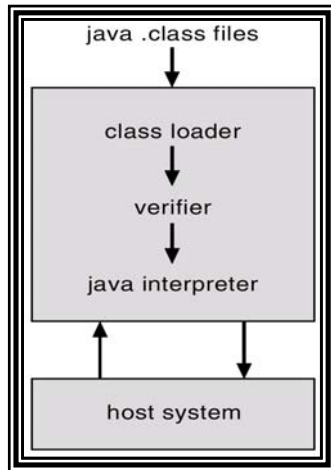
Figure 4.4 Java Virtual Machine

System Design and Implementation

■ Design Goals

At the highest level, the deign of the system will be affected by the choice of hardware and type of system: batch , time shared, single user, multi user, distributed , real time or general purpose. Beyond this highest level, the requirements may be much harder to specify. The requirements can be divided into much two basic groups: *user goal* and *system goals.* Users desire a system that is easy to use, reliable, safe and fast. People who design, implement and operate the system, require a system that is easy to design, implement and maintain. An important design goal is separation of mechanisms and policies.

▪ **Mechanism:** they determine how to do something. A general mechanism is more desirable. Example: CPU protection.
▪ **Policy:** determine what will be done. Example: Initial value in the counter used for CPU protection.

The separation of policy and mechanism is important for flexibility, as policies are likely to change across places or over time. For example, the system administrator can set the initial value in counter before booting a system.

■ Implementation

Once an operating system is designed, it must be implemented. Traditionally operating systems have been written in assembly language. Now however they are written in higher-level languages such as C/ C++ since these allow the code to be written faster, more compact, easier to understand and easier to port.

UNIX/LINUX Directory Structure

Dennis Ritchie and Ken Thomsom wrote UNIX at the Bell Labs in 1969. It was initially written in assembly language and a high-level language called Bit was later converted from B to C language. Linus Torvalds, an undergraduate student at the University of

Helsinki, Finland, wrote Linux in 1991. It is one of the most popular operating systems, certainly for PCs.

UNIX has a **hierarchical file system structure** consisting of a **root directory** (denoted as /) with other directories and files hanging under it. Unix uses a directory hierarchy that is commonly represented as folders. However, instead of using graphical folders typed commands (in a command line user interface) are used to navigate the system. Particular files are then represented by paths and filenames much like they are in html addresses. A pathname is the list of directories separated by slashes (/). If a pathname starts with a /, it refers to the root directory. The last component of a path may be a file or a directory. A pathname may simply be a file or directory name. For example, /usr/include/sys/param.h, ~/courses/cs604, and prog1.c are pathnames.

When you log in, the system places you in a directory called your **home directory** (also called **login directory**). You can refer to your home directory by using the ~ or $PATH in Bash, Bourne shell, and Korn shells and by using $path in the C and TC shells.

Shells also understand both **relative** and **absolute pathnames**. An absolute pathname starts with the root directory (/) and a relative pathname starts with your home directory, your current directory, or the parent of your **current directory** (the directory that you are currently in). For example, /usr/include/sys/param.h is an absolute pathname and ~/courses/cs604 and prog1.c are relative pathnames.

You can refer to your current directory by using **.** (pronounced dot) and the parent of your current directory by using **..** (pronounced dotdot). For example, if nadeem is currently in the courses directory, he can refer to his home directory by using .. and his personal directory by using ../personal. Similarly, he can refer to the directory for this course by using cs604.

Figures 4.5 and 4.6 show sample directory structures in a UNIX/Linux system. The user nadeem has a subdirectory under his home directory, called courses. This directory contains subdirectories for the courses that you have taken, including one for this course.
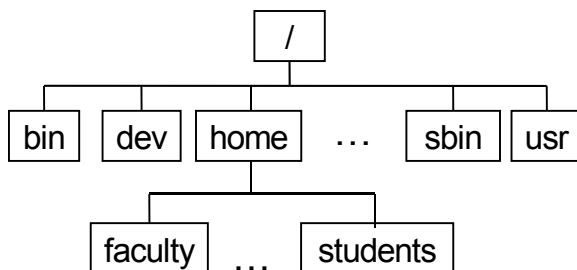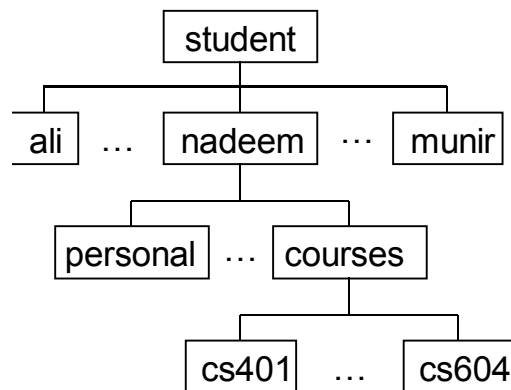
Figure 4.5 UNIX/Linux directory hierarchy

Figure 4.6 Home directories of students

Directory Structure

Some of the more important and commonly used directories in the Linux directory hierarchy are listed in Table 4.1. Many of the directories listed in the table are also found in a UNIX file system.

Table 4.1 <mark>Important directories in the Linux operating system and their purpose</mark>

/	**The root directory** (not to be concerned with the root account) is similar to a drive letter in Windows (C:\, D:\, etc.) except that in the Linux directory structure there is only one root directory and everything falls under it (including other file systems and partitions). The root directory is the directory that contains all other directories. When a directory structure is displayed as a tree, the root directory is at the top. Typically no files or programs are stored directly under root.
/bin	<mark>This directory holds binary executable</mark> files that are essential for correct operation of the system (exactly which binaries are in this directory is often dependent upon the distribution). These binaries are usually available for use by all users. **/usr/bin** can also be used for this purpose as well.
/boot	<mark>This directory includes essential system boot files including the kernel image</mark> .
/dev	<mark>This directory contains the devices available to Linux</mark>. Remember that Linux treats devices like files and you can read and write to them as if they were. Everything from floppy drives to printers to your mouse is contained in this directory. <mark>Included in this directory is the notorious /dev/null, which is most useful for deleting outputs of various, functions and programs</mark>.
/etc	<mark>Linux uses this directory to store system configuration files</mark>. Most files in this directory are <mark>text and can be edited with your favorite text editor</mark>. This is one of Linux's greatest advantages because there is never a hidden check box and just about all your configurations are in one place. **/etc/inittab** <mark>is a text file that details what processes are started at system boot up and during regular operation.</mark> **/etc/fstab** <mark>identifies file systems and their mount points (like floppy, CD-ROM, and hard disk drives).</mark> **/etc/passwd** <mark>is where users are defined.</mark>
/home	<mark>This is where every user on a Linux system will have a personal directory</mark>. If your username is "chris" then your home directory will be "/home/chris". A quick way to return to your home directory is by entering the "<mark>cd</mark>" <mark>command.</mark> <mark>Your current working directory will be changed to your home directory</mark>. Usually, the permissions on user directories are set so that only root and the user the directory belongs to can access or store information inside of it. When partitioning a Linux file system this directory will typically need the most space.
/lib	<mark>Shared libraries and kernel modules are stored in this directory</mark>. The

libraries can be dynamically linked which makes them very similar to DLL files in the Windows environment.

/lost+found This is the directory where Linux keeps files that are restored after a crash or when a partition hasn't been unmounted properly before a shutdown.

/mnt Used for mounting temporary filesystems. Filesystems can be mounted anywhere but the /mnt directory provides a convenient place in the Linux directory structure to mount temporary file systems.

/opt Often used for storage of large applications packages

/proc This is a special, "virtual" directory where system processes are stored. This directory doesn't physically exist but you can often view (or read) the entries in this directory.

/root The home directory for the superuser (root). Not to be confused with the root (/) directory of the Linux file system.

/sbin Utilities used for system administration (halt, ifconfig, fdisk, etc.) are stored in this directory. /usr/sbin, and /usr/local/sbin are other directories that are used for this purpose as well. **/sbin/init.d** are scripts used by /sbin/init to start the system.

/tmp Used for storing temporary files. Similar to C:\Windows\Temp.

/usr Typically a shareable, read-only directory. Contains user applications and supporting files for those applications. **/usr/X11R6** is used by the X Window System. **/usr/bin** contains user accessible commands. **/usr/doc** holds documentation for /usr applications. **/usr/include** this directory contains header files for the C compiler. **/usr/include/g++** contains header files for the C++ compiler. **/usr/lib** libraries, binaries, and object files that aren't usually executed directly by users. **/usr/local** used for installing software locally that needs to be safe from being overwritten when system software updates occur. **/usr/man** is where the manual pages are kept. **/usr/share** is for read-only independent data files. **/usr/src** is used for storing source code of applications installed and kernel sources and headers.

/var This directory contains variable data files such as logs (**/var/log**), mail (**/var/mail**), and spools (**/var/spool**) among other things.

(Source: http://www.chrisshort.net/archives/2005/01/linux-directory-structure.php)

Operating Systems
Lecture No. 5

Reading Material
- Operating Systems Structures, Chapter 4
- PowerPoint Slides for Lecture 3

Summary
- Browsing UNIX/Linux directory structure
- Useful UNIX/Linux commands
- Process concept
- Process scheduling concepts
- Process creation and termination

Browsing UNIX/Linux directory structure

We discussed in detail the UNIX/Linux directory structure in lecture 4. We will continue that discussion and learn how to browse the UNIX/Linux directory structure. In Figure 5.1, we have repeated for our reference the home directory structure for students. In the rest of this section, we discuss commands for creating directories, removing directories, and browsing the UNIX/Linux directory structure.
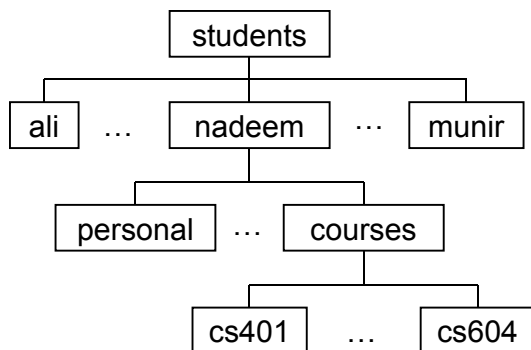
Figure 5.1 Home directories for students

Displaying Directory Contents

You can display the contents (names of files and directories) of a directory with the `ls` command. Without an argument, it assumes your current working directory. So, if you run the `ls` command right after you login, it displays names of files and directories in your home directory. It does not list those files whose names start with a dot (.). Files that start with a dot are known as **hidden files** (also called dot files). You should not modify these files unless you are quite familiar with the

purpose of these files and why you want to modify them. You can display all the files in a directory by using `ls -a` command. Your can display the long listing for the contents of a directory by using the `ls -l` command. The following session shows sample runs of these commands.

```
$ ls
books          courses          LinuxKernel      chatClient.c  chatServer.c
$ ls -a
.              .bash_history   courses          .login        .profile
..             .bash_profile   .cshrc           books
chatClient.c  chatServer.c    LinuxKernel
$ ls -l
drwxr-xr-x   3 msarwar   faculty     512 Oct 28 10:28 books
-rw-r--r--   1 msarwar   faculty    9076 Nov  4 10:14 chatClient.c
-rw-r--r--   1 msarwar   faculty    8440 Nov  4 10:16 chatServer.c
drwxr-xr-x   2 msarwar   faculty     512 Feb 27 17:21 courses
drwxr-xr-x   2 msarwar   faculty     512 Oct 21 14:55 LinuxKernel
$
```

The output of the `ls -l` command gives you the following information about a file:

- 1st character: type of a file
- Rest of letters in the 1st field: access privileges on the file
- 2nd field: number of hard links to the file
- 3rd field: owner of the file
- 4th field: Group of the owner
- 5th field: File size in bytes
- 6th and 7th fields: Date last updated
- 8th field: Time last updated
- 9th field: File name

We will talk about file types and hard links later in the course.

Creating Directories

You can use the `mkdir` command to create a directory. In the following session, the first command creates the courses directory in your current directory. If we assume that your current directory is your home directory, this command creates the courses directory under your home directory. The second command creates the cs604 directory under the ~/courses directory (i.e., the under the courses directory under your home directory). The third command creates the programs directory under your ~/courses/cs604 directory.

```
$ mkdir courses
$ mkdir ~/courses/cs604
$ mkdir ~/courses/cs604/programs
$
```

You could have created all of the above directories with the `mkdir -p` `~/courses/cs604/programs` command.

Removing (Deleting) Directories

You can remove (delete) an empty directory with the `mkdir` command. The command in the following session is used to remove the ~/courses/cs604/programs directory if it is empty.

```
$ rmdir courses
$
```

Changing Directory

You can jump from one directory to another (i.e., change your working directory) with the `cd` command. You can use the `cd ~/courses/cs604/programs` command to make ~/courses/cs604/programs directory your working directory. The `cd` or `cd $HOME` command can be used to make your home directory your working directory.

Display Absolute Pathname of Your Working Directory

You can display the absolute pathname of your working directory with the `pwd` command, as shown below.

```
$ pwd
/home/students/nadeem/courses/cs604/programs
$
```

Copying, Moving, and Removing Files

We now discuss the commands to copy, move (or rename), and remove files.

Copying Files

You can use the `cp` command for copying files. You can use the `cp file1 file2` command to copy file1 to file2. The following command can be used to copy file1 in your home directory to the ~/memos directory as file2.

```
$ cp ~/file1 ~/memos/file2
$
```

Moving Files

You can use the `mv` command for moving files. You can use the `mv file1 file2` command to move file1 to file2. The following command can be used to move file1 in your home directory to the ~/memos directory as file2.

```
$ mv ~/file1 ~/memos/file2
$
```

Removing Files

You can use the `rm` command to remove files. You can use the `rm file1` command to remove file1. You can use the first command the following command

to remove the test.c file in the ~/courses/cs604/programs directory and the second command to remove all the files with .o extension (i.e., all object files) in your working directory.

```
$ rm ~/courses/cs604/programs/test.c
$ rm *.o
$
```

Compiling and Running C Programs

You can compile your program with the gcc command. The output of the compiler command, i.e., the executable program is stored in the a.out file by default. To compile a source file titled program.c, type:

```
$ gcc program.c
$
```

You can run the executable program generated by this command by typing ./a.out and hitting the <Enter> key, as shown in the following session.

```
$ ./a.out
[ ... program output ... ]
$
```

You can store the executable program in a specific file by using the −o option. For example, in the following session, the executable program is stored in the assignment file.

```
$ gcc program.c −o assignment
$
```

The gcc compiler does not link many libraries automatically. You can link a library explicitly by using the −l option. In the following session, we are asking the compiler to link the math library with our object file as it creates the executable file.

```
$ gcc program.c −o assignment -lm
$ assignment
[ ... program output ... ]
$
```

Process Concept

A process can be thought of as a program in execution. A process will need certain resources – such as CPU time, memory, files, and I/O devices – to accompany its task. These resources are allocated to the process either when it is created or while it is executing.

A process is the unit of work in most systems. Such a system consists of a collection of processes: operating system processes execute system code and user processes execute user code. All these processes may execute concurrently.

Although traditionally a process contained only a single thread of control as it ran, most modern operating systems now support processes that have multiple threads.

A batch system executes jobs (background processes), whereas a time-shared system has user programs, or tasks. Even on a single user system, a user may be able to run several programs at one time: a word processor, web browser etc.

A process is more than program code, which is sometimes known as the text section. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's register. In addition, a process generally includes the process stack, which contains temporary data (such as method parameters, the process stack, which contains temporary data), and a data section, which contains global variables.

A program by itself is not a process: a program is a passive entity, such as contents of a file stored on disk, whereas a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources. Although two processes may be associated with the same program, they are considered two separate sequences of execution. E.g. several users may be running different instances of the mail program, of which the text sections are equivalent but the data sections vary.

Processes may be of two types:

- **IO bound processes:** spend more time doing IO than computations, have many short CPU bursts. Word processors and text editors are good examples of such processes.
- **CPU bound processes:** spend more time doing computations, few very long CPU bursts.

Process States

As a process executes, it changes states. The state of a process is defined in part by the current activity of that process. Each process may be in either of the following states, as shown in Figure 5.2:

- **New:** The process is being created.
- **Running:** Instructions are being executed.
- **Waiting:** The process is waiting for some event to occur (such as an I/O completion or reception of a signal.
- **Ready:** The process is waiting to be assigned to a processor.
- **Terminated:** The process has finished execution.
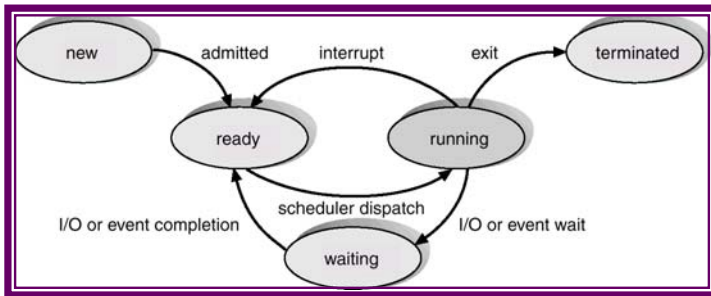
Figure 5.2 Process state diagram

Process Control Block

Each process is represented in the operating system by a **process control block** (PCB) – also called a task control block, as shown in Figure 5.3. A PCB contains many pieces of information associated with a specific process, including these:

- **Process state:** The state may be new, ready, running, waiting, halted and so on.
- **Program counter:** The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers:** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers and general-purpose registers, plus any condition code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterwards.
- **CPU Scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information:** This information may include such information such as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
- **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information:** The information includes the list of I/O devices allocated to the process, a list of open files, and so on.

Figure 5.3 Process control block (PCB)

Process Scheduling

The objective of multiprogramming is to have some process running all the time so as to maximize CPU utilization. The objective of time-sharing is to switch the CPU among processors so frequently that users can interact with each program while it is running. A uniprocessor system can have only one running process at a given time. If more processes exist, the rest must wait until the CPU is free and can be rescheduled. Switching the CPU from one process to another requires saving of the context of the current process and loading the state of the new process, as shown in Figure 5.4. This is called **context switching**.
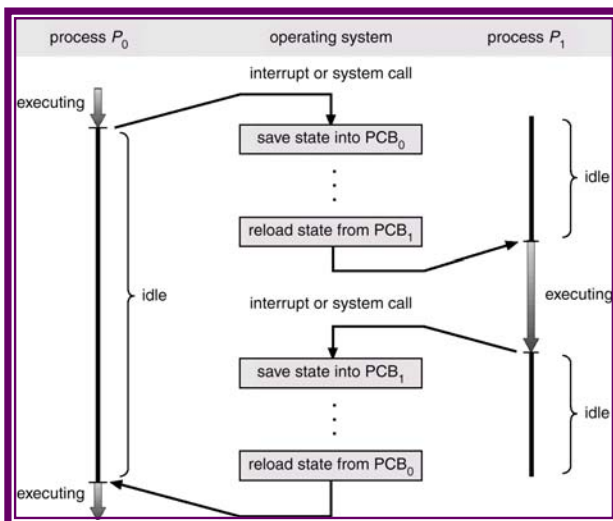
Figure 5.4 Context switching

Scheduling Queues

As shown in Figure 5.5, a contemporary computer system maintains many scheduling queues. Here is a brief description of some of these queues:

- **Job Queue:** <mark>As processes enter the system, they are put into a job queue. This queue consists of all processes in the system.</mark>
- **Ready Queue:** <mark>The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue.</mark> This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB is extended to include a pointer field that points to the next PCB in the ready queue.
- **Device Queue:** <mark>When a process is allocated the CPU, it executes for a while, and eventually quits, is interrupted or waits for a particular event, such as completion of an I/O request. In the case of an I/O request, the device may be busy with the I/O request of some other process, hence the list of processes waiting for a particular I/O device is called a device queue.</mark> Each device has its own device queue.
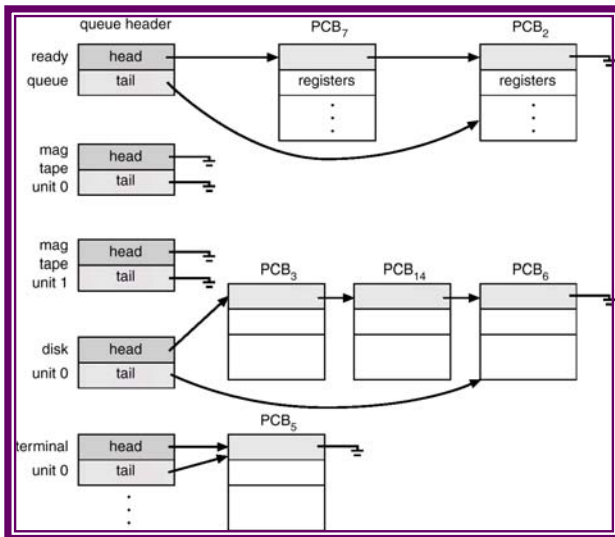
Figure 5.5 Scheduling queue

In the queuing diagram shown in Figure 5.6 below, each rectangle box represents a queue, and two such queues are present, the ready queue and an I/O queue. A new process is initially put in the ready queue, until it is dispatched. Once the process is executing, one of the several events could occur:
- The process could issue an I/O request, and then be placed in an I/O queue.
- The process could create a new sub process and wait for its termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.
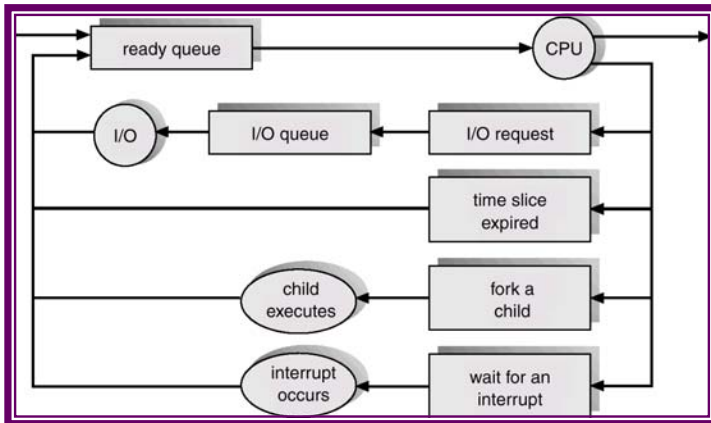
Figure 5.6 Queuing diagram of a computer system

Schedulers

A process migrates between the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The appropriate scheduler carries out this selection process. The **Long-term scheduler** (or job scheduler) selects which processes should be brought into the ready queue, from the job pool that is the list of all jobs in the system. The **Short-term scheduler** (or CPU scheduler) selects which process should be executed next and allocates CPU.

The primary distinction between the two schedulers is the frequency of execution. The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Often the short-term scheduler executes at least once every 100 milliseconds. Because of the brief time between executions, the short-term scheduler must be fast. If it takes 10 milliseconds to decide to execute a process for 100 milliseconds, then 10/(100+10)=9 % of the CPU is being used for scheduling only. The long-term scheduler, on the other hand executes much less frequently. There may be minutes between the creations of new processes in the system. The long-term scheduler controls the degree of multiprogramming – the number of processes in memory. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average department rate of processes leaving the system. Because of the longer interval between execution s, the long-term scheduler can afford to take more time to select a process for execution.

The long-term scheduler must select a good mix of I/O bound and CPU bound jobs. The reason why the long-term scheduler must select a good mix of I/O bound and CPU bound jobs is that if the processes are I/O bound, the ready queue will be mostly empty and the short-term scheduler will have little work. On the other hand, if the processes are mostly CPU bound, then the devices will go unused and the system will be unbalanced.

Some operating systems such as time-sharing systems may introduce a **medium-term scheduler**, which removes processes from memory (and from active contention for the CPU) and thus reduces the degree of multiprogramming. At some later time the process can be reintroduced at some later stage, this scheme is called **swapping.** The process is swapped out, and is later swapped in by the medium term scheduler. Swapping may be necessary to improve the job mix, or because a change is memory requirements has over committed available memory, requiring memory to be freed up. As shown in Figure 5.7, the work carried out by the swapper to move a process from the main memory to disk is known as swap out and moving it back into the main memory is called swap in. The area on the disk where swapped out processes are stored is called the **swap space**.
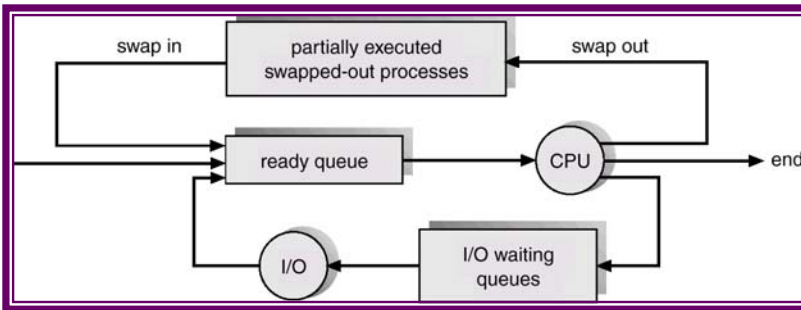
Figure 5.7 Computer system queues, servers, and swapping

Operating Systems
Lecture No. 6

Reading Material
- Operating Systems Concepts, Chapter 4
- UNIX/Linux manual pages for the `fork()` system call

Summary
- Process creation and termination
- Process management in UNIX/Linux— system calls: `fork`, `exec`, `wait`, `exit`
- Sample codes

Operations on Processes
The processes in the system execute concurrently and they must be created and deleted dynamically thus the operating system must provide the mechanism for the creation and deletion of processes.

Process Creation
A process may create several new processes via a create-process system call during the course of its execution. The creating process is called a **parent process** while the new processes are called the **children** of that process. Each of these new processes may in turn create other processes, forming a tree of processes. Figure 6.1 shows partially the process tree in a UNIX/Linux system.
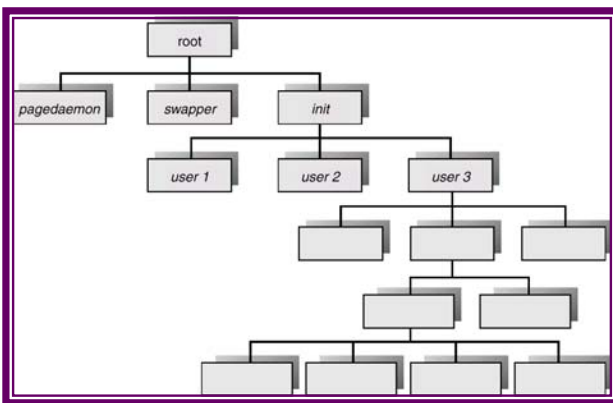
Figure 6.1 Process tree in UNIX/Linux

In general, a process will need certain resources (such as CPU time, memory files, I/O devices) to accomplish its task. When a process creates a sub process, also known as a child, that sub process may be able to obtain its resources directly from the operating system or may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among several of its children. Restricting a

process to a subset of the parent's resources prevents a process from overloading the system by creating too many sub processes.

When a process is created it obtains in addition to various physical and logical resources, initialization data that may be passed along from the parent process to the child process. When a process creates a new process, two possibilities exist in terms of execution:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process.
2. The child process has a program loaded into it.

In order to consider these different implementations let us consider the UNIX operating system. In UNIX its process identifier identifies a process, which is a unique integer. A new process is created by the `fork` system call. The new process consists of a copy of the address space of the parent. This mechanism allows the parent process to communicate easily with the child process. Both processes continue execution at the instruction after the `fork` call, with one difference, the return code for the `fork` system call is zero for the child process, while the process identifier of the child is returned to the parent process.

Typically the `execlp` system call is used after a `fork` system call by one of the two processes to replace the process' memory space with a new program. The `execlp` system call loads a binary file in memory –destroying the memory image of the program containing the `execlp` system call.—and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways. The parent can then create more children, or if it has nothing else to do while the child runs, it can issue a `wait` system call to move itself off the ready queue until the termination of the child. The parent waits for the child process to terminate, and then it resumes from the call to wait where it completes using the `exit` system call.

Process termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by calling the exit system call. At that point, the process may return data to its parent process (via the wait system call). All the resources of the process including physical and virtual memory, open the files and I/O buffers – are de allocated by the operating system.

Termination occurs under additional circumstances. A process can cause the termination of another via an appropriate system call (such as abort). Usually only the parent of the process that is to be terminated can invoke this system call. Therefore parents need to know the identities of its children, and thus when one process creates another process, the identity of the newly created process is passed to the parent.

A parent may terminate the execution of one of its children for a variety of reasons, such as:

- The child has exceeded its usage of some of the resources that it has been allocated. This requires the parent to have a mechanism to inspect the state of its children.
- The task assigned to the child is no longer required.

- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates. On such a system, if a process terminates either normally or abnormally, then all its children must also be terminated. This phenomenon referred to as cascading termination, is normally initiated by the operating system.

Considering an example from UNIX, we can terminate a process by using the `exit` system call, its parent process may wait for the termination of a child process by using the wait system call. The wait system call returns the process identifier of a terminated child, so that the parent can tell which of its possibly many children has terminated. If the parent terminates however all its children have assigned as their new parent, the `init` process. Thus the children still have a parent to collect their status and execution statistics.

The `fork()` system call

When the fork system call is executed, a new process is created. The original process is called the parent process whereas the process is called the child process. The new process consists of a copy of the address space of the parent. This mechanism allows the parent process to communicate easily with the child process. On success, both processes continue execution at the instruction after the `fork` call, with one difference, the return code for the `fork` system call is zero for the child process, while the process identifier of the child is returned to the parent process. On failure, a -1 will be returned in the parent's context, no child process will be created, and an error number will be set appropriately.

The synopsis of the fork system call is as follows:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

```
main()
{
    int pid;
    ...
    pid = fork();
    if (pid == 0) {
    /* Code for child */
    ...
    }
    else {
    /* Code for parent */
    ...
    }
    ...
}
```

Figure 6.2 Sample code showing use of the `fork()` system call

Figure 6.2 shows sample code, showing the use of the `fork()` system call and Figure 6.3 shows the semantics of the fork system call. As shown in Figure 6.3, `fork()`

creates an exact memory image of the parent process and returns 0 to the child process and the process ID of the child process to the parent process.

Parent Process   `pid = 12345`

Child Process   `pid = 0`

Kernel Space
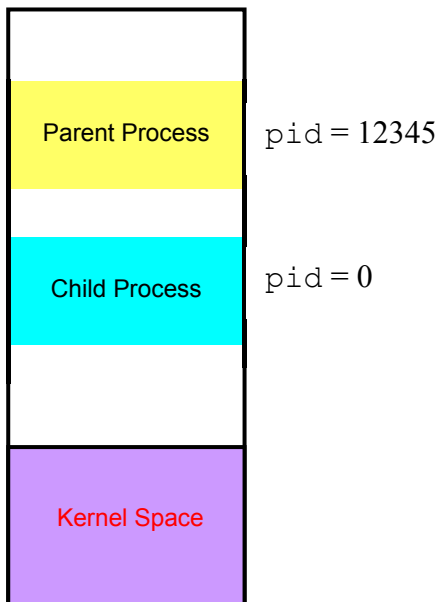
Figure 6.3 Semantics of the fork system call

After the `fork()` system call the parent and the child share the following:
- Environment
- Open file descriptor table
- Signal handling settings
- Nice value
- Current working directory
- Root directory
- File mode creation mask (umask)

The following things are different in the parent and the child:
- Different process ID (PID)
- Different parent process ID (PPID)
- Child has its own copy of parent's file descriptors

The `fork()` system may fail due to a number of reasons. One reason maybe that the maximum number of processes allowed to execute under one user has exceeded, another could be that the maximum number of processes allowed on the system has exceeded. Yet another reason could be that there is not enough swap space.

Operating Systems
Lecture No. 7

Reading Material
- Operating Systems Concepts, Chapter 4
- UNIX/Linux manual pages for `execlp()`, `exit()`, and `wait()` system calls

Summary
- The `execlp()`, `wait()`, and `exec()` system calls and sample code
- Cooperating processes
- Producer-consumer problem
- Interprocess communication (IPC) and process synchronization

The `wait()` system call

The wait system call suspends the calling process until one of the immediate children terminate, or until a child that is being traced stops because it has hit an event of interest. The wait will return prematurely if a signal is received. If all child processes stopped or terminated prior to the call on wait, return is immediate. If the call is successful, the process ID of a child is returned. If the parent terminates however all its children have assigned as their new parent, the init process. Thus the children still have a parent to collect their status and execution statistics. The synopsis of the wait system call is as follows:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
```

A **zombie process** is a process that has terminated but whose exit status has not yet been received by its parent process or by init. Sample code showing the use of `fork()` and `wait()` system calls is given in Figure 7.1 below.

```
#include <stdio.h>
void main(){
  int pid, status;
  pid = fork();
  if(pid == -1) {
   printf("fork failed\n");
   exit(1);
  }
  if(pid == 0) {  /* Child */
     printf("Child here!\n");
     exit(0);
  }
  else { /* Parent */
    wait(&status);
```

```
          printf("Well done kid!\n");
          exit(0);
      }
}
```

Figure 7.1 Sample code showing use of the `fork()`
and `wait()` system calls

The `execlp()` system call

Typically, the `execlp()` system call is used after a `fork()` system call by one of the two processes to replace the process' memory space with a new program. The new process image is constructed from an ordinary, executable file. This file is either an executable object file, or a file of data for an interpreter. There can be no return from a successful *exec* because the calling process image is overlaid by the new process image. In this manner, the two processes are able to communicate and then go their separate ways. The synopsis of the `execlp()` system call is given below:

```
#include <unistd.h>
int execlp (const char *file, const,char *arg0, ...,
            const char *argn,(char *)0);
```

Sample code showing the use of `fork()` and `execlp()` system calls is given in Figure 7.2 below.

```
#include <stdio.h>
void main()
{
  int pid, status;

  pid = fork();
  if(pid == -1) {
    printf("fork failed\n");
    exit(1);
  }
  if(pid == 0) {   /* Child */
      if (execlp("/bin/ls", "ls", NULL)< 0) {
        printf("exec failed\n");
        exit(1);
      }
  }
  else { /* Parent */
    wait(&status);
    printf("Well done kid!\n");
    exit(0);
  }
}
```
Figure 7.2 Sample code showing use of `fork()`, `execlp()`, `wait()`, and `exit()`

The semantics of `fork()`, followed by an execlp() system call are shown In Figure 7.3 below.
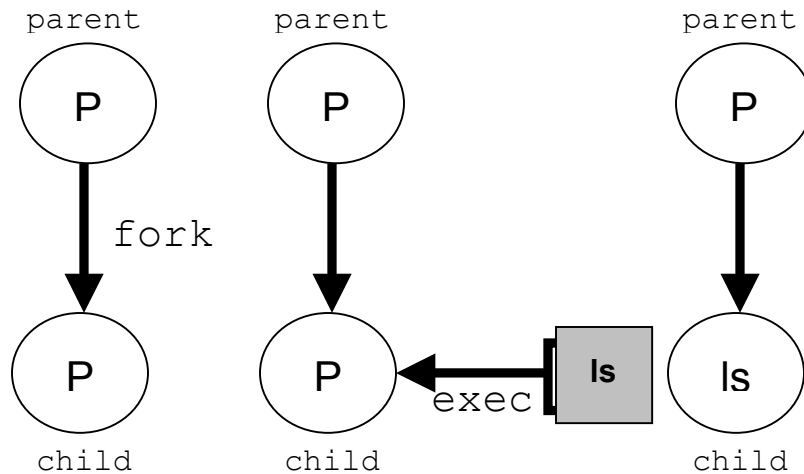
Figure 7.3 Semantics of `fork()` followed by `exec()`

Cooperating Processes

The concurrent processes executing in the operating system may be either independent processes or cooperating processes. A process is independent if it cannot affect or be affected by any other process executing in the system. Clearly any process that shares data with other processes is a cooperating process. The advantages of cooperating processes are:

- **Information sharing:** Since several users may be interested in the same piece of information (for instance, a shared file) we must provide an environment to allow concurrent users to access these types of resources.
- **Computation speedup:** If we want a particular task to run faster, we must break it into subtasks each of which will be running in parallel with the others. Such a speedup can be obtained only if the computer has multiple processing elements (such as CPU's or I/O channels).
- **Modularity:** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
- **Convenience:** Even an individual user may have many tasks on which to work at one time. For instance, a user may be editing, printing, and compiling in parallel.

To illustrate the concept of communicating processes, let us consider the producer-consumer problem. A **producer** process produces information that is consumed by a **consumer** process. For example, a compiler may produce assembly code that is consumed by an assembler. To allow a producer and consumer to run concurrently, we must have available a buffer of items that can be filled by a producer and emptied by a consumer. The producer and consumer must be synchronized so that the consumer does not try to consume an item that has not yet been produced. The bounded buffer problem assumes a fixed buffer size, and the consumer must wait if the buffer is empty and the producer must wait if the buffer is full, whereas the unbounded buffer places no practical limit on the size of the buffer. Figure 7.4 shows the problem in a diagram. This buffer may be provided by interprocess communication (discussed in the next section) or with the use of shared memory.
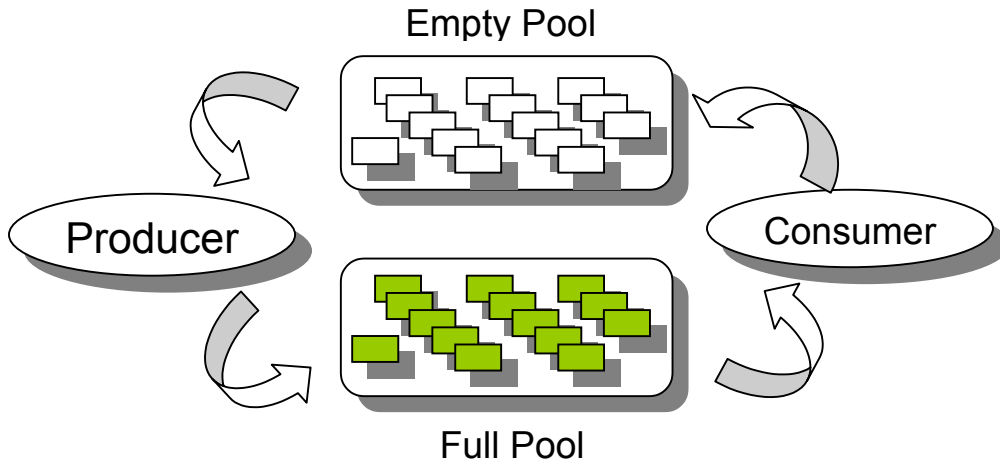
Figure 7.4 The producer-consumer problem

Figure 7.5 shows the shared buffer and other variables used by the producer and consumer processes.

```
#define BUFFER_SIZE 10
typedef struct
{
…
} item;
item buffer[BUFFER_SIZE];
int in=0;
int out=0;
```

Figure 7.5 Shared buffer and variables used by the producer and consumer processes

The shared buffer is implemented as a circular array with two logical pointers: in an out. The 'in' variable points to the next free position in the buffer; 'out' points to the first full position in the buffer. The buffer is empty when in==out, the buffer is full when ((in+1)%BUFFER_SIZE)==out. The code structures for the producer and consumer processes are shown in Figure 7.6.

```
Producer Process
while(1) {
    /*Produce an item in nextProduced*/
    while(((in+1)%BUFFER_SIZE)==out); /*do nothing*/
      buffer[in]=nextProduced;
      in=(in+1)%BUFFER_SIZE;
}
Consumer Process
while(1) {
   while(in == out); //do nothing
   nextConsumed=buffer[out];
   out=(out+1)%BUFFER_SIZE;
   /*Consume the item in nextConsumed*/
}
```

Figure 7.6 Code structures for the producer and consumer processes

Operating Systems
Lecture No. 8

Reading Material
- Operating Systems Concepts, Chapter 4
- UNIX/Linux manual pages for `pipe()`, `fork()`, `read()`, `write()`, `close()`, and `wait()` system calls

Summary
- Interprocess communication (IPC) and process synchronization
- UNIX/Linux IPC tools (pipe, named pipe—FIFO, socket, TLI, message queue, shared memory)
- Use of UNIC/Linux pipe in a sample program

Interprocess Communication (IPC)
IPC provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. We discuss in this section the various message passing techniques and issues related to them.

Message Passing System
The function of a message system is to allow processes to communicate without the need to resort to the shared data. Messages sent by a process may be of either fixed or variable size. If processes P and Q want to communicate, a communication link must exist between them and they must send messages to and receive messages from each other through this link. Here are several methods for logically implementing a link and the send and receive options:
- Direct or indirect communication
- Symmetric or asymmetric communication
- Automatic or explicit buffering
- Send by copy or send by reference
- Fixed size or variable size messages

We now look at the different types of message systems used for IPC.

Direct Communication
With direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. The send and receive primitives are defined as:
- Send(P, message) – send a message to process P
- Receive(Q, message) – receive a message from process Q.

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate
- A link is associated with exactly two processes.
- Exactly one link exists between each pair of processes.

Unlike this symmetric addressing scheme, a variant of this scheme employs asymmetric addressing, in which the recipient is not required to name the sender.

- Send(P, message) – send a message to process P
- Receive(id, message) – receive a message from any process; the variable id is set to the name of the process with which communication has taken place.

Indirect Communication

With indirect communication, messages can be sent to and received from ==mailboxes==. Here, two processes can communicate only if they share a mailbox. The send and receive primitives are defined as:

- Send(A, message) – send a message to mailbox A.
- Receive(A, message) – receive a message from mailbox A.

A communication link in this scheme has the following properties:

- A link is established between a pair of processes only if both members have a shared mailbox.
- A link is associated with more than two processes.
- A number of different links may exist between each pair of communicating processes, with each link corresponding to one mailbox.

Synchronization

Communication between processes takes place by calls to send and receive primitives (i.e., functions). ==Message passing may be either blocking or non-blocking also called as synchronous and asynchronous.==

- **Blocking send:** The sending process is blocked until the receiving process or the mailbox receives the message.
- **Non-blocking send:** The sending process sends the message and resumes operation.
- **Blocking receive:** The receiver blocks until a message is available.
- **Non-blocking receiver:** The receiver receives either a valid message or a null.

Buffering

==Whether the communication is direct or indirect, messages exchanged by the processes reside in a temporary queue==. This queue can be implemented in three ways:

- **Zero Capacity:** The queue has maximum length zero, thus the link cannot have any messages waiting in it. In this case the sender must block until the message has been received.
- **Bounded Capacity:** This queue has ==finite== length n; thus at most n messages can reside in it. If the queue is not full when a new message is sent, the later is placed in the queue and the sender resumes operation. If the queue is full, the sender blocks until space is available.

- **Unbounded Capacity:** The queue has infinite length; thus the sender never blocks.

UNIX/Linux IPC Tools

UNIX and Linux operating systems provide many tools for interprocess communication, mostly in the form of APIs but some also for use at the command line. Here are some of the commonly supported IPC tools in the two operating systems.
- Pipe
- Named pipe (FIFO)
- BSD Socket
- TLI
- Message queue
- Shared memory
- Etc.

Overview of `read()`, `write()`, and `close()` System Calls

We need to understand the purpose and syntax of the read, write and close system calls so that we may move on to understand how communication works between various Linux processes. The read system call is used to read data from a file descriptor. The synopsis of this system call is:

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

`read()` attempts to read up to count bytes from file descriptor fd into the buffer starting at buf. If count is zero, `read()` returns zero and has no other results. If count is greater than SSIZE_MAX, the result is unspecified. On success, `read()` returns the number of bytes read (zero indicates end of file) and advances the file position pointer by this number.

The `write()` system call is used to write to a file. Its synopsis is as follows:

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

`write()` attempts to write up to count bytes to the file referenced by the file descriptor fd from the buffer starting at buf. On success, `write()` returns the number of bytes written are returned (zero indicates nothing was written) and advances the file position pointer by this number. On error, `read()` returns -1, and *errno* is set appropriately. If count is zero and the file descriptor refers to a regular file, 0 will be returned without causing any other effect.

The `close()` system call is used to close a file descriptor. Its synopsis is:

```
#include <unistd.h>
int close(int fd);
```

`close()` closes a file descriptor, so that it no longer refers to any file and may be reused. If fd is the last copy of a particular file descriptor the resources associated with it are freed; if the descriptor was the last reference to a file which has been removed using

unlink(2) the file is deleted. close() returns zero on success, or -1 if an error occurred.

Pipes

A UNIX/Linux pipe can be used for IPC between related processes on a system. Communicating processes typically have sibling or parent-child relationship. At the command line, a pipe can be used to connect the standard output of one process to the standard input of another. Pipes provide a method of one-way communication and for this reason may be called half-duplex pipes.

The pipe() system call creates a pipe and returns two file descriptors, one for reading and second for writing, as shown in Figure 8.1. The files associated with these file descriptors are streams and are both opened for reading and writing. Naturally, to use such a channel properly, one needs to form some kind of protocol in which data is sent over the pipe. Also, if we want a two-way communication, we'll need two pipes.
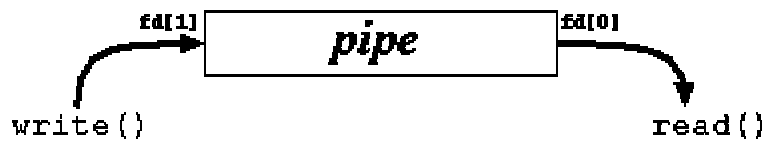
Figure 8.1 A UNIX/Linux pipe with a read end and a write end

The system assures us of one thing: the order in which data is written to the pipe, is the same order as that in which data is read from the pipe. The system also assures that data won't get lost in the middle, unless one of the processes (the sender or the receiver) exits prematurely. The pipe() system call is used to create a read-write pipe that may later be used to communicate with a process we'll fork off. The synopsis of the system call is:

```
#include <unistd.h>
int pipe (int fd[2]);
```

Each array element stores a file descriptor. fd[0] is the file descriptor for the read end of the pipe (i.e., the descriptor to be used with the read system call), whereas fd[1] is the file descriptor for the write end of the pipe. (i.e., the descriptor to be used with the write system call).The function returns -1 if the call fails. A pipe is a bounded buffer and the maximum data written is PIPE_BUF, defined in <sys/param.h> in UNIX and in <linux/param.h> in Linux as 5120 and 4096, respectively.

Lets see an example of a two-process system in which the parent process creates a pipe and forks a child process. The child process writes the 'Hello, world!' message to the pipe. The parent process reads this messages and displays it on the monitor screen. Figure 8.2 shows the protocol for this communication and Figure 8.3 shows the corresponding C source code.

Half-duplex pipe after a fork
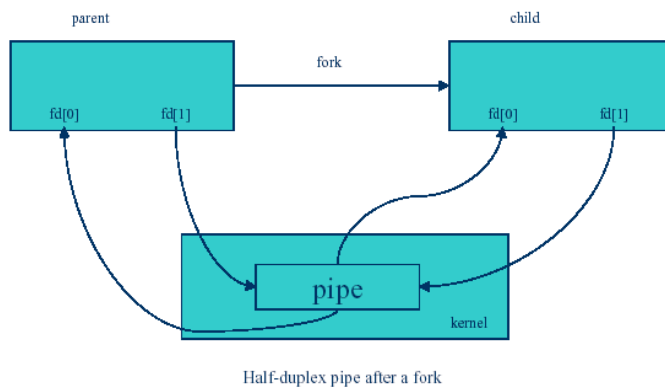
Figure 8.2 Use of UNIX/Linux pipe by parent and child for half-duplex communication

```
/* Parent creates pipe, forks a child, child writes into
   pipe, and parent reads from pipe */
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
main()
{
        int pipefd[2], pid, n, rc, nr, status;
        char *testString = "Hello, world!\n", buf[1024];

        rc = pipe (pipefd);
        if (rc < 0) {
                perror("pipe");
                exit(1);
        }
        pid = fork ();
        if (pid < 0) {
                perror("fork");
                exit(1);
        }
        if (pid == 0) {   /* Child's Code */
            close(pipefd[0]);
            write(pipefd[1], testString, strlen(testString));
            close(pipefd[1]);
            exit(0);
        }
        /* Parent's Code */
        close(pipefd[1]);
        n = strlen(testString);
        nr = read(pipefd[0], buf, nA);
        rc = write(1, buf, nr);
        wait(&status);
        printf("Good work child!\n");
        return(0);
}
```

Figure 8.3 Sample code showing use of UNIX/Linux pipe for IPC between related processes—child write the "Hello, world!" message to the parent, who reads its and displays it on the monitor screen

In the given program, the parent process first creates a pipe and then forks a child process. On successful execution, the `pipe()` system call creates a pipe, with its read end descriptor stored in pipefd[0] and write end descriptor stored in pipefd[1]. We call `fork()` to create a child process, and then use the fact that the memory image of the child process is identical to the memory image of the parent process, so the pipefd[] array is still defined the same way in both of them, and thus they both have the file descriptors of the pipe. Further more, since the file descriptor table is also copied during the fork, the file descriptors are still valid inside the child process. Thus, the parent and child processes can use the pipe for one-way communication as outlined above.

Operating Systems
Lecture No. 9

Reading Material
- Operating Systems Concepts, Chapter 4
- UNIX/Linux manual pages for `pipe()`, `fork()`, `read()`, `write()`, `close()`, and `wait()` system calls
- Lecture 9 on Virtual TV

Summary
- UNIX/Linux interprocess communication (IPC) tools and associated system calls
- UNIX/Linux standard files and kernel's mechanism for file access
- Use of pipe in a program and at the command line

Unix/Linux IPC Tools
The UNIX and Linux operating systems provide many tools for interprocess communication (IPC). The three most commonly used tools are:

- **Pipe:** Pipes are used for communication between related processes on a system, as shown in Figure 9.1. The communicating processes are typically related by sibling or parent-child relationship.
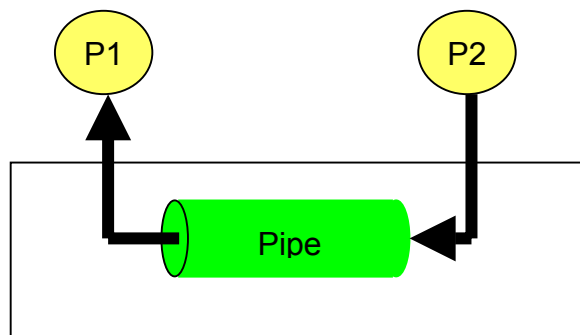
Figure 9.1 Pipes on a UNIX/Linux system

- **Named pipe (FIFO)**: FIFOs (also known as named pipes) are used for communication between related or unrelated processes on a UNIX/Linux system, as shown in Figure 9.2.
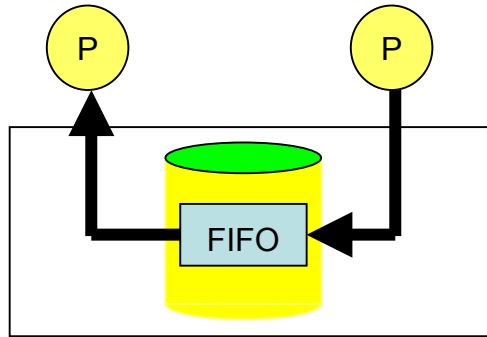
Figure 9.2 Pipes on a UNIX/Linux system

- **BSD Socket**: The BSD sockets are used for communication between related or unrelated processes on the same system or unrelated processes on different systems, as shown in Figure 9.3.
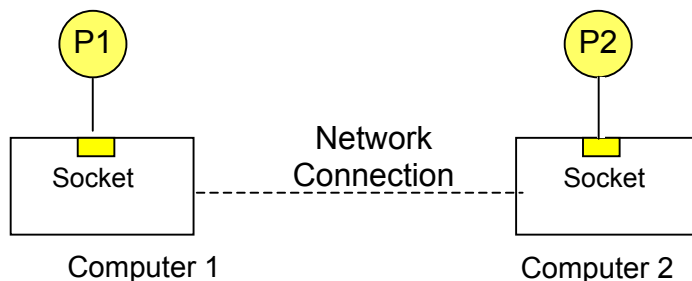
Figure 9.3 Sockets used for IPC between processes on different UNIX/Linux systems

The open() System call

The open() system call is used to open or create a file. Its synopsis is as follows:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char pathname, int oflag, /* mode_t mode */);
```

The call converts a pathname into a file descriptor (a small, non-negative integer for use in subsequent I/O as with read, write, etc.). When the call is successful, the file descriptor returned will be the lowest file descriptor not currently open for the process. This system call can also specify whether read or write will be blocking or non-blocking.

The 'oflag' argument specifies the purpose of opening the file and 'mode' specifies permission on the file if it is to be created. 'oflag' value is constructed by ORing various flags: O_RDONLY, O_WRONLY, O_RDWR, O_NDELAY (or O_NONBLOCK), O_APPEND, O_CREAT, etc.

The open() system call can fail for many reasons, some of which are:

- Non-existent file
- Operation specified is not allowed due to file permissions

- Search not allowed on a component of pathname
- User's disk quota on the file system has been exhausted

<mark>The file descriptor returned by the `open()` system call is used in the `read()` and `write()` calls for file (or pipe) I/O.</mark>

The `read()` system call

We discussed the `read()` system call in the notes for lecture 8. The call may fail for various reasons, including the following:
- <mark>Invalid 'fildes', 'buf', or 'nbyte'</mark>
- Signal caught during read

<mark>The `write()` system call</mark>

The call may fail for various reasons, including the following:
- Invalid argument
- File size limit for process or for system would exceed
- Disk is full

The `close()` system call

As discussed in the notes for lecture 8, <mark>the close() system call is used to close a file descriptor.</mark> It takes a file (or pipe) descriptor as an argument and closes the corresponding file (or pipe end).

Kernel Mapping of File Descriptors

Figure 9.4 shows the kernel mapping of a file descriptor to the corresponding file. The system-wide File Table contains entries for all of the open files on the system. UNIX/Linux allocates an inode to every (unique) file on the system to store most of the attributes, including file's location. On a read or write call, kernel traverses this mapping to reach the corresponding file.
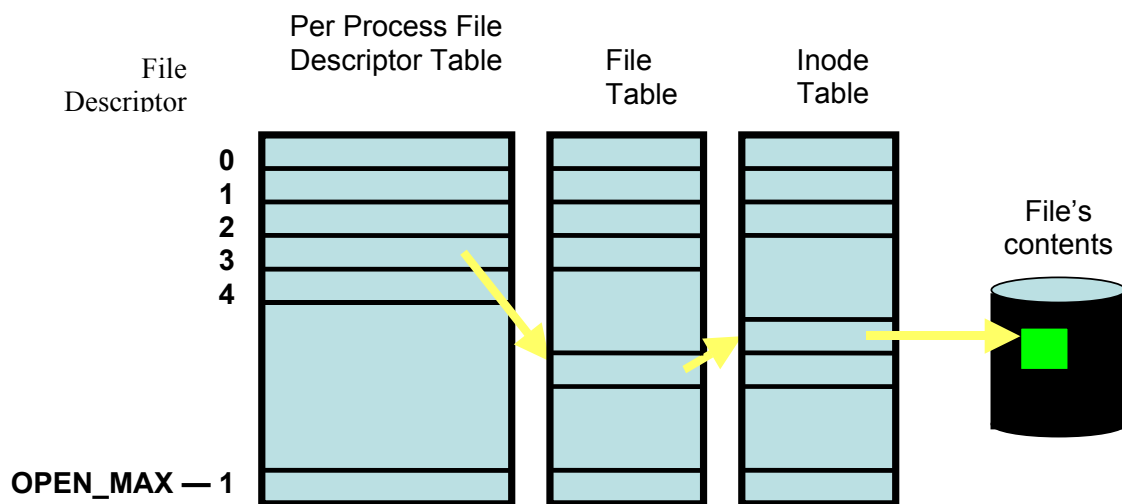
Figure 9.4 File descriptors and their mapping to files

Standard Descriptors in Unix/Linux

<mark>Three files are automatically opened by the kernel for every process for the process to read its input from and send its output and error messages to. These files are called **standard files**: standard input, standard output, and standard error.</mark> By default, standard files are attached to the terminal on which a process runs. The descriptors for standard files are known as **standard file descriptors**. Standard files, their descriptors, and their default attachments are:

- Standard input: 0 (keyboard)
- Standard output: 1 (display screen)
- Standard error: 2 (display screen)

The `pipe()` System Call

We discussed the `pipe()` system call in the notes for lecture 8. The `pipe()` system call fails for many reasons, including the following:

- At least two slots are not empty in the PPFDT—too many files or pipes are open in the process
- Buffer space not available in the kernel
- File table is full

Sample Code for IPC with a UNIX/Linux Pipe

We discussed in the notes for lecture 8 a simple protocol for communication between a parent and its child process using a pipe. Figure 9.5 shows the protocol. Code is reproduced in Figure 9.6.
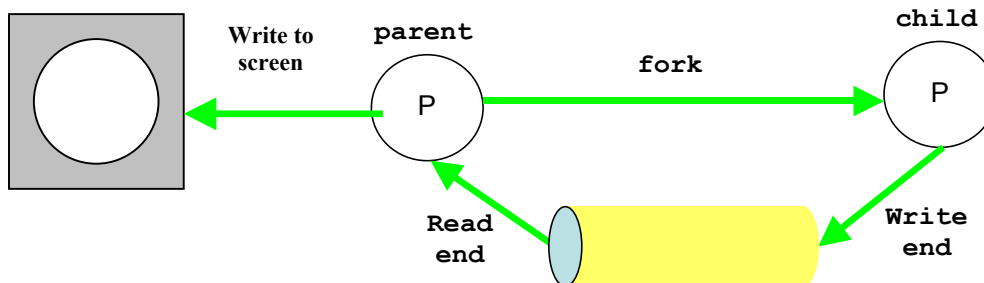
Figure 9.5 IPC between parent and child processes with a UNIX/Linux pipe

```
/* Parent creates pipe, forks a child, child writes into
   pipe, and parent reads from pipe */
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
main()
{
        int pipefd[2], pid, n, rc, nr, status;
        char *testString = "Hello, world!\n", buf[1024];

        rc = pipe (pipefd);
        if (rc < 0) {
                perror("pipe");
```

```
                exit(1);
        }
        pid = fork ();
        if (pid < 0) {
                perror("fork");
                exit(1);
        }
        if (pid == 0) {   /* Child's Code */
            close(pipefd[0]);
            write(pipefd[1], testString, strlen(testString));
            close(pipefd[1]);
            exit(0);
        }
        /* Parent's Code */
        close(pipefd[1]);
        n = strlen(testString);
        nr = read(pipefd[0], buf, nA);
        rc = write(1, buf, nr);
        wait(&status);
        printf("Good work child!\n");
        return(0);
}
```

Figure 9.6 Sample code showing use of UNIX/Linux pipe for IPC between related processes—child write the "Hello, world!" message to the parent, who reads its and displays it on the monitor screen

Command Line Use of UNIX/Linux Pipes

Pipes can also be used on the command line to connect the standard input of one process to the standard input of another. This is done by using the pipe operator which is | and the syntax is as follows:

```
cmd1 | cmd2 | ... | cmdN
```

The semantics of this command line are shown in Figure 9.7.
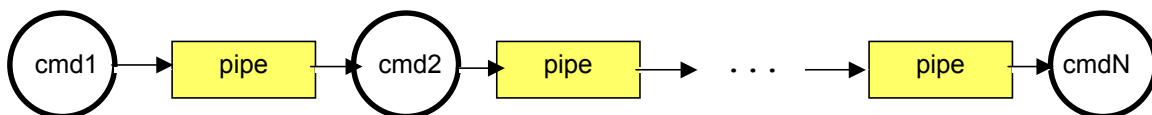
Figure 9.7 Semantics of the command line that connects cmd1 through cmdN via pipes.

The following example shows the use of the pipe operator in a shell command.

```
cat /etc/passwd | grep zaheer
```

The effect of this command is that grep command displays lines in the /etc/passwd file that contain the string "zaheer". Figure 9.8 illustrates the semantics of this command.
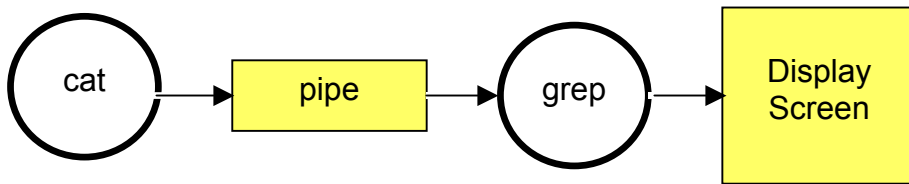
Figure 9.8 Semantics of the cat /etc/passwd | grep zaheer command

The work performed by the above command can be performed by the following sequence of commands without using the pipe operator. The first command saves the /etc/passwd file in the temp1 file and the second command displays those lines in temp1 which contain the string "zaheer". After the temp1 file has been used for the desired work, it is deleted.

```
$ cat /etc/passwd > temp1
$ grep "zaheer" temp1
$ rm temp1
```

Operating Systems
Lecture No. 10

Reading Material
- UNIX/Linux manual pages for the `mknod()` system call, the `mkfifo()` library call, and the `mkfifo` command
- Lecture 10 on Virtual TV

Summary
- Input, output, and error redirection in UNIX/Linux
- FIFOs in UNIX/Linux
- Use of FIFOs in a program and at the command line

Input, output and error redirection in UNIX/Linux

Linux redirection features can be used to detach the default files from stdin, stdout, and stderr and attach other files with them for a single execution of a command. The act of detaching defaults files from stdin, stdout, and stderr and attaching other files with them is known as input, output, and error redirection. In this section, we show the syntax, semantics, and examples of I/O and error redirection.

Input Redirection: Here is the syntax for input redirection:

```
command < input-file
```

or
```
command 0< input-file
```

With this command syntax, keyboard is detached from stdin of 'command' and 'input-file' is attached to it, i.e., 'command' reads input from 'input-file' and not keyboard. Note that 0< operator cannot be used with the C and TC shells. Here is an example use of input redirection. In these examples, the `cat` and `grep` commands read input from the Phones file and not from keyboard.

```
$ cat < Phones
[ contents of Phones ]
$ grep "Nauman" < Phones
[ output of grep ]
$
```

Output Redirection: Here is the syntax for output redirection:

```
command > output-file
```
or
```
command 1> output-file
```

Note that 1> operator cannot be used with the C and TC shells. Here is an example use of input redirection. In these examples, the `cat`, `grep`, and `find` commands send their outputs to the Phones, Ali.Phones, and foo.log files, respectively, and not to the display screen.

```
$ cat > Phones
[ your input ]
<Ctrl-D>
$ grep "Ali" Phones > Ali.phones
[ output of grep ]
$ find ~ -name foo -print > foo.log
[ error messages ]
$
```

Error Redirection: Here is the syntax for error redirection:

```
command 2> error-file
```

With this command syntax, the display screen is detached from stderr and 'error-file' is attached to it, i.e., error messages are sent to 'error-file' and not the display screen. Note that 2> cannot be used under C and TC shells. The following are a few examples of error redirection. In these examples, the first `find` command sends its error messages to the errors file and the second `find` command sends its error messages to the /dev/null file. The `ls` command sends its error messages to the error.log file and not to the display screen.

```
$ find ~ -name foo -print 2> errors
[ output of the find command ]
$ ls -l foo 2> error.log
[ output of the find command ]
$ cat error.log
ls: foo: No such file or directory
$ find / -name ls -print 2> /dev/null
/bin/ls
$
```

UNIX/Linux FIFOs

A named pipe (also called a named FIFO, or just FIFO) is a pipe whose access point is a file kept on the file system. By opening this file for reading, a process gets access to the FIFO for reading. By opening the file for writing, the process gets access to the FIFO for writing. By default, a FIFO is opened for blocking I/O. This means that a process reading from a FIFO blocks until another process writes some data in the FIFO. The same goes the other way around. Unnamed pipes can only be used between processes that have an ancestral relationship. And they are temporary; they need to be created every time and are destroyed when the corresponding processes exit. Named pipes (FIFOs) overcome both of these limitations. Figure 10.1 shows two unrelated processes, P1 and P2, communicating with each other using a FIFO.
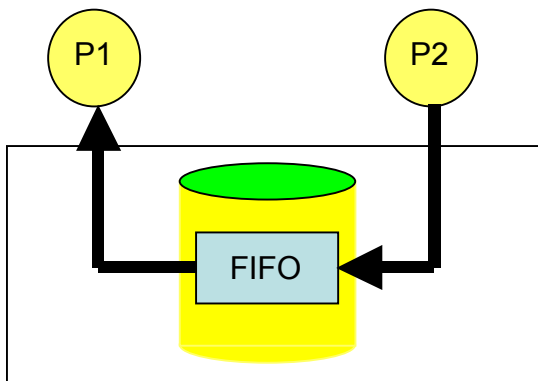
Figure 10.1 Communication between two related or unrelated processes on the same
UNIX/Linux machine

Named pipes are created via the `mknod()` system call or `mkfifo()` C library call
or by the `mkfifo` command. Here is the synopsis of the mknod() system call.

```
#include <sys/types.h>
#include <sys/stat.h>
int mknod (const char *path, mode_t mode, dev_t dev);
```

The `mknod()` call is normally used for creating special (i.e., device) files but it can be
used to create FIFOs too. The 'mode' argument should be permission mode OR-ed with
S_IFIFO and 'dev' is set to 0 for creating a FIFO. As is the case with all system calls in
UNIX/Linux, `mknod()` returns –1 on failure and *errno* is set accordingly. Some of the
reasons for this call to fail are:
- File with the given name exists
- Pathname too long
- A component in the pathname not searchable, non-existent, or non-directory
- Destination directory is read-only
- Not enough memory space available
- Signal caught during the execution of `mknod()`

Here is the synopsis of the `mkfifo()` library call.

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo (const char *path, mode_t mode)
```

The argument path is for the name and path of the FIFO created, where was the argument
mode is for specifying the file permissions for the FIFO. The specification of the mode
argument for this function is the same as for the `open()`. Once we have created a FIFO
using `mkfifo()`, we open it using `open()`. In fact, the normal file I/O system calls
(`close()`, `read()`, `write()`, `unlink()`, etc.) all works with FIFOs. Since
`mkfifo()` invokes the `mknod()` system call, the reasons for its failure are pretty much
the same as for the `mknod()` call given above.

Unlike a pipe, a FIFO must be opened before using it for communication. A write to a FIFO that no process has opened for reading results in a SIGPIPE signal. When the last process to write to a FIFO closes it, an EOF is sent to the reader. Multiple processes can write to a FIFO are atomic writes to prevent interleaving of multiple writes.

Two common uses of FIFOs are:

- In client-server applications, FIFOs are used to pass data between a server process and client processes
- Used by shell commands to pass data from one shell pipeline to another, without creating temporary files

In client-server software designed for use on the same machine, the server process creates a "well-known" FIFO. Clients communicate send their requests to the server process via the well-known FIFO. Server sends its response to a client via the client-specific FIFO that each client creates and informs the server process about it. Figure 10.2 shows the diagrammatic view of such a software model.
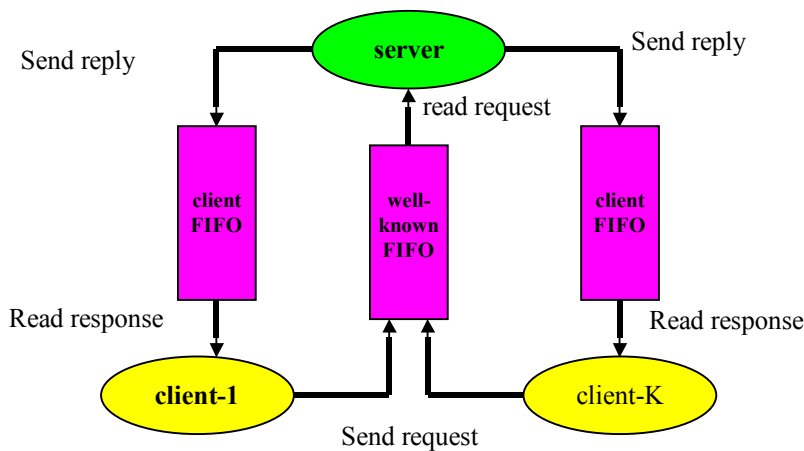
Figure 10.2   Use of FIFOs to implement client-server software on a UNIX/Linux machine

On the command line, `mkfifo` may be used as shown in the following session. As shown in Figure 10.3, the semantics of this session are that prog1 reads its inputs from infile and its output is sent to prog2 and prog3.

```
$ mkfifo fifo1
$ prog3 < fifo1 &
$ prog1 < infile | tee fifo1 | prog2
[ Output ]
$
```
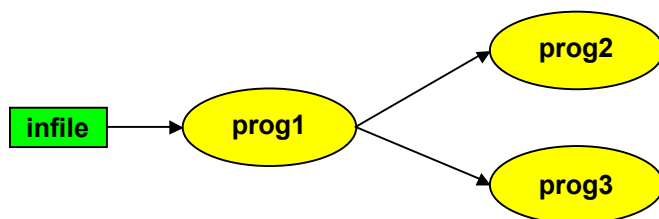

Figure 10.3 Semantics of the above shell session

In the following session, we demonstrate the command line use of FIFOs. The semantics of this session are shown in Figure 10.4. The output of the second command line is the number of lines in the ls.dat file containing ls (i.e., the number of lines in the manual page of the ls command containing the string ls) and the output of the third command line is the number of lines in the ls.dat file (i.e., the number of lines in the manual page for the ls command).

```
$ man ls > ls.dat
$ cat < fifo1 | grep ls | wc -l &
[1] 21108
$ sort < ls.dat | tee fifo1 | wc -l
  31
  528
$
```
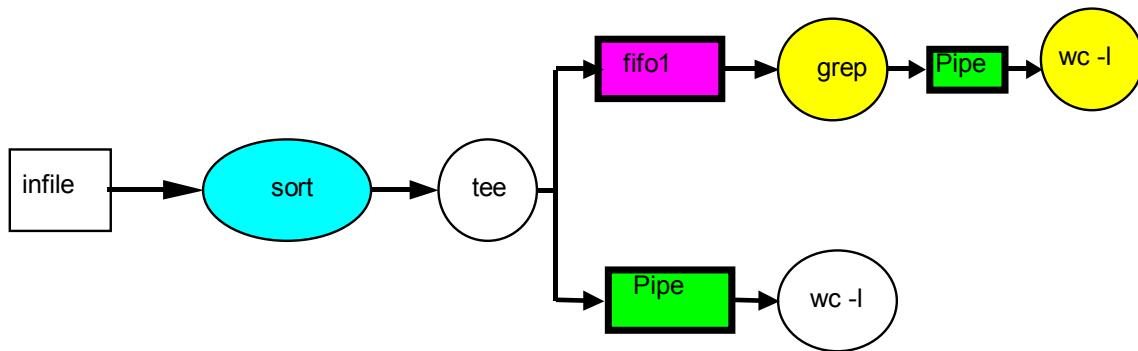
Figure 10.4 Pictorial representation of the semantics of the above shell session

Operating Systems
Lecture No. 11

Reading Material
- UNIX/Linux manual pages for the `mknod()` system call, the `mkfifo()` library call, and the `mkfifo`, `ps`, and `top` commands
- Lecture 11 on Virtual TV

Summary
- More on the use of FIFOs in a program
- Example code for a client-server model
- A few UNIX/Linux process management commands

Use of FIFOs

We continue to discuss the API for using FIFOs for IPC between UNIX/Linux processes. We call these processes client and server. The server process creates two FIFOs, FIFO1 and FIFO2, and opens FIFO1 for reading and FIFO2 for writing. The client process opens FIFO1 for writing and FIFO2 for reading. The client process writes a message to the server process and waits for a response from the server process. The server process reads the message sent by the client process and displays it on the monitor screen. It then sends a message to the client through FIFO2, which the client reads and displays on the monitor screen. The server process then closes the two FIFOs and terminates. The client, after displaying server's message, deletes the two FIFOs and terminates. The protocol for the client-server interaction is shown in Figure 10.1.
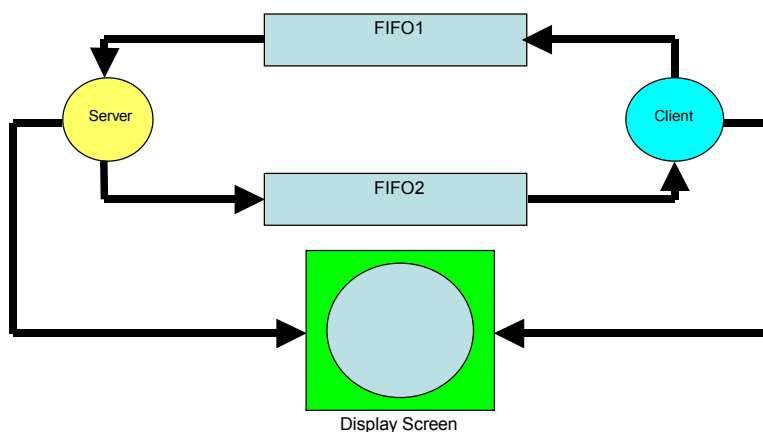
Figure 10.1 Client-server communication using UNIX/Linux FIFOs

The codes for the server and client processes are shown in Figure 10.2 and Figure 10.3, respectively.

```c
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>

extern int      errno;

#define FIFO1    "/tmp/fifo.1"
#define FIFO2    "/tmp/fifo.2"
#define PERMS    0666
#define MESSAGE1         "Hello, world!\n"
#define MESSAGE2         "Hello, class!\n"
#include "fifo.h"
main()
{
    char buff[BUFSIZ];
    int readfd, writefd;
    int n, size;

    if ((mknod (FIFO1, S_IFIFO | PERMS, 0) < 0)
                        && (errno != EEXIST)) {
        perror ("mknod FIFO1");
        exit (1);
    }
    if (mkfifo(FIFO2, PERMS) < 0) {
        unlink (FIFO1);
        perror("mknod FIFO2");
        exit (1);
    }
    if ((readfd = open(FIFO1, 0)) < 0) {
        perror ("open FIFO1");
        exit (1);
    }
    if ((writefd = open(FIFO2, 1)) < 0) {
        perror ("open FIFO2");
        exit (1);
    }
    size = strlen(MESSAGE1) + 1;
    if ((n = read(readfd, buff, size)) < 0) {
        perror ("server read"); exit (1);
    }
    if (write (1, buff, n) < n) {
        perror("server write1"); exit (1);
    }
    size = strlen(MESSAGE2) + 1;
    if (write (writefd, MESSAGE2, size) != size) {
        perror ("server write2"); exit (1);
    }
    close (readfd); close (writefd);
}
```

Figure 10.2 Code for the server process

```
#include "fifo.h"
main()
{
    char buff[BUFSIZ];
    int readfd, writefd, n, size;

    if ((writefd = open(FIFO1, 1)) < 0) {
        perror ("client open FIFO1"); exit (1);
    }
    if ((readfd = open(FIFO2, 0)) < 0) {
        perror ("client open FIFO2"); exit (1);
    }
    size = strlen(MESSAGE1) + 1;
    if (write(writefd, MESSAGE1, size) != size) {
        perror ("client write1"); exit (1);
    }
    if ((n = read(readfd, buff, size)) < 0) {
        perror ("client read"); exit (1);
    }
    else
        if (write(1, buff, n) != n) {
            perror ("client write2"); exit (1);
        }
    close(readfd); close(writefd);
     /* Remove FIFOs now that we are done using them */
    if (unlink (FIFO1) < 0) {
        perror("client unlink FIFO1");
        exit (1);
    }
    if (unlink (FIFO2) < 0) {
        perror("client unlink FIFO2");
        exit (1);
    }
    exit (0);
}
```

Figure 10.3 Code for the client process

In the session shown in Figure 10.4, we show how to compile and run the client-server software. We run the server process first so it could create the two FIFOs to be used for communication between the two processes. Note that the server process is run in the background by terminating its command line with an ampersand (&).

```
$ gcc server.c –o server
$ gcc client.c –o client
$ ./server &
[1] 432056
$ ./client
Hello, world!
Hello, class!
$
```

Figure 10.4 Compilation and execution of the client-server software

UNIX/Linux Command for Process Management

We now discuss some of the UNIX/Linux commands for process management, including `ps` and `top`. More commands will be discussed in lecture 12.

<u>ps – Display status of processes</u>

`ps` gives a snapshot of the current processes. Without options, ps prints information about processes owned by the user. Some of the commonly used options are `-u`, `-e`, and `-l`.

- `-e` selects all processes
- `-l` formats the output in the long format
- `-u` displays the information in user-oriented format

The shell session in Figure 10.5 shows sample use of the `ps` command. The first command shows the processes running in your current session. The second command shows, page by page, the status of all the processes belonging to root. The last command shows the status of all the processes running on your system.

```
$ ps
  PID TTY           TIME CMD
 1321 pts/0    00:00:00 csh
 1345 pts/0    00:00:00 bash
 1346 pts/0    00:00:00 ps
$ ps -u root | more
  PID TTY           TIME CMD
    1 ?         00:00:04 init
    5 ?         00:00:01 kswapd
  712 ?         00:00:00 inetd
  799 ?         00:00:00 cron
  864 ?         00:00:00 sshd
  934 ?         00:00:00 httpd
 1029 tty1      00:00:00 getty
...
$ ps -e | more
    PID TTY           TIME CMD
      1 ?        00:00:04 init
      2 ?        00:00:00 keventd
      3 ?        00:00:00 ksoftirqd_CPU0
      4 ?        00:00:00 ksoftirqd_CPU1
      5 ?        00:00:01 kswapd
      6 ?        00:00:00 kreclaimd
      7 ?        00:00:00 bdflush
      8 ?        00:00:00 kupdated
...
$
```

Figure 10.5 Use of the `ps` command

<u>top</u> – Display CPU usage of processes

top displays information about the top processes (as many as can fit on the terminal or around 20 by default) on the system and periodically updates this information. Raw CPU percentage is used to rank the processes. A sample run of the top command is shown in Figure 10.6. The output of the command also shows the current time, how long the system has been up and running, number of processes running on the system and their status, number of CPUs in the system and their usage, amount of main memory in the system and its usage, and the size of swap space and its usage. The output also shows a lot of information about each process, including process ID, owner's login name, priority, nice value, and size. Information about processes is updated periodically. See the manual page for the top command for more information by using the man top command.

```
$ top
9:42am  up  5:15,  2 users,  load average: 0.00, 0.00, 0.00
55 processes: 54 sleeping, 1 running, 0 zombie, 0 stopped
CPU0 states:  0.0% user,  0.1% system,  0.0% nice, 99.4% idle
CPU1 states:  0.0% user,  0.2% system,  0.0% nice, 99.3% idle
Mem:   513376K av,  237732K used,  275644K free,      60K shrd,   17944K buff
Swap:  257032K av,       0K used,  257032K free                  106960K cached

  PID USER      PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
 1406 sarwar     19   0   896  896   700 R     0.3  0.1   0:00 top
 1382 nobody     10   0   832  832   660 S     0.1  0.1   0:00 in.telnetd
    1 root        9   0   536  536   460 S     0.0  0.1   0:04 init
    2 root        9   0     0    0     0 SW    0.0  0.0   0:00 keventd
...
$
```

Figure 10.6 Use of the top command

Operating Systems
Lecture No. 12

Reading Material
- UNIX/Linux manual pages for `fg`, `bg`, `jobs`, and `kill` commands
- Chapter 5 of the textbook
- Lecture 12 on Virtual TV

Summary
- Process Management commands and key presses: `fg`, `bg`, `jobs`, and `kill` commands and `<Ctrl-Z>` and `<Ctrl-C>` command presses
- Thread Concept (thread, states, attributes, etc)

Process Management commands
In the last lecture, we started discussing a few UNIX/Linux process management command. In particular, we discussed the `ps` and `top` commands. We now discuss the `fg`, `bg`, `jobs`, and `kill` commands and `<Ctrl-Z>` and `<Ctrl-C>` key presses.

Moving a process into foreground
You can use the `fg` command to resume the execution of a suspended job in the foreground or move a background job into the foreground. Here is the syntax of the command.

```
fg [%job_id]
```

where, job_id is the job ID (not process ID) of the suspended or background process. If %job_id is omitted, the current job is assumed.

Moving a process into background
You can use the bg command to put the current or a suspended process into the background. Here is the syntax of the command.

```
bg [%job_id]
```

If %job_id is omitted the current job is assumed.

Displaying status of jobs (background and suspended processes)
You can use the jobs command to display the status of suspended and background processes.

Suspending a process
You can suspend a foreground process by pressing `<Ctrl-Z>`, which sends a STOP/SUSPEND signal to the process. The shell displays a message saying that the job has been suspended and displays its prompt. You can then manipulate the state of this

job, put it in the background with the `bg` command, run some other commands, and then eventually bring the job back into the foreground with the `fg` command.

The following session shows the use of the above commands. The `<Ctrl-Z>` command is used to suspend the `find` command and the `bg` command puts it in the background. We then use the `jobs` command to display the status of jobs (i.e., the background or suspended processes). In our case, the only job is the find command that we explicitly put in the background with the `<Ctrl-Z>` and `bg` commands.

```
$ find / -name foo -print 2> /dev/null
^Z
[1]+  Stopped    find / -name foo -print 2> /dev/null
$ bg
[1]+ find / -name foo -print 2> /dev/null &
$ jobs
[1]+  Running    find / -name foo -print 2> /dev/null &
$ fg
find / -name foo -print 2> /dev/null
[ command output ]
$
```

Terminating a process

==You can terminate a foreground process by pressing `<Ctrl-C>`.== Recall that this key press sends the SIGINT signal to the process and the default action is termination of the process. Of course, if your foreground process intercepts SIGINT and ignores it, you cannot terminate it with `<Ctrl-C>`. In the following session, we terminate the find command with `<Ctrl-C>`.

```
$ find / -name foo -print 1> out 2> /dev/null
^C
$
```

You can also terminate a process with the kill command. When executed, this command sends a signal to the process whose process ID is specified in the command line. Here is the syntax of the command.

```
kill [-signal] PID
```

where, 'signal' is the signal number and PID is the process ID of the process to whom the specified signal is to be sent. For example, `kill -2 1234` command sends signal number 2 (which is also called SIGINT) to the process with ID 1234. The default action for a signal is termination of the process identified in the command line. When executed without a signal number, the command sends the SIGTERM signal to the process. A process that has been coded to intercept and ignore a signal, can be terminated by sending it the 'sure kill' signal, SIGKILL, whose signal number is 9, as in `kill -9 1234`.

You can display all of the signals supported by your system, along with their numbers, by using the `kill -l` command. On some systems, the signal numbers are not displayed. Here is a sample run of the command on Solaris 2.

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGEMT       8) SIGFPE
 9) SIGKILL     10) SIGBUS      11) SIGSEGV     12) SIGSYS
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGUSR1
...
$
```

The Thread Concept

There are two main issues with processes:

1. The `fork()` system call is expensive (it requires memory to memory copy of the executable image of the calling process and allocation of kernel resources to the child process)
2. An inter-process communication channel (IPC) is required to pass information between a parent process and its children processes.

These problems can be overcome by using threads.

A thread, sometimes called a **lightweight process (LWP)**, is a basic unit of CPU utilization and executes within the address space of the process that creates it. It comprises a thread ID, a program counter, a register set, errno, and a stack. It shares with other threads belonging to the same process its code sections, data section, current working directory, user and group IDs, signal setup and handlers, PCB and other operating system resources, such as open files and system. A traditional (heavy weight) process has a single thread of control. If a process has multiple threads of control, it can do more than one task at a time. Figure 12.1 shows processes with single and multiple threads. Note that, as stated above, threads within a process share code, data, and open files, and have their own register sets and stacks.
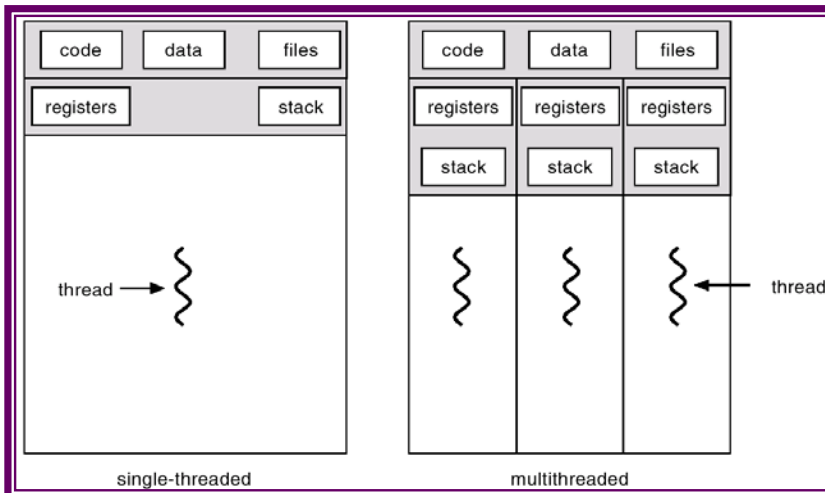
Figure 12.1 Single- and multi-threaded processes

In Figure 12.2, we show the code structure for a sequential (single-threaded) process and how the control thread moves from the main function to the f1 function and back, and from f1 to main and back. The important point to note here is that there is just one thread of control that moves around between various functions.

```
main()
{

    f1(

    f2(…);
    …
}
f1(…)
{ … }
f2(…)
{ … }
```

Thread

f1

F2

Process
Terminated
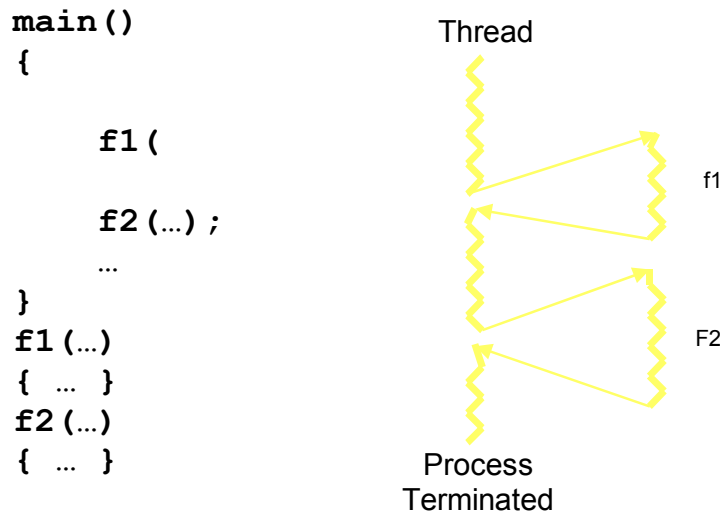
Figure 12.2 Code structure of a single-threaded (sequential) process

In Figure 12.3, we show the code structure for a multi-threaded process and how multiple threads of control are active simultaneously. We use hypothetical function thread() to create a thread. This function takes two arguments: the name of a function for which a thread has to be created and a variable in which the ID of the newly created thread is to be stored. The important point to note here is that multiple threads of control are simultaneously active within the same process; each thread steered by its own PC.

```
main()
{
    …
    thread(t1,f1);
    …
    thread(t2,f2);
    …
}
f1(…)
{ … }
f2(…)
{ … }
```

Process Address Space

main   t1   t2

PC

PC

PC
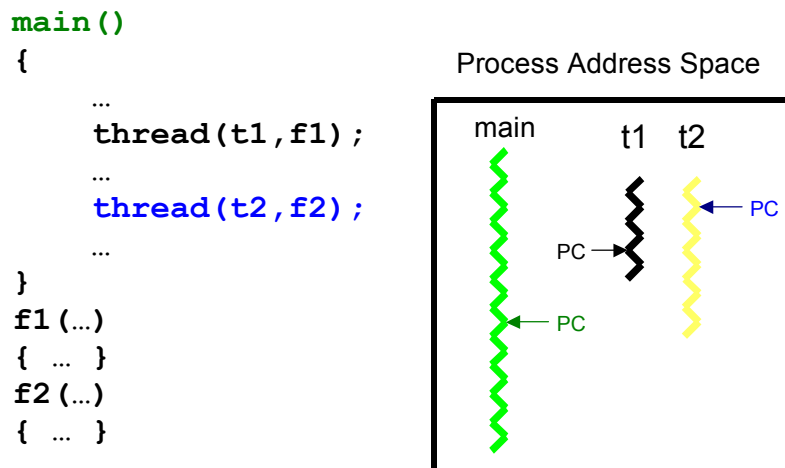
Figure 12.3 Code structure of a multi-threaded process

The Advantages and Disadvantages of Threads

Four main advantages of threads are:

1. <u>Responsiveness</u>: Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
2. <u>Resource sharing</u>: By default, threads share the memory and the resources of the process to which they belong. Code sharing allows an application to have several different threads of activity all within the same address space.
3. <u>Economy</u>: Allocating memory and resources for process creation is costly. Alternatively, because threads share resources of the process to which they belong, it is more economical to create and context switch threads.
4. <u>Utilization of multiprocessor architectures</u>: The benefits of multithreading of multithreading can be greatly increased in a multiprocessor environment, where each thread may be running in parallel on a different processor. A single threaded process can only run on one CPU no matter how many are available. Multithreading on multi-CPU machines increases concurrency.

Some of the main disadvantages of threads are:

1. <u>Resource sharing</u>: Whereas resource sharing is one of the major advantages of threads, it is also a disadvantage because proper synchronization is needed between threads for accessing the shared resources (e.g., data and files).
2. <u>Difficult programming model</u>: It is difficult to write, debug, and maintain multi-threaded programs for an average user. This is particularly true when it comes to writing code for synchronized access to shared resources.

Operating Systems
Lecture No. 13

Reading Material
- UNIX/Linux manual pages `pthread_create()`, `pthread_join()`, and `pthread_exit()` calls
- Chapter 5 of the textbook
- Lecture 13 on Virtual TV

Summary
- User- and Kernel –level threads
- Multi-threading models
- Solaris 2 threads model
- POSIX threads (the pthread library)
- Sample code

User and Kernel Threads
Support for threads may be provided at either user level for *user threads* or by kernel for *kernel threads.*

    User threads are supported above kernel and are implemented by a thread library at the user level. The library provides support for thread creation, scheduling, and management with no support from the kernel. Since the kernel is unaware of user-level threads, all thread creation and scheduling are done in the user space without the need for kernel intervention, and therefore are fast to create and manage. If the kernel is single threaded, then any user level thread performing a blocking system call will cause the entire process to block, even if other threads are available to run within the application. User thread libraries include POSIX Pthreads , Solaris 2 UI-threads, and Mach C-threads.

    Kernel threads are supported directly by the operating system. The kernel performs the scheduling, creation, and management in kernel space; the kernel level threads are hence slower to create and manage, compared to user level threads. However since the kernel is managing threads, if a thread performs a blocking system call, the kernel can schedule another thread in the application for execution. Windows NT, Windowss 2000, Solaris, BeOS and Tru64 UNIX support kernel threads.

Multi-threading Models
There are various models for mapping user-level threads to kernel-level threads. We describe briefly these models, their main characteristics, and examples.
1. **Many-to-One**: In this model, many user-level threads are supported per kernel thread, as shown in Figure 13.1. Since only one kernel-level thread supports many user threads, there is no concurrency. This means that a process blocks when a thread makes a system call. Examples of these threads are Solaris Green threads POSIX Pthreads.
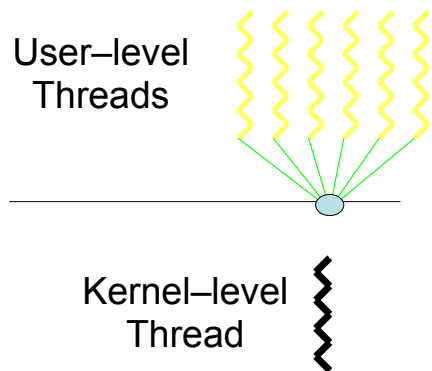
Figure 13.1 Many –to-One Model

2. **One-to-One**: In this model, there is a kernel thread for every user thread, as shown in Figure 13.2. Thus, this model provides true concurrency. This means that a process does not block when a thread makes a system call. The main disadvantage of this model is the overhead of creating a kernel thread per user thread. Examples of these threads are WindowsNT, Windows 2000, and OS/2.
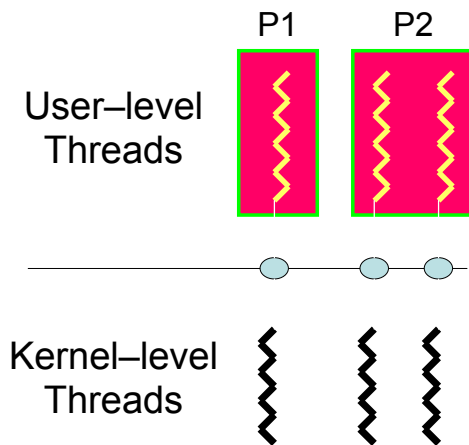
Figure 13.2 One-to-One Model

3. **Many-to-One**: In this model, multiple user-level threads are multiplexed over a smaller or equal number of kernel threads, as shown in Figure 13.2. Thus, true concurrency is not achieved through this model. Examples of these threads are Solais 2 and HP-UX.
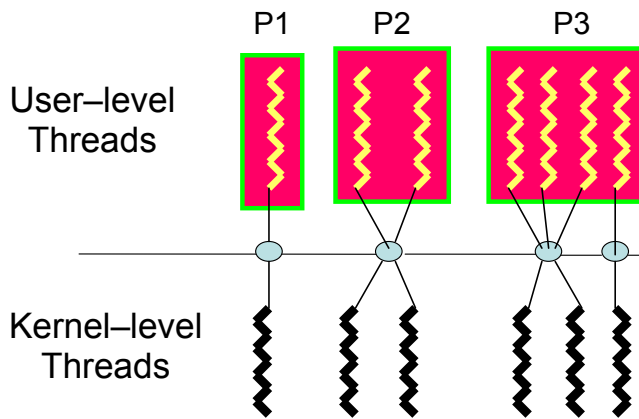
Figure 13.3 Many-to Many Model

Solaris 2 Threads Model

Solaris 2 has threads, lightweight processes (LWPs), and processes, as shown in Figure 13.4. At least one LWP is assigned to every user process to allow a user thread to talk to a kernel thread. User level threads are switched and scheduled among LWPs without kernel's knowledge. One kernel thread is assigned per LWP. Some kernel threads have no LWP associated with them because these threads are not executed for servicing a request by a user-level thread. Examples of such kernel threads are clock interrupt handler, swapper, and short-term (CPU) shceduler.
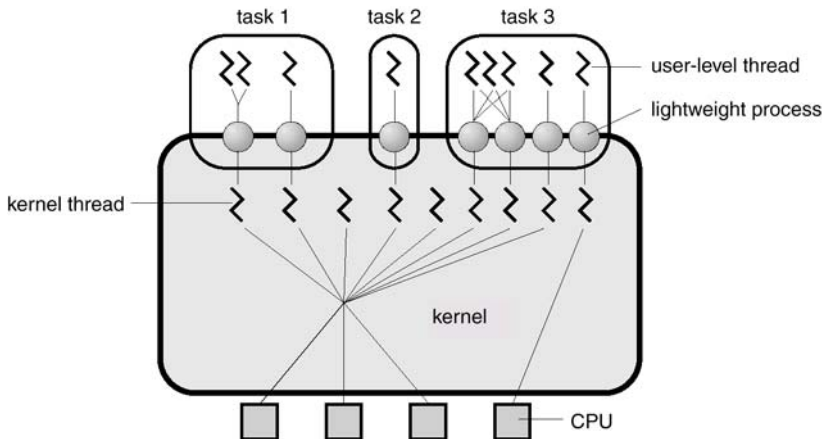
Figure 13.4 Solaris 2 Threads Model

POSIX Threads (the pthread library)

<mark>Pthreads refers to the POSIX standard defining an API for thread creation, scheduling, and synchronization.</mark> This is a specification for thread behavior not an implementation. OS designers may implement the specification in any way they wish. Generally, libraries implementing the Pthreads specification are restricted to UNIX-based systems such as Solaris 2. In this section, we discuss the Pthreads library calls for creating, joining, and terminating threads and use these calls to write small multi-threaded C programs.

Creating a Thread

You can create a threads by using the `pthread_create()` call. Here is the syntax of this call.

```
int pthread_create(pthread_t *threadp, const pthread_attr_t *attr,
                   void* (*routine)(void *), arg *arg);
```

where, 'threadp' contains thread ID (TID) of the thread created by the call, 'attr' is used to modify the thread attributes (stack size, stack address, detached, joinable, priority, etc.), 'routine' is the thread function, and 'arg' is any argument we want to pass to the thread function. The argument does not have to be a simple native type; it can be a 'struct' of whatever we want to pass in.

The `pthread_create()` call fails and returns the corresponding value if any of the following conditions is detected:

- **EAGAIN** The system-imposed limit on the total number of threads in a process has been exceeded or some system resource has been exceeded (for example, too many LWPs were created).
- **EINVAL** The value specified by 'attr' is invalid.
- **ENOMEM** Not enough memory was available to create the new thread.

You can do error handling by including the <errno.h> file and incorporating proper error handling code in your programs.

Joining a Thread

You can have a thread wait for another thread within the same process by using the pthread_join() call. Here is the syntax of this call.

```
int pthread_join(pthread_t aThread, void **statusp);
```

where, 'aThread' is the thread ID of the thread to wait for and 'statusp' gets the return value of `pthread_exit()` call made in the process for whom wait is being done.

A thread can only wait for a joinable thread in the same process address space; a thread cannot wait for a detached thread. Multiple threads can join with a thread but only one returns successfully; others return with an error that no thread could be found with the given TID

Terminating a Thread

You can terminate a thread explicitly by either returning from the thread function or by using the `pthread_exit()` call. Here is the syntax of the `pthread_exit()` call.

```
void pthread_exit(void *valuep);
```

where, 'valuep' is a pointer to the value to be returned to the thread which is waiting for this thread to terminate (i.e., the thread which has executed `pthread_join()` for this thread).

A thread also terminates when the main thread in the process terminates. When a thread terminates with the exit() system call, it terminates the whole process because the purpose of the exit() system call is to terminate a process and not a thread.

Sample Code

The following code shows the use of the pthread library calls discussed above. The program creates a thread and waits for it. The child thread displays the following message on the screen and terminates.

```
Hello, world! ... The threaded version.
```

As soon as the child thread terminates, the parent comes out of wait, displays the following message and terminates.

```
Exiting the main function.
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
/* Prototype for a function to be passed to our thread */
void* MyThreadFunc(void *arg);
int main()
{
   pthread_t aThread;
   /* Create a thread and have it run the MyThreadFunction */
   pthread_create(&aThread, NULL, MyThreadFunc, NULL);
   /* Parent waits for the aThread thread to exit */
   pthread_join(aThread, NULL);
   printf ("Exiting the main function.\n");
   return 0;
}
void* MyThreadFunc(void* arg)
{
        printf ("Hello, world! ... The threaded version.\n");
        return NULL;
}
```

The following session shows compilation and execution of the above program. Does the output make sense to you?

```
$ gcc hello.c –o hello –lpthread –D_REENTRANT
$ hello
Hello, world! ... The threaded version.
Exiting the main function.
$
```

Note that you need to take the following steps in order to be able to use the pthread library.

1. Include <pthread.h> in your program
2. Link the pthread library with your program (by using the –lpthread option in the compiler command)
3. Pass the _REENTRANT macro from the command line (or define it in your program)

Here is another program that uses the pthread library to create multiple threads and have them display certain messages. Read through the code to understand what it does. Then compile and run it on your UNIX/Linux system to display output of the program and to see if you really understood the code.

```c
/**********************************************************************
* FILE: hello_arg2.c
* DESCRIPTION:
*   A "hello world" Pthreads program which demonstrates another safe way
*   to pass arguments to threads during thread creation.  In this case,
*   a structure is used to pass multiple arguments.
*
* LAST REVISED: 09/04/02 Blaise Barney
**********************************************************************/
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS   8

char *messages[NUM_THREADS];

struct thread_data
{
  int      thread_id;
  int  sum;
  char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
  int taskid, sum;
  char *hello_msg;
  struct thread_data *my_data;

  sleep(1);
  my_data = (struct thread_data *) threadarg;
  taskid = my_data->thread_id;
  sum = my_data->sum;
  hello_msg = my_data->message;
  printf("Thread %d: %s  Sum=%d\n", taskid, hello_msg, sum);
  pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
```

```
    pthread_t threads[NUM_THREADS];
    int *taskids[NUM_THREADS];
    int rc, t, sum;

    sum=0;
    messages[0] = "English: Hello World!";
    messages[1] = "French: Bonjour, le monde!";
    messages[2] = "Spanish: Hola al mundo";
    messages[3] = "Klingon: Nuq neH!";
    messages[4] = "German: Guten Tag, Welt!";
    messages[5] = "Russian: Zdravstvytye, mir!";
    messages[6] = "Japan: Sekai e konnichiwa!";
    messages[7] = "Latin: Orbis, te saluto!";

    for(t=0; t<NUM_THREADS; t++) {
      sum = sum + t;
      thread_data_array[t].thread_id = t;
      thread_data_array[t].sum = sum;
      thread_data_array[t].message = messages[t];
      printf("Creating thread %d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &thread_data_array[t]);
      if (rc) {
        printf("ERROR; return code from pthread_create() is %d\n", rc);
        exit(-1);
      }
    }
    pthread_exit(NULL);
}
```

Reference

The above code was taken from the following website.

```
http://www.llnl.gov/computing/tutorials/pthreads/samples/hello_arg2.c
```

Operating Systems
Lecture No. 14

Reading Material
- Chapter 6 of the textbook
- Lecture 14 on Virtual TV

Summary
- Basic concepts
- Scheduling criteria
- Preemptive and non-preemptive algorithms
- First-Come-First-Serve scheduling algorithm

Basic Concepts
The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization. In a uniprocessor system, only one process may run at a time; any other processes much wait until the CPU is free and can be rescheduled.
In multiprogramming, a process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU would then sit idle; all this waiting time is wasted. Multiprogramming entails productive usage of this time. When one process has to wait, the OS takes the CPU away from that process and gives the CPU to another process. Almost all computer resources are scheduled before use.

Life of a Process
As shown in Figure 14.1, process execution consists of a cycle of CPU execution and I/O wait. Processes alternates between these two states. Process execution begins with a **CPU burst.** An I/O burst follows that, and so on**.** Eventually, the last CPU burst will end with a system request to terminate execution, rather than with another I/O burst.

An I/O bound program would typically have many very short CPU bursts. A CPU-bound program might have a few very long CPU bursts. This distribution can help us select an appropriate CPU-scheduling algorithm. Figure 14.2 shows results on an empirical study regarding the CPU bursts of processes. The study shows that most of the processes have short CPU bursts of 2-3 milliseconds.
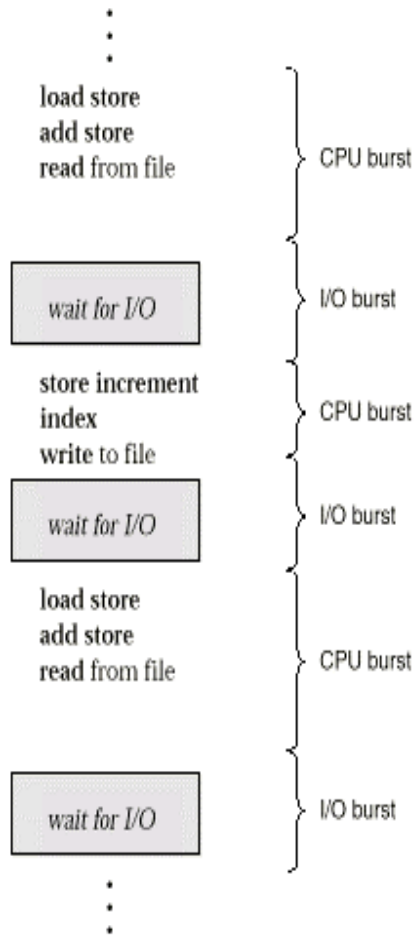
Figure 14.1 Alternating Sequence of CPU and I/O Bursts
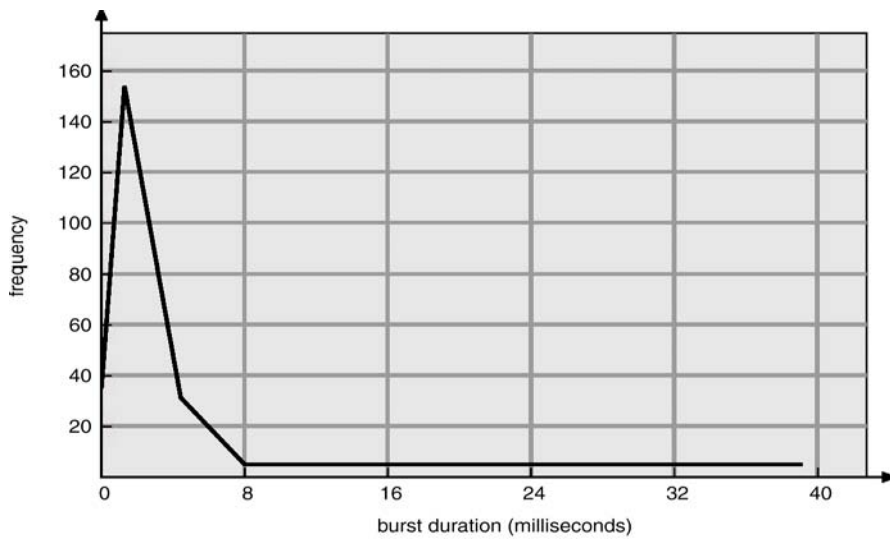
Figure 14.2 Histogram of CPU-burst Times

CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The **short-term scheduler** (i.e., the CPU scheduler) selects a process to give it the CPU. It selects from among the processes in memory that are ready to execute, and invokes the dispatcher to have the CPU allocated to the selected process.

A ready queue can be implemented as a FIFO queue, a tree, or simply an unordered linked list. The records (nodes) in the ready queue are generally the process control blocks (PCBs) of processes.

Dispatcher

The **dispatcher** is a kernel module that takes control of the CPU from the current process and gives it to the process selected by the short-term scheduler. This function involves:

- Switching the context (i.e., saving the context of the current process and restoring the context of the newly selected process)
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

Preemptive and Non-Preemptive Scheduling

CPU scheduling can take place under the following circumstances:
1. When a process switches from the running state to the waiting state (for example, an I/O request is being completed)
2. When a process switches from the running state to the ready state (for example when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, completion of I/O)
4. When a process terminates

In 1 and 4, there is no choice in terms of scheduling; a new process must be selected for execution. There is a choice in case of 2 and 3. When scheduling takes place only under 1 and 4, we say, scheduling is **non-preemptive**; otherwise the scheduling scheme is **preemptive**. Under non-preemptive scheduling once the CPU has been allocated to a process the process keeps the CPU until either it switches to the waiting state, finishes its CPU burst, or terminates. This scheduling method does not require any special hardware needed for preemptive scheduling.

Preemptive scheduling incurs a cost. Consider the case of two processes sharing data. One may be in the midst of updating the data when it is preempted and the second process is run. The second process may try to read the data, which are currently in an inconsistent state. New mechanisms are needed to coordinate access to shared data. We discuss this topic in Chapter 7 of the textbook.

Scheduling Criteria

The scheduling criteria include:

- **CPU utilization**: We want to keep CPU as busy as possible. In a real system it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system)
- **Throughput:** If CPU is busy executing processes then work is being done. One measure of work is the number of processes completed per time, called, **throughput.** We want to maximize the throughput.
- **Turnaround time:** The interval from the time of submission to the time of completion is the **turnaround time.** Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU and doing I/O. We want to minimize the turnaround time.
- **Waiting time:** Waiting time is the time spent waiting in the ready queue. We want to minimize the waiting time to increase CPU efficiency.
- **Response time:** It is the time from the submission of a request until the first response is produced. Thus response time is the amount of time it takes to start responding but not the time it takes to output that response. Response time should be minimized.

Scheduling Algorithms

We will now discuss some of the commonly used short-term scheduling algorithms. Some of these algorithms are suited well for batch systems and others for time-sharing systems. Here are the algorithms we will discuss:

- First-Come-First-Served (FCFS) Scheduling
- Shorted Job First (SJF) Scheduling
- Shortest Remaining Time First (SRTF) Scheduling
- Priority Scheduling
- Round-Robin Scheduling
- Multilevel Queues Scheduling
- Multilevel Feedback Queues Scheduling
- UNIX System V Scheduling

First-Come, First-Served (FCFS) Scheduling

The process that requests the CPU first (i.e., enters the ready queue first) is allocated the CPU first. The implementation of an FCFS policy is managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When CPU is free, it is allocated to the process at the head of the queue. The running process is removed from the queue. The average waiting time under FCFS policy is not minimal and may vary substantially if the process CPU-burst times vary greatly. FCFS is a non-preemptive scheduling algorithm.

We use the following system state to demonstrate the working of this algorithm. For simplicity, we assume that processes are in the ready queue at time 0.

Process	Burst Time
P1	24
P2	3
P3	3

Suppose that processes arrive into the system in the order: P1, P2, P3. Processes are served in the order: P1, P2, P3.The **Gantt chart** for the schedule is shown in Figure 14.3.

P₁... (Gantt chart)

| P_1 | P_2 | P_3 |

0   2   2   3
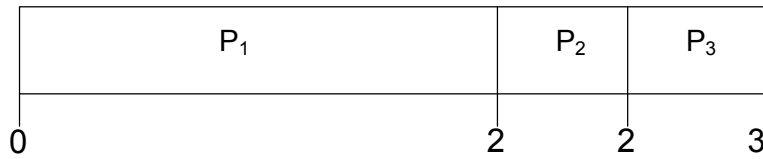
Figure 14.3 Gantt chart showing execution of processes in the order P1, P2, P3

Here are the waiting times for the three processes and the average waiting time per process.
- Waiting times P1 = 0; P2 = 24; P3 = 27
- Average waiting time: (0+24+27)/3 = 17

Suppose that processes arrive in the order: P2, P3, P1. The Gantt chart for the schedule is shown in Figure 14.4:

| P_2 | P_3 | P_1 |

0   3   6   30
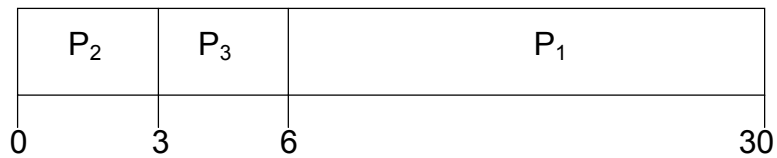
Figure 14.4 Gantt chart showing execution of processes in the order P2, P3, P1

Here are the waiting times for the three processes and the average waiting time per process.
- Waiting time for P1 = 6; P2 = 0; P3 = 3
- Average waiting time: (6 + 0 + 3)/3 = 3

When FCFS scheduling algorithm is used, the **convoy effect** occurs when short processes wait behind a long process to use the CPU and enter the ready queue in a convoy after completing their I/O. This results in lower CPU and device utilization than might be possible if shorter processes were allowed to go first.

In the next lecture, we will discuss more scheduling algorithms.

Operating Systems
Lecture No. 15

Reading Material
- Chapter 6 of the textbook
- Lecture 15 on Virtual TV

Summary
- Scheduling algorithms

Shortest-Job-First Scheduling

This algorithm associates with ach process the length of the latter's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie. The real difficulty with the SJF algorithm is in knowing the length of the next CPU request. For long term scheduling in a batch system, we can use as the length the process time limit that a user specifies when he submits the job.

For short-term CPU scheduling, there is no way to length of the next CPU burst. One approach is to try to approximate SJF scheduling, by assuming that the next CPU burst will be similar in length to the previous ones, for instance.

The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts. Let t_n be the length of the nth CPU burst and let τ_{n+1} be our predicted value for the next CPU burst. We define τ_{n+1} to be

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \tau_n$$

where, $0 \leq \alpha \leq 1$. We discuss this equation in detail in a subsequent lecture.

The SJF algorithm may either be preemptive or non-preemptive. The choice arises when a new process arrives at the ready queue while a previous process is executing. The new process may have a shorter next CPU burst than what is left of the currently executing process. A preemptive SJF algorithm preempts the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first** scheduling.

We illustrate the working of the SJF algorithm by using the following system state.

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

The Gantt chart for the execution of the four processes using SJF is shown in Figure 15.1.
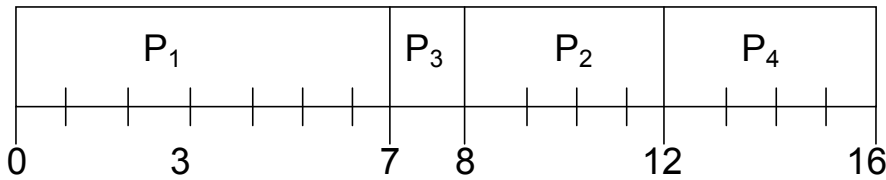
Figure 15.1 Gantt chart showing execution of processes using SJF

Here is the average waiting time per process.

- Average waiting time = (0 + 6 + 3 + 7)/4 = 4 time units

We illustrate the working of the SRTF algorithm by using the system state shown above. The Gantt chart for the execution of the four processes using SRTF is shown in Figure 15.2.
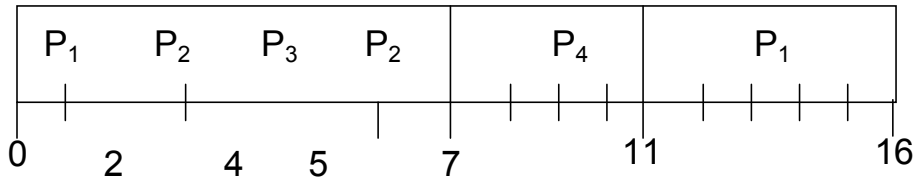
Figure 15.2 Gantt chart showing execution of processes using SRTF

Here is the average waiting time per process.

- Average waiting time = (9 + 1 + 0 +2)/4 = 3 time units

Priority Scheduling

SJF is a special case of the general **priority-scheduling algorithm**. A priority is associated with each process, and the CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority). Equal priority processes are scheduled in FCFS order. The SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst of a process, the lower its priority, and vice versa.

Priority scheduling can either be preemptive or non-preemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority-scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A non-preemptive priority- scheduling algorithm will simply put the new process at the head of ready queue.

A major problem with priority- scheduling algorithms is **indefinite blocking** (or **starvation**). A process that is ready to run but lacking the CPU can be considered blocked-waiting for the CPU. A priority-scheduling algorithm can leave some low priority processes waiting indefinitely for the CPU. Legend has it that when they were phasing out IBM 7094 at MIT in 1973, they found a process stuck in the ready queue since 1967!

Aging is solution to the problem of indefinite blockage of low-priority processes. It involves gradually increasing the priority of processes that wait in the system for a long time. For example, if priority numbers range from 0 (high priority) to 127 (high priority), we could decrement priority of every process periodically (say every 10 minutes). This would result in every process in the system eventually getting the highest priority in a reasonably short amount of time and scheduled to use the CPU.

Operating Systems
Lecture No. 16

Reading Material
- Chapter 6 of the textbook
- Lecture 16 on Virtual TV

Summary
- Scheduling algorithms

Why is SJF optimal?

SJF is an optimal algorithm because it decreases the wait times for short processes much more than it increases the wait times for long processes. Let's consider the example shown in Figure 16.1, in which the next CPU bursts of P1, P2, and P3 are 5, 3, and 2, respectively. The first Gantt chart shows execution of processes according to the longest-job-first algorithm, resulting in the waiting times for P1, P2, and P3 to be 0, 5, and 8 times units. The second Gantt chart shows execution of processes according to the shortest-job-first algorithm, resulting in the waiting times for P1, P2, and P3 to be 0, 2, and 5. Note that the waiting time for P2 has decreased from 5 to 2 and that of P3 has decreased from 8 to 0. The increase in the wait time for P1 is from 0 to 5, which is much smaller than the decrease in the wait times for P2 and P3.

P1	P2	P3

P3	P2	P1
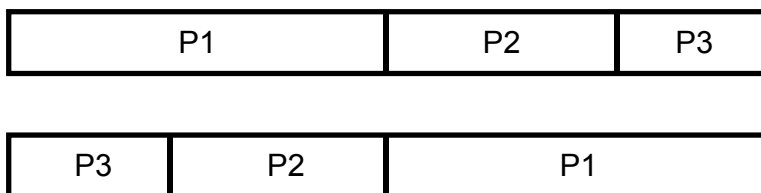
Figure 16.1   Two execution sequences for P1, P2, and P3: longest-job-first and shortest-job-first

Round-Robin Scheduling

The **round-robin (RR) scheduling algorithm** is designed especially for time-sharing systems. It is similar to FCFS scheduling but preemption is added to switch between processes. A small unit of time, called a **time quantum** (or **time slice**) is defined. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum. To implement RR scheduling, we keep ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and then dispatches the process. One of the two things will then happen. The process may have a CPU burst of less than 1 time quantum, in which case the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of currently running process is longer than one time

quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will happen, the current process will be put at the tail of the ready queue, and the newly scheduled process will be given the CPU.

The average waiting time under the RR policy however is often quite long. It is a preemptive scheduling algorithm. If there are n processes n the ready queue, context switch time is t_{cs} and the time quantum is q then each process gets $1/n$ of the CPU time in chunks of at most q time units. Each process must wait no longer than $(n-1)*(q+t_{cs})$ time units until its next time quantum.

The performance of RR algorithm depends heavily on the size of the time quantum. If the time quantum is very large (infinite), the RR policy remains the same as the FCFS policy. If the time quantum is very small, the RR approach is called the **processor sharing** and appears to the users as though each of n processes has its own processor running at $1/n$ the speed of real processor (q must be large with respect to context switch, otherwise the overhead is too high). The drawback of small quantum is more frequent context switches. Since context switching is the cost of the algorithm and no useful work is done for any user process during context switching, the number of context switches should be minimized and the quantum should be chosen such that the ratio of a quantum to context switching is not less than 10:1 (i.e., context switching overhead should not be more than 10% of the time spent on doing useful work for a user process). Figure 16.2 shows increase in the number of context switches with decrease in quantum size.
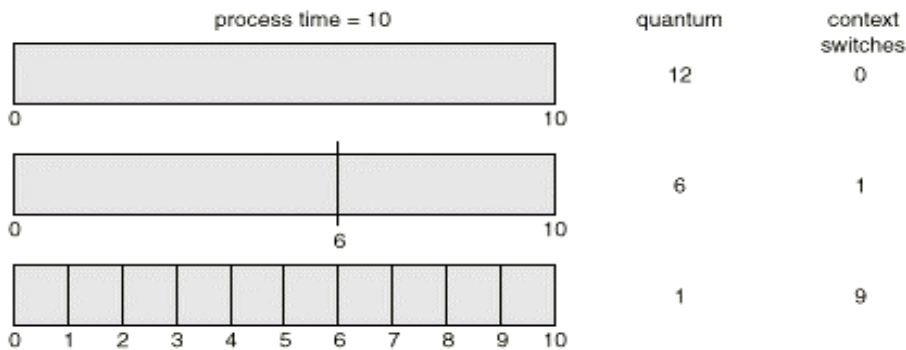
Figure 16.2 Quantum size versus number of context switches

The turnaround time of a process under round robin is also depends on the size of the time quantum. In Figure 16.3 we show a workload of four processes P1, P2, P3, and P4 with their next CPU bursts as 6, 3, 1, and 7 time units. The graph in the figure shows that best (smallest) turnaround time is achieved when quantum size is 6 or greater. Note that most of the given processes finish their next CPU bursts with quantum of 6 or greater. We can make a general statement that the round-robin algorithm gives smallest average turnaround time when quantum value is chosen such that most of the processes finish their next CPU bursts within the quantum.

process	time
P_1	6
P_2	3
P_3	1
P_4	7
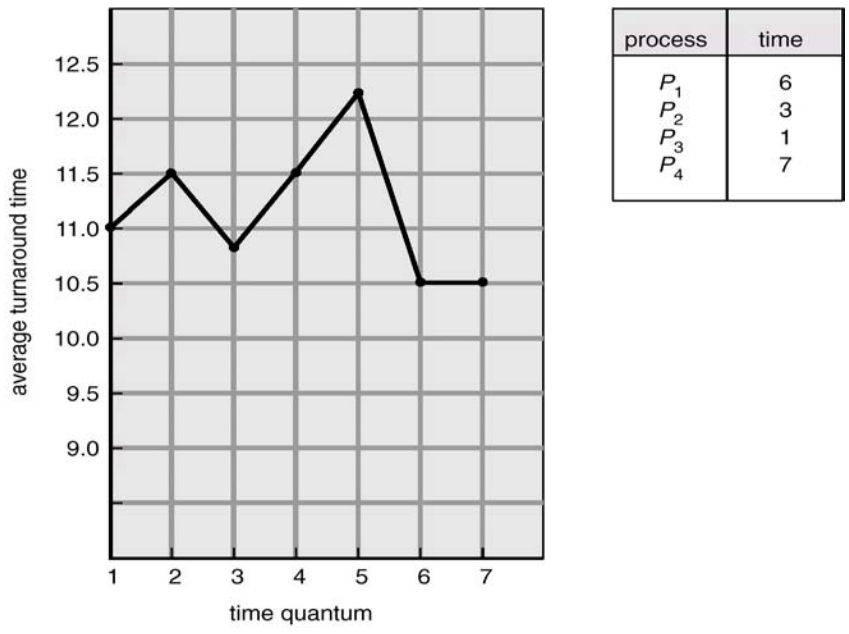
Figure 16.3 Turnaround time versus quantum size

We now consider the following system workload to illustrate working of the round-robin algorithm. Execution of P1 though P4 with quantum 20 is shown in Figure 16.4. In the table, original CPU bursts are shown in bold and remaining CPU bursts (after a process has used the CPU for one quantum) are shown in non-bold font.

Process	Burst Time
P1	**53** — 33 — 13
P2	**17**
P3	**68** — 48 — 28 — 8
P4	**24** — 4
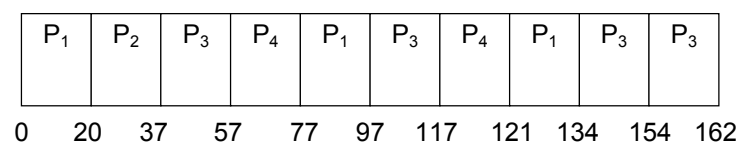
Figure 16.4 Gantt chart showing execution of P1, P2, P3, and P4 with quantum 20 time units

Figure 16.5 shows wait and turnaround times for the four processes. The average wait time for a process comes out to be 73 time units for round robin and 38 for SJF. Typically, RR has a higher average turnaround than SJF, but better response. In time-sharing systems, shorter response time for a process is more important than shorter turnaround time for the process. Thus, round-robin scheduler matches the requirements of time-sharing systems better than the SJF algorithm. SJF scheduler is better suited for batch systems, in which minimizing the turnaround time is the main criterion.

Process	Turnaround Time	Waiting Time
P1	134	134 – 53 = **81**
P2	37	37 – 17 = **20**
P3	162	162 – 68 = **94**
P4	121	121 – 24 = **97**

Figure 16.5   Wait and turnaround times for processes

Multilevel Queue Scheduling
Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. For example, a common division is made between **foreground** (or **interactive**) processes and **background** (or **batch) processes**. These two types of processes have different response time requirements and so might have different scheduling needs. In addition, foreground processes may have priority over background processes.

   A **multilevel queue-scheduling algorithm** partitions the ready queue into several separate queues, as shown in Figure 16.5. Each queue has its own priority and scheduling algorithm. Processes are permanently assigned to one queue, generally based o some property of the process, such as memory size, process priority or process type. In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling i.e., serve all from foreground then from background. Another possibility is to time slice between queues. Each queue gets a certain portion of the CPU time, which it can then schedule among the various processes in its queue, e.g., 80% to foreground in RR and 20% to background in FCFS. Scheduling across queues prevents starvation of processes in lower-priority queues.
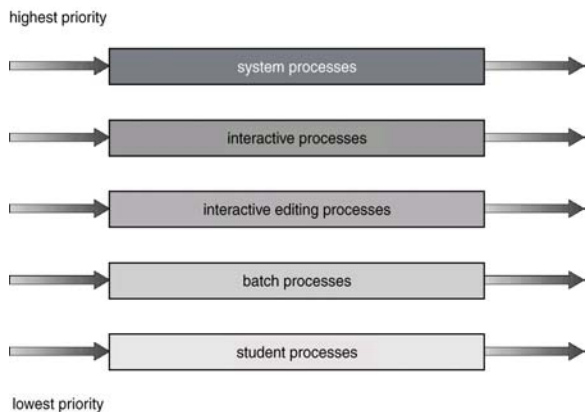
Figure 16.5   Multilevel queues scheduling

Operating Systems
Lecture No. 17

Reading Material
- Chapter 6 of the textbook
- Lecture 16 on Virtual TV

Summary
- Scheduling algorithms
- UNIX System V scheduling algorithm
- Optimal scheduling
- Algorithm evaluation

Multilevel Feedback Queue Scheduling

Multilevel feedback queue scheduling allows a process to move between queues. The idea is to separate processes with different CPU burst characteristics. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O bound and interactive processes in the higher-priority queues. Similarly a process that waits too long in a lower-priority queue may be moved o a higher priority queue. This form of aging prevents starvation.

In general, a multi-level feedback queue scheduler is defined by the following parameters:
- Number of queues
- Scheduling algorithm for each queue
- Method used to determine when to upgrade a process to higher priority queue
- Method used to determine when to demote a process
- Method used to determine which queue a process enters when it needs service

Figure 17.1 shows an example multilevel feedback queue scheduling system with the ready queue partitioned into three queues. In this system, processes with next CPU bursts less than or equal to 8 time units are processed with the shortest possible wait times, followed by processes with CPU bursts greater than 8 but no greater than 16 time units. Processes with CPU greater than 16 time units wait for the longest time.
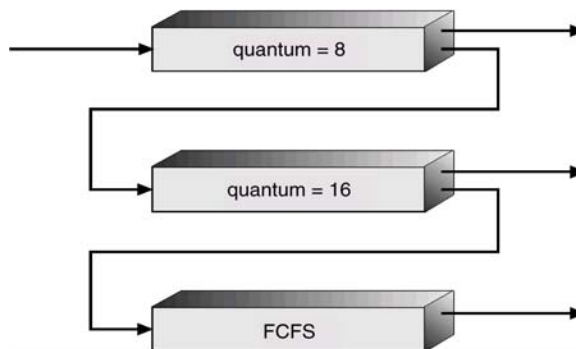
Figure 17.1 Multilevel Feedback Queues Scheduling

UNIX System V scheduling algorithm

UNIX System V scheduling algorithm is essentially a multilevel feedback priority queues algorithm with round robin within each queue, the quantum being equal to1 second. The priorities are divided into two groups/bands:
- Kernel Group
- User Group

Priorities in the Kernel Group are assigned in a manner to minimize bottlenecks, i.e, processes waiting in a lower-level routine get higher priorities than those waiting at relatively higher-level routines. We discuss this issue in detail in the lecture with an example. In decreasing order of priority, the kernel bands are:
- Swapper
- Block I/O device control processes
- File manipulation
- Character I/O device control processes
- User processes

The priorities of processes in the Kernel Group remain fixed whereas the priorities of processes in the User Group are recalculated every second. Inside the User Group, the CPU-bound processes are penalized at the expense of I/O-bound processes. Figure 17.2 shows the priority bands for the various kernel and user processes.
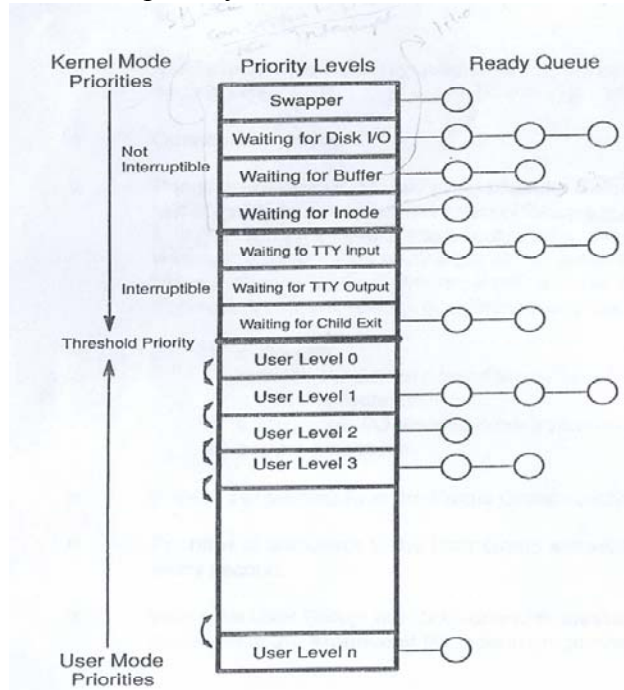
Figure 17.2. UNIX System V Scheduling Algorithm

Every second, the priority number of all those processes that are in the main memory and ready to run is updated by using the following formula:

Priority # = (Recent CPU Usage)/2 + Threshold Priority + nice

The 'threshold priority' and 'nice' values are always positive to prevent a user from migrating out of its assigned group and into a kernel group. You can change the nice value of your process with the `nice` command.

In Figure 17.3, we illustrate the working of the algorithm with an example. Note that recent CPU usage of the current process is updated every clock tick; we assume that clock interrupt occurs every sixtieth of a second. The priority number of every process in the ready queue is updated every second and the decay function is applied before recalculating the priority numbers of processes.

	P_A		P_B		P_C	
Time	Priority	CPU Count	Priority	CPU Count	Priority	CPU Count
0	60	0 1 ... 60	60	0	60	0
1	75	30	60	0 1 ...	60	0
2	67	30 15	75	60 30	60	0 1 ... 60
3	63	7 8 ...	67	30 15	75	30
4	76	67 33	63	7 8 ... 67	67	30 15
5	68	16	76	67 33	63	7

Figure 17.3 Illustration of the UNIX System V Scheduling Algorithm

Figure 17.4 shows that in case of a tie, processes are scheduled on First-Come-First-Serve basis.

60

Higher
Priority

B

B

A

B

A

A

B
A

A   B

B

A runs first
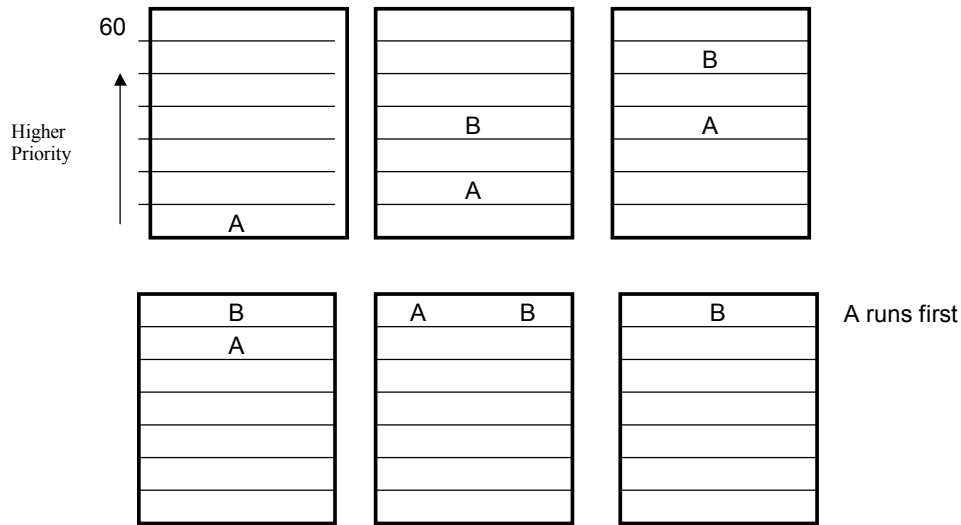
Figure 17.4 FCFS Algorithm is Used in Case of a Tie

Algorithm Evaluation

To select an algorithm, we must take into account certain factors, defining their relative importance, such as:

- Maximum CPU utilization under the constraint that maximum response time is 1 second.
- Maximize throughput such that turnaround time is (on average) linearly proportional to total execution time.

Scheduling algorithms can be evaluated by using the following techniques:

Analytic Evaluation
A scheduling algorithm and some system workload are used to produce a formula or number, which gives the performance of the algorithm for that workload. Analytic evaluation falls under two categories:

Deterministic modeling
Deterministic modeling is a type of analytic evaluation. This method takes a particular predetermined workload and defines the performance of each algorithm for workload in terms of numbers for parameters such as average wait time, average turnaround time, and average response time. Gantt charts are used to show executions of processes. We have been using this technique to explain the working of an algorithm as well as to evaluate the performance of an algorithm with a given workload.

Deterministic modeling is simple and fast. It gives exact numbers, allowing the algorithms to be compared. However it requires exact numbers for input and its answers apply to only those cases.

Queuing Models
The computer system can be defined as a network of servers. Each server has a queue of waiting processes. The CPU is a server with its ready queue, as are I/O systems with their device queues. Knowing the arrival and service rates of processes for various servers, we can compute utilization, average queue length, average wait time, and so on. This kind of study is called **queuing-network analysis**. If n is the average queue length, W is the

average waiting time in the queue, and let λ is the average arrival rate for new processes in the queue, then

$$n = \lambda * W$$

This formula is called the **Little's formula**, which is the basis of **queuing theory**, a branch of mathematics used to analyze systems involving queues and servers.

At the moment, the classes of algorithms and distributions that can be handled by queuing analysis are fairly limited. The mathematics involved is complicated and distributions can be difficult to work with. It is also generally necessary to make a number of independent assumptions that may not be accurate. Thus so that they will be able to compute an answer, queuing models are often an approximation of real systems. As a result, the accuracy of the computed results may be questionable.

The table in Figure 17.5 shows the average waiting times and average queue lengths for the various scheduling algorithms for a pre-determined system workload, computed by using Little's formula. The average job arrival rate is 0.5 jobs per unit time.

Algorithm	Average Wait Time $W = t_w$	Average Queue Length (n)
FCFS	4.6	2.3
SJF	3.6	1.8
SRTF	3.2	1.6
RR (q=1	7.0	3.5
RR (q=4)	6.0	3.0

Figure 17.5   Average Wait Time and Average Queue Length Computed With Little's Equation

Simulations
Simulations involve programming a model of the computer system, in order to get a more accurate evaluation of the scheduling algorithms. Software date structures represent the major components of the system. The simulator has a variable representing a clock; as this variable's value is increased, the simulator modifies the system state to reflect the activities of the devices, the processes and the scheduler. As the simulation executes, statistics that indicate algorithm performance are gathered and printed. Figure 17.6 shows the schematic for a simulation system used to evaluate the various scheduling algorithms.

Some of the major issues with simulation are:
- Expensive: hours of programming and execution time are required
- Simulations may be erroneous because of the assumptions about distributions used for arrival and service rates may not reflect a real environment
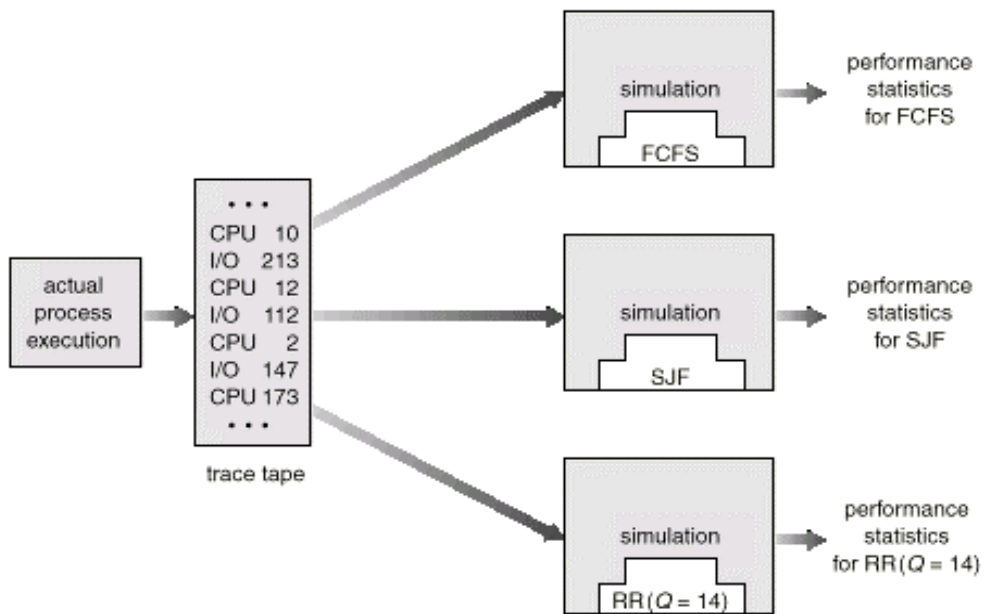
Figure 17.6 Simulation of Scheduling Algorithms

Implementation

Even a simulation is of limited accuracy. The only completely accurate way to evaluate a scheduling algorithm is to code it, put it in the operating system and see how it works. This approach puts the actual algorithm in the real system for evaluation under real operating conditions. The Open Source software licensing has made it possible for us to test various algorithms by implementing them in the Linux kernel and measuring their true performance.

The major difficulty is the cost of this approach. The expense is incurred in coding the algorithm and modifying the operating system to support it, as well as its required data structures. The other difficulty with any algorithm evaluation is that the environment in which the algorithm works will change.

Operating Systems
Lecture No. 18 and 19

Reading Material
- Chapter 7 of the textbook
- Lectures 18 and 19 on Virtual TV

Summary
- Process Synchronization: the basic concept
- The Critical Section Problem
- Solutions for the Critical Section Problem
- 2-Process Critical Section Problem solutions

Process Synchronization

Concurrent processes or threads often need access to shared data and shared resources. If there is no controlled access to shared data, it is often possible to obtain an inconsistent state of this data. Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes, and hence various process synchronization methods are used. In the producer-consumer problem that was discussed earlier, the version only allows one item less than the buffer size to be stored, to provide a solution for the buffer to use its entire capacity of N items is not simple. The producer and consumer share data structure 'buffer' and use other variables shown below:

```
#define BUFFER_SIZE 10
typedef struct
{
   ...
} item;
item buffer[BUFFER_SIZE];
int in=0;
int out=0;
```

The code for the producer process is:

```
while(1)
{
   /*Produce an item in nextProduced*/
   while(counter == BUFFER_SIZE); /*do nothing*/
   buffer[in]=nextProduced;
   in=(in+1)%BUFFER_SIZE;
   counter++;
}
```

The code for the consumer process is:

```
while(1)
{
    while(counter==0); //do nothing
    nextConsumed=buffer[out];
    out=(out+1)%BUFFER_SIZE;
    counter--;
    /*Consume the item in nextConsumed*/
}
```

Both producer and consumer routines may not execute properly if executed concurrently. Suppose that the value of the counter is 5, and that both the producer and the consumer execute the statement counter++ and counter- - concurrently. Following the execution of these statements the value of the counter may be 4,5,or 6! The only correct result of these statements should be counter= =5, which is generated if the consumer and the producer execute separately. Suppose counter++ is implemented in machine code as the following instructions:

```
MOV   R1, counter
INC   R1
MOV   counter, R1
```

whereas counter- - maybe implemented as:

```
MOV   R2, counter
DEC   R2
MOV   counter, R2
```

    If both the producer and consumer attempt to update the buffer concurrently, the machine language statements may get interleaved. Interleaving depends upon how the producer and consumer processes are scheduled. Assume counter is initially 5. One interleaving of statements is:

```
producer: MOV   R1, counter    (R1 = 5)
          INC   R1             (R1 = 6)
consumer: MOV   R2, counter    (R2 = 5)
          DEC   R2             (R2 = 4)
producer: MOV   counter, R1    (counter = 6)
consumer: MOV   counter, R2    (counter = 4)
```

The value of count will be 4, where the correct result should be 5. The value of count could also be 6 if producer executes MOV counter, R1 at the end. The reason for this state is that we allowed both processes to manipulate the variable counter concurrently. A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the manipulation depends on the particular order in which the access takes place, is called a **race condition**. To guard against such race conditions, we require synchronization of processes.

    Concurrent transactions in a bank or in an airline reservation (or travel agent) office are a couple of other examples that illustrates the critical section problem. We show

interleaving of two bank transactions, a deposit and a withdrawal. Here are the details of the transactions:

- Current balance = Rs. 50,000
- Check deposited = Rs. 10,000
- ATM withdrawn = Rs. 5,000

The codes for deposit and withdrawal are shown in Figure 18.1.
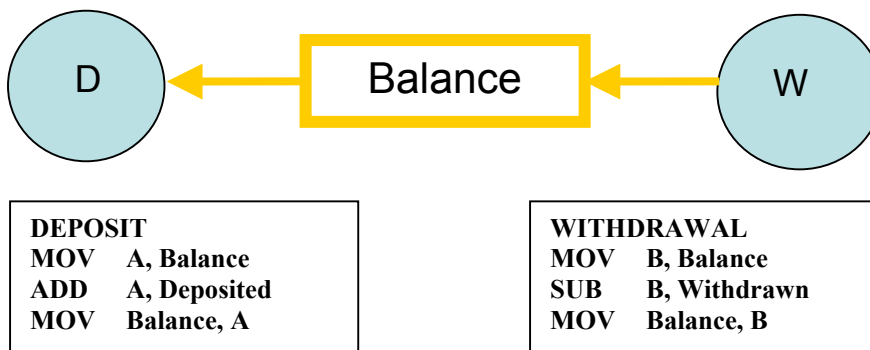
Figure 18.1 Bank transactions—deposit and withdrawal

Here is what may happen if the two transactions are allowed to execute concurrently, i.e., the transactions are allowed to interleave. Note that in this case the final balance will be Rs. 45,000, i.e., a loss of Rs. 5,000. If `MOV Balance,A` executes at the end, the result will be a gain of Rs. 5,000. In both cases, the final result is wrong.

```
Check Deposit:
        MOV   A, Balance        // A = 50,000
        ADD   A, Deposited      // A = 60,000
ATM Withdrawal:
        MOV   B, Balance        // B = 50,000
        SUB   B, Withdrawn      // B = 45,000
Check Deposit:
        MOV   Balance, A        // Balance = 60,000
ATM Withdrawal:
        MOV   Balance, B        // Balance = 45,000
```

The Critical Section Problem

Critical Section: A piece of code in a cooperating process in which the process may updates shared data (variable, file, database, etc.).

Critical Section Problem: Serialize executions of critical sections in cooperating processes.

When a process executes code that manipulates shared data (or resource), we say that the process is in its critical section (for that shared data). The execution of critical sections must be mutually exclusive: at any time, only one process is allowed to execute in its critical section (even with multiple processors). So each process must first request permission to enter its critical section. The section of code implementing this request is

called the **entry section**. The remaining code is the **remainder section**. The critical section problem is to design a protocol that the processes can use so that their action will not depend on the order in which their execution is interleaved (possibly on many processors).

There can be three kinds of solution to the critical section problem:

- Software based solutions
- Hardware based solutions
- Operating system based solution

We discuss the software solutions first. Regardless of the type of solution, the structure of the solution should be as follows. The Entry and Exist sections comprise solution for the problem.

```
do
{
        Entry section

            critical section

        Exit section

            remainder section

} while(1)
```
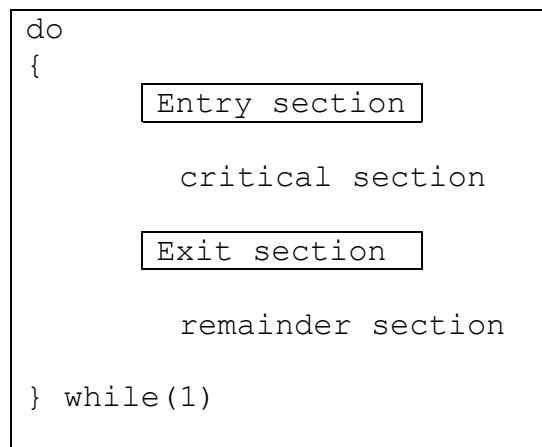
Solution to the Critical Section Problem

A solution to the critical section problem must satisfy the following three requirements:

1. **Mutual Exclusion**

   If process P_i is executing in its critical section, then no other process can be executing in their critical section.

2. **Progress**

   If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. **Bounded Waiting**

   There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Assumptions

While formulating a solution, we must keep the following assumptions in mind:

- Assume that each process executes at a nonzero speed
- No assumption can be made regarding the relative speeds of the N processes.

2-Process Solutions to the Critical Section Problem

In this section algorithms that are applicable to two processes will be discussed. The processes are P_0 and P_1. When presenting P_i, we use P_j to denote the other process. An assumption is that the basic machine language instructions such as load and store are executed atomically, that is an operation that completes in its entirety without interruption.

Algorithm 1

The first approach is to let the processes share a common integer variable **turn** initialized to 0 or 1. If turn == i, then process P_i is allowed to execute in its critical section. The structure of the process P_i is as follows:

```
do
{
    while(turn!=j);

       critical section

    turn=j;

       remainder section
} while(1)
```

This solution ensures mutual exclusion, that is only one process at a time can be in its critical section. However it does not satisfy the progress requirement, since it requires strict alternation of processes in the execution of the critical section. For example, if turn==0 and P_1 is ready to enter its critical section, P_1 cannot do so even though P_0 may be in its remainder section. The bounded wait condition is satisfied though, because there is an alternation between the turns of the two processes.

Algorithm 2

In algorithm two, the variable turn is replaced with an array `boolean flag[2]` whose elements are initialized to false. If flag is true for a process that indicates that the process is ready to enter its critical section. The structure of process P_i is shown:

```
do
{
        flag[i]=true;
        while(flag[j]);

                critical section

        flag[i]=false;

                remainder section
} while(1)
```

In this algorithm P_i sets `flag[i]= true` signaling that it is ready to enter its critical section. Then P_i checks to verify that process P_j is not also ready to enter its critical section. If P_j were ready, then P_i would wait until P_j had indicated that it no longer needed to be in the critical section (that is until `flag[j]=false`). At this point P_i would enter the critical section. On exiting the critical section, P_i would set `flag[i]=false` allowing the other process to enter its critical section. In this solution, the mutual exclusion requirement is satisfied. Unfortunately the progress condition is not met; consider the following execution sequence:

T_0: P_0 sets `flag[0]= true`
T_1: P_1 sets `flag[1]= true`

Now both the processes are looping forever in their respective while statements.

Operating Systems
Lecture No. 20

Reading Material
- Chapter 7 of the textbook
- Lecture 20 on Virtual TV

Summary
- 2-Process Critical Section Problem (continued)
- n-Process Critical Section Problem
- The Bakery Algorithm

2-Process Critical Section Problem (continued)
We discussed two solutions for the 2-process critical section problem in lecture 19 but both were not acceptable because they did not satisfy the progress condition. Here is a good solution for the critical section problem that satisfies all three requirements of a good solution.

Algorithm 3
The processes share two variables:

```
boolean flag[2];
int turn;
```

The boolean array of 'flag' is initialized to false, whereas 'turn' maybe 0 or 1. The structure of the process is as follows:

```
do
{
        flag[i]=true;
        turn=j;
        while(flag[j] && turn==j);

            critical section

        flag[i]=false;

            remainder section
} while(1)
```

To enter its critical section, P$_i$ sets flag[i] to true, and sets 'turn' to j, asserting that if the other process wishes to enter its critical section, it may do so. If both try to enter at the

same time, they will attempt to set 'turn' to i and j. However, only one of these assignments will last, the other will occur but be overwritten instantly. Hence, the eventual value of 'turn' will decide which process gets to enter its critical section.

To prove mutual exclusion, note that P_i enters its critical section only if either flag[j]=false or turn=i. Also, if both processes were executing in their critical sections at the same time, then flag[0]= = flag[1]= = true. These two observations suggest that P_0 and P_1 could not have found both conditions in the while statement true at the same time, since the value of 'turn' can either be 0 or 1. Hence only one process say P_0 must have successfully exited the while statement. Hence mutual exclusion is preserved.

To prove bounded wait and progress requirements, we note that a process P_i can be prevented the critical section only if it is stuck in the while loop with the condition flag[j]= =true and turn=j. If P_j is not ready to enter the critical section, then flag[j]=flase and P_i can enter its critical section. If P_j has set flag[j]=true and is also executing its while statement then either turn=i or turn=j. If turn=i then P_i enters its critical section, otherwise P_j. However, whenever a process finishes executing in its critical section, lets assume P_j, it resets flag[j] to false allowing P_i to enter its critical section. If P_j resets flag[j]=true, then it must also set 'turn' to i, and since P_i does not change the value of 'turn' while executing in its while statement, P_i will enter its critical section (progress) after at most one entry by P_j (bounded waiting).

N-Process Critical Section Problem

In this section we extend the critical section problem of two processes to include n processes. Consider a system of n processes (P_o, P_1 …… P_{n-1}). Each process has a segment of code called a critical section in which the process may be changing common variables, updating a table, writing a file and so on. The important feature of the system in that, when one process enters its critical section, no other process is allowed to execute in its critical section. Thus the execution of critical sections by the processes is mutually exclusive in time. The critical section problem is to design a protocol to serialize executions of critical sections. Each process must request permission to enter its critical section. Many solutions are available in the literature to solve the N-process critical section problem. We will discuss a simple and elegant solution, known as the Bakery algorithm.

The Bakery Algorithm

The bakery algorithm is due to Leslie Lamport and is based on a scheduling algorithm commonly used in bakeries, ice-cream stores, and other locations where order must be made out of chaos. On entering the store, each customer receives a number. The customer with the lowest number is served next. Before entering its critical section, process receives a ticket number. Holder of the smallest ticket number enters its critical section. Unfortunately, the bakery algorithm cannot guarantee that two processes (customers) will not receive the same number. In the case of a tie, the process with the lowest ID is served first. If processes Pi and Pj receive the same number, if i < j, then Pi is served first; else Pj is served first. The ticket numbering scheme always generates numbers in the increasing order of enumeration; i.e., 1, 2, 3, 4, 5 ...

Since process names are unique and totally ordered, our algorithm is completely deterministic. The common data structures are:

```
boolean choosing [n];
int number[n];
```

Initially these data structures are initialized to false and 0, respectively. The following notation is defined for convenience:

- (ticket #, process id #)
- (a,b) < (c,d) if a<c or if a= =c and b<d.
- $\max(a_{0,\ldots}a_{n-1})$ is a number, k, such that k>= a_i for i=0,…n-1

The structure of process P_i used in the bakery algorithm is as follows:

```
do
{
  choosing[i] = true;
  number[i] = max(number[0],number[1],..number[n-1])+1;
  choosing[i] = false;

  for(j=0; j<n; j++) {
    while(choosing[j]);
    while((number[j]!=0) && ((number[j],j) < (number[i],i)));
  }
                 Critical section

  number[i]=0;
                 Remainder section

} while(1);
```

To prove that the bakery algorithm is correct, we need to first show that if P_i is in its critical section and P_k has already chosen its number k!=0, then ((number [i],i) < (number[k],k)). Consider P_i in its critical section and P_k trying to enter its critical section. When process P_k executes the second while statement for j= = i it finds that,

- number[i] != 0
- (number[i],i) < (number[k],k)

Thus it keeps looping in the while statement until P_i leaves the P_i critical section. Hence mutual exclusion is preserved. For progress and bounded wait we observe that the processes enter their critical section on a first come first serve basis.

Following is an example of how the Bakery algorithm works. In the first table, we show that there are five processes, P0 through P4. P1's number is 0 because it is not interested in getting into its critical section at this time. All other processes are interested in entering their critical sections and have chosen non-zero numbers by using the `max()` function in their entry sections.

Process	Number
P0	3
P1	0
P2	7
P3	4
P4	8

The following table shows the status of all the processes as they execute the 'for' loops in their entry sections. The gray cells show processes waiting in the second while loops in their entry sections. The table shows that P0 never waits for any process and is, therefore, the first process to enter its critical section, while all other processes wait in their second while loops for j == 0, indicating that they are waiting for P0 to get out of its critical section and then they would make progress (i.e., they will get out the while loop, increment j by one, and continue their execution).

You can make the following observations by following the Bakery algorithm closely with the help of this table:

- P1 not interested to get into its critical section $\Rightarrow$ number[1] is 0
- P2, P3, and P4 wait for P0
- P0 gets into its CS, get out, and sets its number to 0
- P3 get into its CS and P2 and P4 wait for it to get out of its CS
- P2 gets into its CS and P4 waits for it to get out
- P4 gets into its CS
- Sequence of execution of processes: <P0, P3, P2, P4>

j	P0	P2	P3	P4
0	(3,0) < (3,0)	(3,0) < (7,2)	(3,0) < (4,3)	(3,0) < (8,4)
1	Number[1] = 0	Number[1] = 0	Number[1] = 0	Number[1] = 0
2	(7,2) < (3,0)	(7,2) < (7,2)	(7,2) < (4,3)	(7,2) < (8,4)
3	(4,3) < (3,0)	(4,3) < (7,2)	(4,3) < (4,3)	(4,3) < (8,4)
4	(8,4) < (3,0)	(8,4) < (7,2)	(8,4) < (4,3)	(8,4) < (8,4)

Operating Systems
Lecture No. 21

Reading Material
- Chapter 7 of the textbook
- Lecture 21 on Virtual TV

Summary
- Hardware solutions

Hardware Solutions for the Critical Section Problem

In this section, we discuss some simple hardware (CPU) instructions that can be used to provide synchronization between processes and are available on many systems.

The critical section problem can be solved simply in a uniprocessor environment if we could forbid interrupts to occur while a shared variable is being modified. In this manner, we could be sure that the current sequence of instructions would be run, so no unexpected modifications could be made to the shared variable.

Unfortunately this solution is not feasible in a multiprocessing environment, as disabling interrupts can be time consuming as the message is passed to all processors. This message passing delays entry into each critical section, and system efficiency decreases.

Normally, access to a memory location excludes other accesses to that same location. Designers have proposed machine instructions that perform two operations atomically (indivisibly) on the same memory location (e.g., reading and writing). The execution of such an instruction is also mutually exclusive (even on Multiprocessors). They can be used to provide mutual exclusion but other mechanisms are needed to satisfy the other two requirements of a good solution to the critical section problem.

We can use these special instructions to solve the critical section problem. These instructions are TestAndSet (also known as TestAndSetLock; TSL) and Swap. The semantics of the TestAndSet instruction are as follows:

```
boolean TestAndSet(Boolean &target)
{
  boolean rv=target;
  target=true;
  return rv;
}
```

The semantics simply say that the instruction saves the current value of 'target', set it to true, and returns the saved value.

The important characteristic is that this instruction is executed atomically. Thus if two TestAndSet instructions are executed simultaneously, they will be executed sequentially in some arbitrary order.

If the machine supports TestAndSet instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false. The structure of process P_i becomes:

```
do
{
        while (TestAndSet(lock)) ;
            Critical section
        lock=false;
            Remainder section
} while(1);
```

The above TSL-based solution is no good because even though mutual exclusion and progress are satisfied, bounded waiting is not.

The semantics of the Swap instruction, another atomic instruction, are, as expected, as follows:

```
boolean Swap(boolean &a, boolean &b)
{
  boolean temp=a;
  a=b;
  b=temp;
}
```

If the machine supports the Swap instruction, mutual exclusion can be implemented as follows. A global Boolean variable lock is declared and is initialized to false. In addition each process also has a local Boolean variable key. The structure of process P_i is:

```
do
{
        key=true;
        while(key == true)
            Swap(lock,key);
            Critical section
        lock=false;
            Remainder section
} while(1);
```

Just like the TSL-based solution shown in this section, the above Swap-based solution is not good because even though mutual exclusion and progress are satisfied, bounded waiting is not. In the next lecture, we will discuss a good solution for the critical section problem by using the hardware instructions.

Operating Systems
Lecture No. 22

Reading Material
- Chapter 7 of the textbook
- Lecture 22 on Virtual TV

Summary
- Hardware based solutions
- Semaphores
- Semaphore based solutions for the critical section problem

Hardware Solutions
In lecture 21 we started discussing the hardware solutions for the critical section problem. We discussed two possible solutions but realized that whereas both solutions satisfied the mutual exclusion and bounded waiting conditions, neither satisfied the progress condition. We now describe a solution that satisfies all three requirements of a solution to the critical section problem.

Algorithm 3
In this algorithm, we combine the ideas of the first two algorithms. The common data structures used by a cooperating process are:

```
boolean waiting[n];
boolean lock;
```

The structure of process P_i is:

```
do
{
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = TestAndSet(lock);
    waiting[i] = false;

        Critical section

    j = (i+1) % n;
    while ((j!=i) && !waiting[j])
        j = (j+1)% n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;

        Remainder section
} while(1);
```

These data structures are initialized to false. To prove that the mutual exclusion requirement is met, we note that process P_i can enter its critical section only if either waiting[i]= = false or key = = false. The value of key can become false only if TestAndSet is executed. The first process to execute the TestAndSet instruction will find key= =false; all others must wait. The variable waiting[i] can only become false if another process leaves its critical section; only one waiting[i] is set to false, maintaining the mutual exclusion requirement.

To prove the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets lock to false or sets waiting[j] to false. Both allow a process that is waiting to enter its critical section to proceed.

To prove that the bounded waiting requirement is met, we note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i+1, i+2, …, n-1, 0, 1, …, i-1). It designates the first process it sees that is in its entry section with waiting[j]=true as the next one to enter its critical section. Any process waiting to do so will enter its critical section within n-1 turns.

Semaphores

Hardware solutions to synchronization problems are not easy to generalize to more complex problems. To overcome this difficulty we can use a synchronization tool called a semaphore. A **semaphore** S is an integer variable that, apart from initialization is accessible only through two standard atomic operations: wait and signal. These operations were originally termed P (for wait) and V (for signal). The classical definitions of wait and signal are:

```
wait(S) {
   while(S<=0)
   ;// no op
   S--;
}
```

```
signal(S) {
   S++;
}
```

Modifications to the integer value of the semaphore in the wait and signal operations must be executed indivisibly. That is, when one process is updating the value of a semaphore, other processes cannot simultaneously modify that same semaphore value. In addition, in the case of the wait(S), the testing of the integer value of S (S<=0) and its possible modification (S--) must also be executed without interruption.

We can use semaphores to deal with the n-process critical section problem. The n processes share a semaphore, **mutex** (standing for mutual exclusion) initialized to 1. Each process P_i is organized as follows:

```
do
{
        wait(mutex);

            Critical section

        signal(mutex);

            Remainder section

} while(1);
```

As was the case with the hardware-based solutions, this is not a good solution because even though it satisfies mutual exclusion and progress, it does not satisfy bounded wait.

In a uni-processor environment, to ensure atomic execution, while executing wait and signal, interrupts can be disabled. In case of a multi-processor environment, to ensure atomic execution is one can lock the data bus, or use a soft solution such as the Bakery algorithm.

The main disadvantage of the semaphore discussed in the previous section is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process may be able to use productively. This type of semaphore is also called a **spinlock** (because the process spins while waiting for the lock). Spinlocks are useful in multiprocessor systems. The advantage of a spinlock is that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. This is, spinlocks are useful when they are expected to be held for short times. The definition of semaphore should be modified to eliminate busy waiting. We will discuss the modified definition of semaphore in the next lecture.

Operating Systems
Lecture No. 23

Reading Material
- Chapter 7 of the textbook
- Lecture 23 on Virtual TV

Summary
- Busy waiting
- New definition of semaphore
- Process synchronization
- Problems with the use of semaphore: deadlock, starvation, and violation of mutual exclusion

Semaphores

The main disadvantage of the semaphore discussed in the previous section is that they all require **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process may be able to use productively. This type of semaphore is also called a **spinlock** (because the process spins while waiting for the lock). Spinlocks are useful in multiprocessor systems. The advantage of a spinlock is that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. This, when locks are expected to be held for short times, spinlocks are useful.

To overcome the need for busy waiting, we can modify the definition of semaphore and the wait and signal operations on it. When a process executes the wait operation and finds that the semaphore value is not positive, it must wait. However, rather than busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then, control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal operation. The process is restarted by a wakeup operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU scheduling algorithm.)

Such an implementation of a semaphore is as follows:

```
typedef struct {
  int value;
  struct process *L;
} semaphore;
```

Each semaphore has an integer value and a list of processes. When a process must wait on a semaphore; it is added to the list of processes. A signal operation removes one process from the list of the waiting processes and awakens that process. The wait operation can be defined as:

```
void wait(semaphore S) {
  S.value--;
  if(S.value < 0) {
      add this process to S.L;
      block();
  }
}
```

The signal semaphore operation can be defined as

```
void signal wait(semaphore S) {
  S.value++;
  if(S.value <= 0) {
      remove a process P from S.L;
      wakeup(P);
  }
}
```

The block operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls. The negative value of S.value indicates the number of processes waiting for the semaphore. A pointer in the PCB needed to maintain a queue of processes waiting for a semaphore. As mentioned before, the busy-waiting version is better when critical sections are small and queue-waiting version is better for long critical sections (when waiting is for longer periods of time).

Process Synchronization

You can use semaphores to synchronize cooperating processes. Consider, for example, that you want to execute statement B in Pj only after statement A has been executed in Pi. You can solve this problem by using a semaphore S initialized to 0 and structuring the codes for Pi and Pj as follows:

Pi	**Pj**
...	...
A;	wait(S);
signal(S);	B;
...	...

Pj will not be able to execute statement B until Pi has executed its statements A and signal(S).

Here is another synchronization problem that can be solved easily using semaphores. We want to ensure that statement S1 in P1 executes only after statement S2 in P2 has

executed, and statement S2 in P2 should execute only after statement S3 in P3 has executed. One possible semaphore-based solution uses two semaphores, A and B. Here is the solution.

```
semaphore A=0, B=0;
P1              P2              P3

...             ...             ...

wait(A);        wait(B);        S3;

S1;             S2;             signal(B);

                signal(A);

...             ...             ...
```

Problems with Semaphores
Here are some key points about the use of semaphores:

- Semaphores provide a powerful tool for enforcing mutual exclusion and coordinating processes.
- The wait(S) and signal(S) operations are scattered among several processes. Hence, it is difficult to understand their effects.
- Usage of semaphores must be correct in all the processes.
- One bad (or malicious) process can fail the entire system of cooperating processes.

Incorrect use of semaphores can cause serious problems. We now discuss a few of these problems.

Deadlocks and Starvation
A set of processes are said to be in a deadlock state if every process is waiting for an event that can be caused only by another process in the set. Here are a couple of examples of deadlocks in our daily lives.

- Traffic deadlocks
- One-way bridge-crossing

Starvation is infinite blocking caused due to unavailability of resources. Here is an example of a deadlock.

```
        P0              P1

        wait(S);        wait(Q);
        wait(Q);        wait(S);
        ...             ...
        signal(S);      signal(Q);
        signal(Q);      signal(S);
        ...             ...
```

P0 and P1 need to get two semaphores, S and Q, before executing their critical sections. The following code structures can cause a deadlock involving P0 and P1. In this example, P0 grabs semaphore S and P1 obtains semaphore Q. Then, P0 waits for Q and P1 waits for S. P0 waits for P1 to execute signal(Q) and P1 waits for P0 to execute signal(S).

Neither process will execute the respective instruction—a typical deadlock situation. The following diagram shows the situation pictorially.
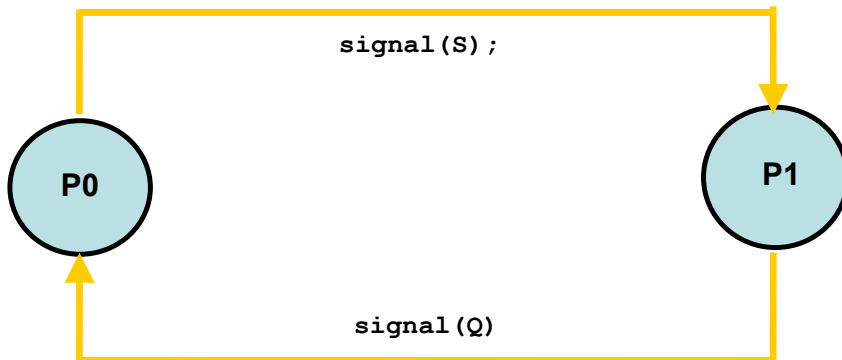
Here is an example of starvation. The code structures are self-explanatory.

```
P0                  P1

wait(S);            wait(S);
...                 ...
wait(S);            signal(S);
...                 ...
```

Violation of Mutual Exclusion
In the following example, the principle of mutual exclusion is violated. Again, the code structures are self-explanatory. If you have any questions about them, please see the lecture video.

```
P0                  P1

signal(S);          wait(S);
...                 ...
wait(S);            signal(S);
...                 ...
```

These problems are due to programming errors because of the tandem use of the wait and signal operations. The solution to these problems is higher-level language constructs such as critical region (region statement) and monitor. We discuss these constructs and their use to solve the critical section and synchronization problems in the next lecture.

Operating Systems
Lecture No. 24

Reading Material
- Chapter 7 of the textbook
- Lecture 24 on Virtual TV

Summary
- Counting semaphores
- Classical synchronization problems
- Bounded buffer problem
- Readers and writers problem
- Dining philosophers problem

Semaphores

There are two kinds of semaphores:

- **Counting semaphore** whose integer value can range over an unrestricted integer domain.
- **Binary semaphore** whose integer value cannot be > 1; can be simpler to implement.

Let S be a counting semaphore. To implement it in terms of binary semaphores we need the following data structures:

```
binary-semaphore S1, S2;
int C;
```

Initially S1=1, S2=0, and the value of integer C is set to the initial value of the counting semaphore S. The wait operation on the counting semaphore S can be implemented as follows:

```
wait(S1);
C--;
if(C < 0) {
    signal(S1);
    wait(S2);
}
signal(S1);
```

The signal operation on the counting semaphore S can be implemented as follows:

```
wait(S1);
C++;
if(C <= 0)
  signal(S2);
else
  signal(S1);
```

Classic Problems of Synchronization
The three classic problems of synchronization are:
- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining Philosophers Problem

Bounded Buffer Problem

The bounded-buffer problem, which was introduced in a previous lecture, is commonly used to illustrate the power of synchronization primitives. The solution presented in this section assumes that the pool consists of n buffers, each capable of holding one item.
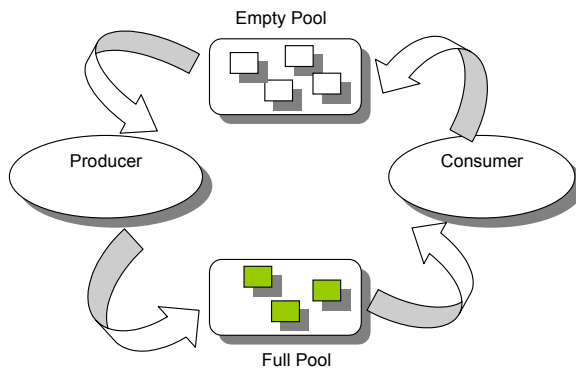
The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers, respectively. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

The code for the producer is as follows:

```
do {
   ...
   produce an item in nextp
   ...
   wait(empty);
   wait(mutex);
   ...
   add nextp to buffer
   ...
   signal(mutex);
   signal(full);
} while(1);
```

And that for the consumer is as follows:

```
do {
    wait(full);
    wait(mutex);
    ...
    remove an item from
    buffer to nextc
    ...
    signal(mutex);
    signal(empty);
    ...
    consume the item in nextc
    ...
} while(1);
```

Note the symmetry between the producer and the consumer process. This code can be interpreted as the producer producing full buffers for the consumer, or as the consumer producing empty buffers for the producer.

Readers Writers Problem

A data object (such as a file or a record) is to be shared among several concurrent processes. Some of these processes, called **readers**, may want only to read the content of the shared object whereas others, called **writers**, may want to update (that is to read and write) the shared object. Obviously, if two readers access the data simultaneously, no adverse effects will result. However, if a writer and some other process (whether a writer or some readers) access the shared object simultaneously, chaos may ensue.

To ensure these difficulties do not arise, we require that the writers have exclusive access to the shared object. This synchronization problem is referred to the readers-writers problem. Since it was originally stated, it has been used to test nearly every new synchronization primitive. The readers-writers problem has several variations, all involving priorities. The simplest one, referred to as the **first readers-writers problem**, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The **second readers-writers problem** requires that once a writer is ready, that writer performs its write as soon as

possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. In this section, we discuss a solution to the first readers-writers problem. In the solution to the first readers-writers problem, processes share the following data structures.

```
semaphore mutex, wrt;
int readcount;
```

The semaphores mutex and wrt are initialized to 1; readcount is initialized to 0. The semaphore wrt is common to both the reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the reader processes update the readcount variable. The readcount variable keeps track of how many processes are currently reading the object. The wrt semaphore is used to ensure mutual exclusion for writers or a writer and readers. This semaphore is also used by the first and last readers to block entry of a writer into its critical section and to allow open access to the wrt semaphore, respectively. It is not used by readers who enter or exit, while at least one reader is in its critical sections.

The codes for reader and writer processes are shown below:

```
wait(mutex);
   readcount++;
   if(readcount == 1)
      wait(wrt);
signal(mutex);
   ...
    reading is performed
   ...
wait(mutex);
   readcount--;
   if(readcount == 0)
      signal(wrt);
signal(mutex);
```

```
wait(wrt);
   ...
    writing is performed
   ...
signal(wrt);
```

Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on wrt, and n-1 readers are queued on mutex. Also observe that when a writer executes signal(wrt) we may resume the execution of either the waiting readers or a single waiting writer; the selection is made by the CPU scheduler.

Dining Philosophers Problem

Consider five philosophers who spend their lives thinking and eating, as shown in the following diagram.
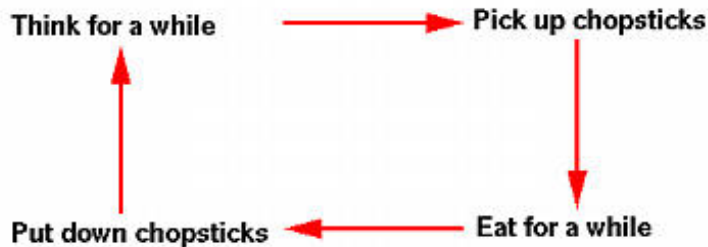
The philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.

When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick 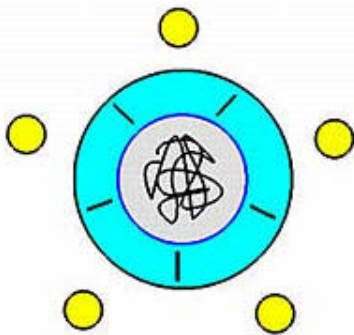that is already in the hand of her neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

The dining philosophers problem is considered to be a classic synchronization problem because it is an example of a large class of concurrency control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock and starvation free manner.

One simple solution is to represent each chopstick by a semaphore. A philosopher tires to grab the chopstick by executing a wait operation on that semaphore; she releases her chopsticks by executing the signal operation on the appropriate semaphores. Thus the shared data are:

```
semaphore chopstick[5];
```

All the chopsticks are initialized to 1. The structure of philosopher i is as follows:

```
do {
    wait(chopstick[i];
    wait(chopstick[(i+1)%5]);
    ...
    eat
    ...
    signal(chopstick[i]);
    signal(chopstick[(i+1)%5]);
    ...
    think
    ...
}
```

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it has the possibility of creating a deadlock.

Suppose that all five gets hungry at the same time and pick up their left chopsticks as shown in the following figure. In this case, all chopsticks are locked and none of the philosophers can successfully lock her right chopstick. As a result, we have a circular waiting (i.e., every philosopher waits for his right chopstick that is currently being locked by his right neighbor), and hence a deadlock occurs.
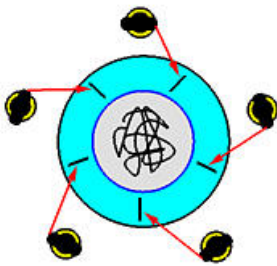
There are several possible good solutions of the problem. We will discuss these in the next lecture.

Operating Systems
Lecture No. 25

Reading Material
- Chapter 7 of the textbook
- Lecture 25 on Virtual TV

Summary
- Dining philosophers problem
- High-level synchronization constructs
- Critical region
- Monitor

Dining Philosophers Problem
Several possibilities that remedy the deadlock situation discussed in the last lecture are listed. Each results in a good solution for the problem.

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section)
- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Removing the possibility of deadlock does not ensure that starvation does not occur. Imagine that two philosophers are fast thinkers and fast eaters. They think fast and get hungry fast. Then, they sit down in opposite chairs as shown below. Because they are so fast, it is possible that they can lock their chopsticks and eat. After finish eating and before their neighbors can lock the chopsticks and eat, they come back again and lock the chopsticks and eat. In this case, the other three philosophers, even though they have been sitting for a long time, they have no chance to eat. This is a starvation. Note that it is not a deadlock because there is no circular waiting, and everyone has a chance to eat!
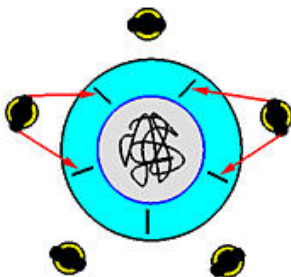
High-level Synchronization Constructs

We discussed the problems of deadlock, starvation, and violation of mutual exclusion caused by the poor use of semaphores in lecture 23. We now discuss some high-level synchronization constructs that help solve some of these problems.

Critical regions

Although semaphores provide a convenient and effective mechanism for process synchronization, their incorrect usage can still result in timing errors that are difficult to detect, since these errors occur only if some particular execution takes place, and these sequences do not always happen.

To illustrate how, let us review the solution to the critical section problem using semaphores. All processes share a semaphore variable mutex, which is initialized to 1. Each process must execute wait(mutex) before entering the critical section and signal(mutex) afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously.

To deal with the type of errors we outlined above and in lecture 23, a number of high-level constructs have been introduced. In this section we describe one fundamental high-level synchronization construct—the **critical region**. We assume that a process consists of some local data, and a sequential program that can operate on the data. Only the sequential program code that is encapsulated within the same process can access the local data. That is, one process cannot directly access the local data of another process. Processes can however share global data.

The critical region high-level synchronization construct requires that a variable v of type T, which is to be shared among many processes, be declared as:

```
v:shared T;
```

The variable v can be accessed only inside a region statement of the following form:

```
region v when B do S;
```

This construct means that, while statement S is being executed, no other process can access the variable v. The expression B is a Boolean expression that governs the access to the critical region. When a process tries to enter the critical-section region, the Boolean expression B is evaluated. If the expression is true, statement S is executed. If it is false, the process relinquishes the mutual exclusion and is delayed until B becomes true and no other process is in the region associated with v. Thus if the two statements,

```
region v when(true) S1;
region v when(true) S2;
```

are executed concurrently in distinct sequential processes, the result will be equivalent to the sequential execution "S1 followed by S2" or "S2 followed by S1".

The critical region construct can be effectively used to solve several certain general synchronization problems. We now show use of the critical region construct to solve the bounded buffer problem. Here is the declaration of buffer:

```
struct buffer {
    item pool[n];
    int count,in,out;
};
```

The producer process inserts a new item (stored in nextp) into the shared buffer by executing

```
region buffer when(count < n) {
    pool[in] = nextp;
    in = (in+1)%n;
    count++;
}
```

The consumer process removes an item from the shared buffer and puts it in nextc by executing

```
region buffer when(count > 0) {
    nextc = pool[out];
    out = (out+1)%n;
    count--;
}
```

Monitors

Another high-level synchronization construct is the monitor type. A **monitor** is characterized by local data and a set of programmer-defined operators that can be used to access this data; local data be accessed only through these operators. The representation of a monitor type consists of declarations of variables whose values define the state of an instance of the type, as well as the bodies of procedures or functions that implement operations on the type. Normal scoping rules apply to parameters of a function and to its local variables. The syntax of the monitor is as follows:

```
monitor monitor_name
{
    shared variable declarations

    procedure body P1(..) { ...}
    procedure body P1(..) { ...}
    ...
    procedure body P1(..) { ...}
    {
        initialization code
    }
}
```

The monitor construct ensures that only one process at a time can be active within the monitor. Consequently, the programmer does not need to code this synchronization

construct explicitly. While one process is active within a monitor, other processes trying to access a monitor wait outside the monitor. The following diagram shows the big picture of a monitor.
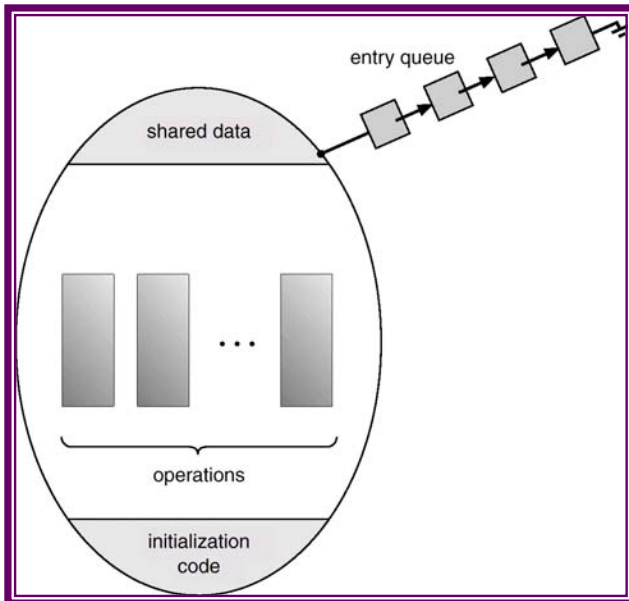
However, the monitor construct as defined so far is not powerful enough to model some synchronization schemes. For this purpose we need to define additional synchronization mechanisms. These mechanisms are provided by the **condition construct** (also called **condition variable**). A programmer who needs to write her own tailor made synchronization scheme can define one or more variables of type condition.

```
condition x,y;
```

The only operations that can be invoked on a condition variable are wait and signal. The operation

```
x.wait();
```

means that the process invoking this operation is suspended until another process invokes.

```
x.signal();
```

The x.signal() operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect; that is, the state of x is as though the operation were never executed. This is unlike the signal operation on a semaphore, where a signal operation always increments value of the semaphore by one. Monitors with condition variables can solve more synchronization problems that monitors alone. Still only one process can be active within a monitor but many processes may be waiting for a condition variable within a monitor, as shown in the following diagram.
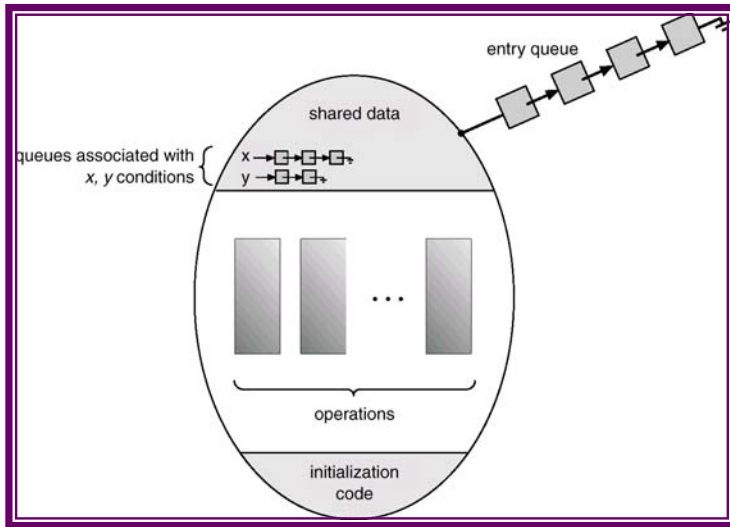
In the next lecture we will discuss a monitor-based solution for the dining philosophers problem.

Operating Systems
Lecture No. 26

Reading Material
- Chapters 7 and 8 of the textbook
- Lecture 26 on Virtual TV

Summary
- Monitor-based solution of the dining philosophers problem
- The deadlock problem
- Deadlock characterization
- Deadlock handling
- Deadlock prevention

Monitor-based Solution for the Dining Philosophers Problem

Let us illustrate these concepts by presenting a deadlock free solution to the dining philosophers problem. Recall that a philosopher is allowed to pick up her chopsticks only if both of them are available. To code this solution we need to distinguish among three states in which a philosopher may be. For this purpose we introduce the following data structure:

```
enum {thinking, hungry, eating} state[5];
```

Philosopher i can set the variable `state[i]=eating` only if her two neighbors are not eating: `(state[(i+4)%5]!=eating)` and `(state[(i+1)%5]!=eating)`.

We also need to declare five condition variables, one for each philosopher as follows. A philosopher uses her condition variable to delay herself when she is hungry, but is unable to obtain the chopsticks she needs.

```
condition self[5];
```

We are now in a position to describe our monitor-based solution to the dining-philosophers problem. The distribution of the chopsticks is controlled by the monitor dp; whose definition is as follows:

```
monitor dp
{
   enum {thinking,hungry,eating} state[5];
   condition self[5];

   void pickup(int i)
   {
      state[i]=hungry;
      test(i);
      if (state[i] != eating)
         self[i].wait();
   }
   void putdown(int i)
   {
      state[i]=thinking;
      test((i+4)%5);
      test((i+1)%5);
   }
   void test(int i)
   {
      if ((state[(i+4)%5]!=eating) &&
          (state[i]==hungry)&& state[(i+1)%5]!=eating)) {
         state[i]=eating;
         self[i].signal();
      }
   }
   void init()
   {
      for(int i=0;i<5;i++)
    state[i]=thinking;
   }
}
```

Each philosopher before starting to eat must invoke the pickup operation. This operation ensures that the philosopher gets to eat if none of its neighbors are eating. This may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the putdown operation and may start to think. The putdown operation checks if a neighbor (right or left—in this order) of the leaving philosopher wants to eat. If a neighboring philosopher is hungry and neither of that philosopher's neighbors is eating, then the leaving philosopher signals it so that she could eat. In order to use this solution, a philosopher i must invoke the operations pickup and putdown in the following sequence:

```
dp.pickup(i);
    ...
    eat
    ...
dp.putdown(i);
```

It is easy to show that this solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur. We note, however, that it is possible for a philosopher to starve to death. You should think about this problem and satisfy yourself.

The Deadlock Problem

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set. Here's an example:

- System has 2 tape drives.
- P1 and P2 each hold one tape drive and each needs another one.

Another deadlock situation can occur when the poor use of semaphores, as discussed in lecture 23. We reproduce that situation here. Assume that two processes, P0 and P1, need to access two semaphores, A and B, before executing their critical sections. Semaphores are initialized to 1 each. The following code snippets show how a situation can arise where P0 holds semaphore A, P1 holds semaphore B, and both wait for the other semaphore—a typical deadlock situation as shown in the figure that follows the code.

P0	**P1**
wait (A);	wait(B);
wait (B);	wait(A);
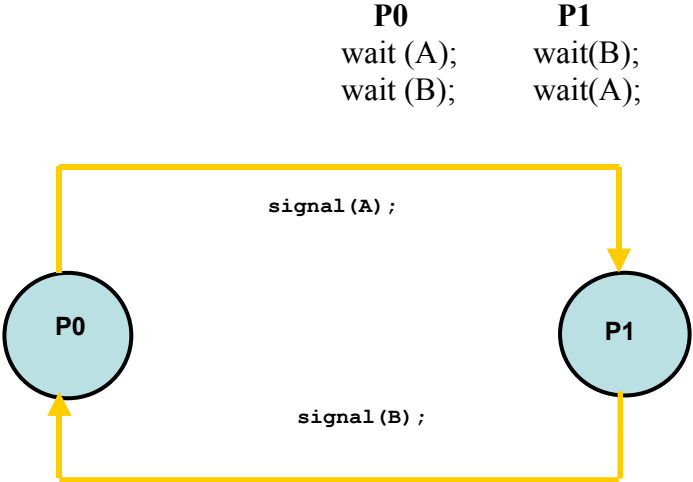
In the first solution for the dining philosophers problem, if all philosophers become hungry at the same time, they will pick up the chopsticks on their right and wait for getting the chopsticks on their left. This causes a deadlock.

Yet another example of a deadlock situation can occur on a one-way bridge, as shown below. Traffic flows only in one direction, and each section of a bridge can be viewed as a resource. If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback). Several cars may have to be backed up if a deadlock occurs. Starvation is possible.
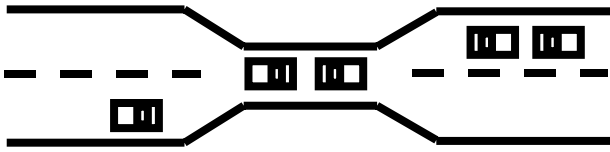
In the next three to four lectures, we will discuss the issue of deadlocks in computer systems in detail.

System Model

A system consists of a finite number of resources to be distributed among a finite number of cooperating processes. The resources are partitioned into several types, each of which consists of some number of identical instances. Memory space, CPU cycles, disk drive, file are examples of resource types. A system with two identical tape drives is said to have two instances of the resource type disk drive.

If a process requests an instance of a resource type, the allocation of any instance of that type will satisfy the request. If it will not, then the instances are not identical and the resource type classes have not been defined properly.

A process must request a resource before using it, and must release the resource after using it. A process may request as many resources as it requires in order to carryout its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. **Request:** The process requests a needed resource. If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can use the resource.
3. **Release:** The process releases the resource.
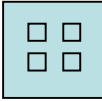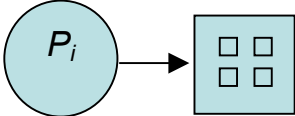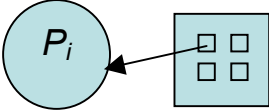
Deadlock Characterization

The following four conditions must hold simultaneously for a deadlock to occur:

1. **Mutual exclusion**: At least one resource must be held in a non-sharable mode; that is only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait**: A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption**: Resources cannot be preempted. That is, after using it a process releases a resource only voluntarily.
4. **Circular wait**: A set $\{P_0, P_1 \ldots P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, and so on, P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0.

Resource Allocation Graphs

Deadlocks can be described more precisely in terms of a directed graph called a system **resource allocation graph**. This graph consists of a set of vertices V and a set of edges E. The set of vertices is portioned into two different types of nodes $P=\{P_0, P_1 \ldots P_n\}$, the set of the active processes in the system, and $R=\{R_0, R_1 \ldots R_n\}$, the set consisting of all resource types in the system. A directed edge from a process P_i to resource type R_j signifies that process P_i requested an instance of R_j and is waiting for that resource. A directed edge from $R_{j\ to}\ P_i$ signifies that an instance of R_j has been allocated to P_i. We will use the following symbols in a resource allocation graph.

- Process

- Resource Type with 4 instances

- P_i requests instance of Rj

- P_i is holding an instance of Rj

The resource allocation graph shown below depicts the following situation:
- $P=\{P_1, P_2, P_3\}$
- $R=\{R_1, R_2, R_3\}$
- $E=\{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, P_3 \rightarrow R_3\}$

Resource Instances
- One instance of resource type R_1
- Two instances of resource type R_2
- One instance of resource type R_3
- Three instances of resource type R_4

Process States
- Process P_1 is holding an instance of resource R_2, and is waiting for an instance of resource R_1.
- Process P_2 is holding an instance of resource R_1 and R_2, and is waiting for an instance of resource R_3.
- Process P_3 is holding an instance of resource R_3.
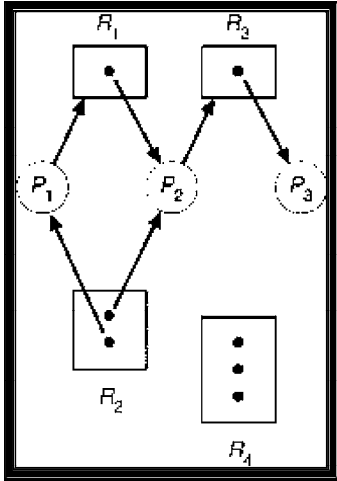
Given the definition of a resource allocation graph, it can be shown that if the graph contains no cycles, then no process is deadlocked. If the graph contains cycles then:

- If only one instance per resource type, then a deadlock exists.
- If several instances per resource type, possibility of deadlock exists.

Here is a resource allocation graph with a deadlock. There are two cycles in this graph:

$\{P_1 \rightarrow R_1, R_1 \rightarrow P_2, P_2 \rightarrow R_3, R_3 \rightarrow P_3, P_3 \rightarrow R_2, R_2 \rightarrow P_1\}$ and

$\{P_2 \rightarrow R_3, R_3 \rightarrow P_3, P_3 \rightarrow R_2, R_2 \rightarrow P_2\}$

No process will release an already acquired resource and the three processes will remain in the deadlock state.
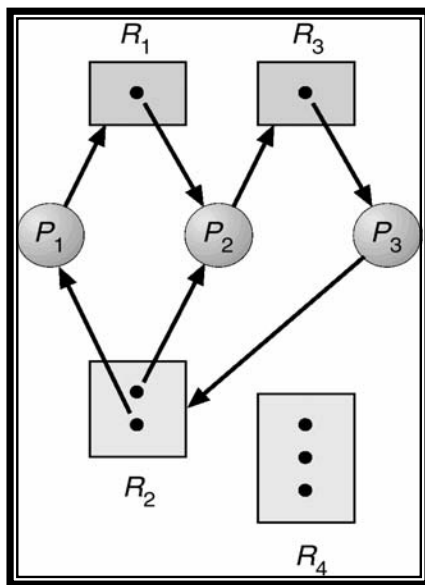
The graph shown below has a cycle but there is no deadlock because processes P2 and P4 do not require further resources to complete their execution and will release the resources they are currently hold in finite time. These resources can then be allocated to P1 and P3 for them to resume their execution.
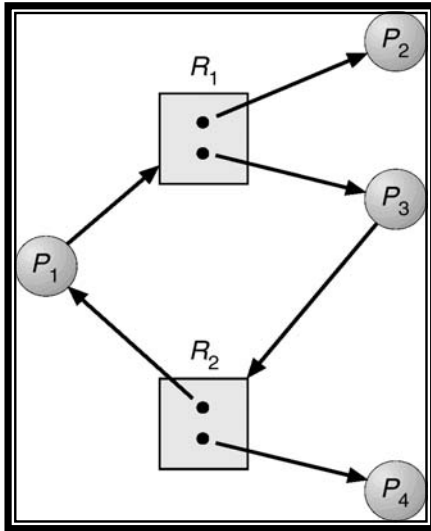
In the next lecture, we will characterize deadlocks. In other words, we will discuss the condition that must hold for a deadlock to occur. Following this we will discuss the various techniques to handle deadlocks.

Operating Systems
Lecture No. 27

Reading Material

- Chapter 8 of the textbook
- Lecture 27 on Virtual TV

Summary
- Deadlock handling
- Deadlock prevention
- Deadlock avoidance

Deadlock Handling

We can deal with deadlocks in a number of ways:
- Ensure that the system will never enter a deadlock state.
- Allow the system to enter a deadlock state and then recover from deadlock.
- Ignore the problem and pretend that deadlocks never occur in the system.

These three ways result in the following general methods of handling deadlocks:

1. **Deadlock prevention:** is a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how processes can request for resources.
2. **Deadlock Avoidance:** This method of handling deadlocks requires that processes give advance additional information concerning which resources they will request and use during their lifetimes. With this information, it may be decided whether a process should wait or not.
3. **Allowing Deadlocks and Recovering:** One method is to allow the system to enter a deadlocked state, detect it, and recover.

Deadlock Prevention

By ensuring that one of the four necessary conditions for a deadlock does not occur, we may prevent a deadlock.

Mutual exclusion

The mutual exclusion condition must hold for non-sharable resources, e.g., printer. Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock, e.g., read-only files. Also, resources whose states can be saved and restored can be shared, such as a CPU. In general, we cannot prevent deadlocks by denying the mutual exclusion condition, as some resources are intrinsically non-sharable.

Hold and Wait

To ensure that the hold and wait condition does not occur in a system, we must guarantee that whenever a process requests a resource, it does not hold any other resources. One protocol that can be used requires each process to request and be allocated all its

resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.

An alternative protocol requires a process to request resources only when it has none. A process may request some resources and use them. But it must release these before requesting more resources.

The two main disadvantages of these protocols are: firstly, resource utilization may be low, since many resources may be allocated but unused for a long time. Secondly, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

No preemption

To ensure that this condition does not hold we may use the protocol: if a process is holding some resources and requests another that cannot be allocated immediately to it, then all resources currently being held by the process are preempted. These resources are implicitly released, and added to the list of resources for which the process is waiting. The process will be restarted when it gets all its old, as well as the newly requested resources.

Circular Wait

One way to ensure that this condition never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing ordering of enumeration.

Let R={ R_1, R_2, R_3 }be resource types. We assign to each a unique integer, which allows us to compare two resources and to determine whether one precedes another in our ordering. For example, if the set of resource types R includes tape drivers, disk drives, and printers then the function F: R→N might be used to assign positive integers to these resources as follows:

F(tape drive) =1
F(disk drive) =5
F(printer)=12

Each process can request resources in an increasing order of enumeration. For example, a process wanting to use the tape and the disk drive must first request the tape drive and then the disk drive.

We can prove that if processes use this protocol then circular wait can never occur. We will prove this by contradiction. Let's assume that there is a cycle involving process P0 through Pk and that P_i is holding an instance of R_i, as shown below. The proof follows.

```
P0 → P1 → P2 → … → Pk → P0
R0    R1    R2          Rk    R0
```

$\Rightarrow$ F(R0) < F(R1) < … < F(Rk) < F(R0)

$\Rightarrow$ F(R0) < F(R0), which is impossible

$\Rightarrow$ There can be no circular wait.

Deadlock Avoidance

One method for avoiding deadlocks is to require additional information about how resources may be requested. Each request for resources by a process requires that the system consider the resources currently available, the resources currently allocated to the process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must wait to avoid a possible future deadlock. The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need. Given a priori information about the maximum number of resources of each type that may be requested by each process, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. A deadlock avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist.

Safe State

A state is safe if the system can allocate resources to each process in some order and still avoid a deadlock. More formally a system is in a safe state only if there exists a safe sequence. A sequence of processes $<P_1, P_2 \ldots P_n>$ is a safe sequence for the current allocation state if, for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus all the resources held by all the P_j with $j < i$. In this situation, if the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished. When they have finished, P_i can obtain all of its needed resources, complete its designated task, return its allocated resources and terminate. When P_i terminates, P_{i+1} can obtain its needed resources and terminate. If no such sequence exists, then the system is said to be unsafe.

If a system is in a safe state, there can be no deadlocks. An unsafe state is not a deadlocked state; a deadlocked state is conversely an unsafe state. Not all unsafe states are deadlocks, however an unsafe state may lead to a deadlock state. Deadlock avoidance makes sure that a system never enters an unsafe state. The following diagram shows the relationship between safe, unsafe, and deadlock states.
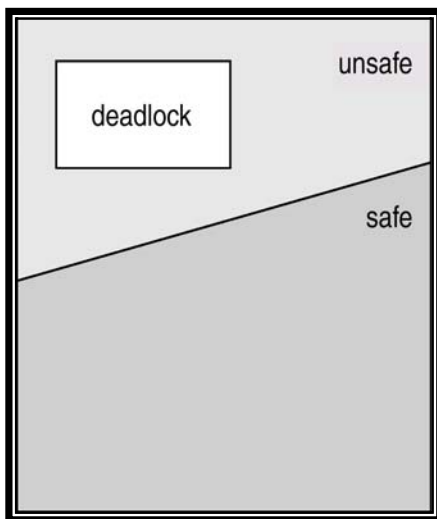
Let' consider the following example to explain how a deadlock avoidance algorithm works. There is a system with 12 tape drives and three processes. The current system state is as shown in the following table. The available column shows that initially there are three tapes drives available and when process P1 finishes, the two rape drives allocated to it are returned, making the total number of tape drives 5. With 5 available tape drives, the maximum remaining future needs of P0 (of 5 tape drives) can be met. Once this happens, the 5 tape drives that P0 currently holds will go back to the available pool of drives, making the grand total of available tape drives 10. With 10 available drives, the maximum future need of P2 of 7 drives can be met. Therefore, system is currently in a safe state, with the safe sequence <P1, P0, P2>.

Process	Max Need	Allocated	Available
P0	10	5	3
P1	4	2	5
P2	9	2	10

Now, consider that P2 requests and is allocated one more tape drive. Assuming that the tape drive is allocated to P2, the new system state will be:

Process	Max Need	Allocated	Available
P0	10	5	2
P1	4	2	4
P2	9	3	

This new system is not safe. With two tape drives available, P1's maximum remaining future need can be satisfied which would increase the number of available tapes to 4. With 4 tapes available, neither P0's nor P2's maximum future needs can be satisfied. This means that if P2 request for an additional tape drive is satisfied, it would the system in an unsafe state. Thus, P2's request should be denied at this time.

Resource Allocation Graph Algorithm
In addition to the request and assignment edges explained in the previous lectures, we introduce a new type of edge called a **claim edge** to resource allocation graphs. A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in the future. A dashed line is used to represent a claim edge. When P_i requests resource R_j the claim edge is converted to a request edge.

Suppose that P_i requests resource R_j. The request can be granted only if converting the request edge $P_i \rightarrow R_j$ into an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle. If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state.

Operating Systems
Lecture No. 28

Reading Material
- Chapter 8 of the textbook
- Lecture 28 on Virtual TV

Summary
- Deadlock avoidance
- Banker's algorithms
- Safety algorithm
- Safe Sequence

Deadlock Avoidance

Resource Allocation Graph Algorithm
In addition to the request and assignment edges explained in the previous lectures, we introduce a new type of edge called a **claim edge** to resource allocation graphs. A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in the future. A dashed line is used to represent a claim edge. When P_i requests resource R_j the claim edge is converted to a request edge. In the following resource allocation graph, the edge $P_2 \rightarrow R_2$ is a claim edge.
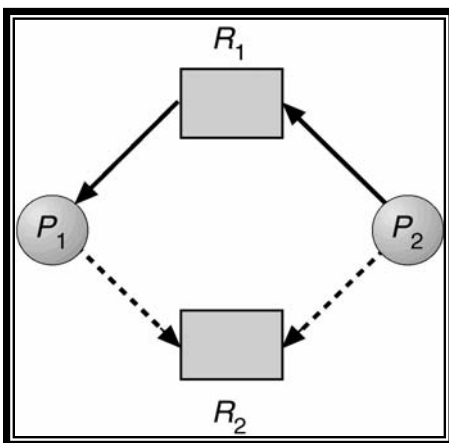
Suppose that P_i requests resource R_j. The request can be granted only if converting the request edge $P_i \rightarrow R_j$ into an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle. If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. The following resource allocation graph shows that the system is in an unsafe state:
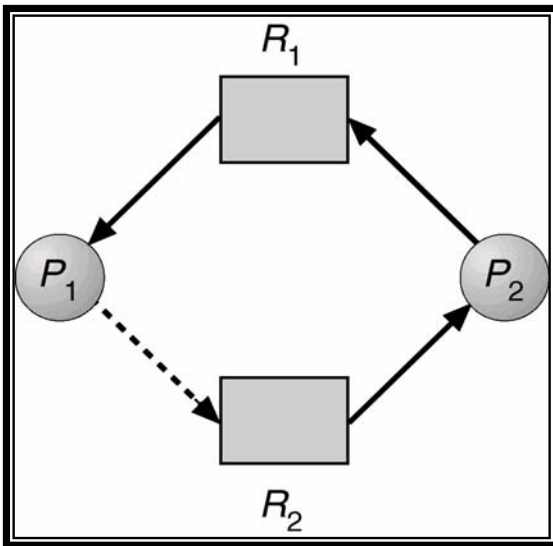
Banker's Algorithm

In this algorithm, when a new process enters the system, it must declare the maximum number of instances of each resource type that it may need, i.e., each process must a priori claim maximum use of various system resources. This number may not exceed the total number of instances of resources in the system, and there can be multiple instances of resources. When a process requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise the process must wait until some other process releases enough resources. We say that a system is in a safe state if all of the processes in the system can be executed to termination in some order; the order of process termination is called safe sequence. When a process gets all its resources, it must use them and return them in a finite amount of time.

Let n be the number of processes in the system and m be the number of resource types. We need the following data structures in the Banker's algorithm:

- **Available:** A vector of length m indicates the number of available instances of resources of each type. Available[j] == k means that there are k available instances of resource R_j.
- **Max:** An n x m matrix defines the maximum demand of resources of each process. Max[i,j] == k means that process P_i may request at most k instances of resource R_j.
- **Allocation:** An n x m matrix defines the number of instances of resources of each type currently allocated to each process. Allocation[i,j] == k means that P_i is currently allocated k instances of resource type R_j.
- **Need:** An n x m matrix indicates the remaining resource need of each process. Need[i,j] == k means that P_i may need k more instances of resource type R_j to complete its task. Note that Need[i,j] == Max[i,j] - Allocation[i,j].

Safety Algorithm
The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work = Available and Finish[i] = false fori = 1, 2, …, n.
2. Find an i such that both
   a) Finish[i] = = false
   b) $Need_i$ <= Work
   If no such i exists go to step 4.
3. Work = Work + $Allocation_i$
   Finish[i] = true
   Go to step 2
4. If Finish[i] = = true for all i, then the system is in a safe mode.

This algorithm may require an order of m x n^2 operations to decide whether a state is safe.

Resource Request Algorithm
Let $Request_i$ be the request vector for process P_i. if $Request_i$ [j]=k, then process P_i wants k instances of resource R_j. When a request for resources is made by process P_i the following actions are taken:

1. If $Request_i$ <= $Need_i$ go to step 2. Otherwise, raise an error condition since the process has exceeded its maximum claim.
2. If $Request_i$ <= Available, go to step 3. Otherwise P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:
   Availabe = Available-$Request_i$;
   $Allocation_i$ = $Allocation_i$ + $Request_i$;
   $Need_i$ = $Need_i$ –$Request_i$;

Invoke the Safety algorithm. If the resulting resource allocation graph is safe, the transaction is completed. Else, the old resource allocation state is restored and process P_i must wait for $Request_i$.

An illustrative example
We now show a few examples to illustrate how Banker's algorithm works. Consider a system with five processes P_0 through P_4 and three resource types: A, B, C. Resource type A has 10 instances, resource type B has 5 instances and resource type C has 7 instances. Suppose that at a time T_0 the following snapshot of the system has been taken:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			

	A	B	C			
P_4	0	0	2	4	3	3

The content of the matrix Need is defined to be Max- Allocation and is:

	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

In the following sequence of snapshots, we show execution of the Safety algorithm for the given system state to determine if the system is in a safe state. We progressively construct a safe sequence.

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	3	3	2
P_1	2	0	0	1	2	2	5	3	2
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

- Safe Sequence: $< P_1>$

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	1	2	2	5	3	2
P_2	3	0	2	6	0	0	7	4	3
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

- Safe Sequence: $< P_1, P_3>$

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	1	2	2	5	3	2
P_2	3	0	2	6	0	0	7	4	3
P_3	2	1	1	0	1	1	7	4	5
P_4	0	0	2	4	3	1			

- Safe Sequence: $< P_1, P_3, P_4>$

	Allocation			Need			Available		
	A	B	C	7	5	3	A	B	C
P_0	0	1	0	1	2	2	3	3	2

P₁	2	0	0	6	0	0	5	3	2
P₂	3	0	2	0	1	1	7	4	3
P₃	2	1	1	4	3	1	7	4	5
P₄	0	0	2	7	5	3	7	5	5

- Safe Sequence: $< P_1, P_3, P_4, P_0>$

The Safety algorithm concludes that the system is in a safe state, with $< P_0, P_1, P_2, P_3, P_4>$ being a safe sequence.

Suppose now that process P_1 requests one additional instance of resource type A and two instances of resource type C so Request $_1$ = (1, 0, 2). To decide whether this request can be fulfilled immediately, we invoke Banker's algorithm, which first check that Request$_1$ <= Available, which is true because (1,0,2)<=(3,3,2). It then pretends that this request has been fulfilled, and arrives at the following state:

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P₀	0	1	0	7	4	3	2	3	0
P₁	3	0	2	0	2	0			
P₂	3	0	2	6	0	0			
P₃	2	1	1	0	1	1			
P₄	0	0	2	4	3	1			

Banker's algorithm then executes the Safety algorithm to determine if the resultant system will be in a safe state. Here is the complete working of Banker's algorithm. If P_1 requests (1,0,2), lets evaluate if this request may be granted immediately. Banker's algorithm takes the following steps.

1. Is Request1 ≤ Need1?
   (1,0,2) ≤ (1,2,2) ⇒ true
2. Is Request1 ≤ Available?
   (1,0,2) ≤ (3,3,2) ⇒ true

It then pretends that request is granted and updates the various data structures accordingly. It then invokes the Safety algorithm to determine if the resultant state is safe. Here is sequence of steps taken by the Safety algorithm. The algorithm progressively constructs a safe sequence.

	Need			Allocation			Available		
	A	B	C	A	B	C	A	B	C
P₀	7	4	3	0	1	0	3	3	2
P₁	0	2	0	3	0	2			
P₂	6	0	0	3	0	2			
P₃	0	1	1	2	1	1			
P₄	4	3	1	0	0	2			

	Need			Allocation			Work		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	2	3	0
P_1	0	2	0	3	0	2			
P_2	6	0	0	3	0	2			
P_3	0	1	1	2	1	1			
P_4	4	3	1	0	0	2			

	Need			Allocation			Work		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	2	3	0
P_1	0	2	0	3	0	2	5	3	2
P_2	6	0	0	3	0	2			
P_3	0	1	1	2	1	1			
P_4	4	3	1	0	0	2			

- Safe Sequence: < P_1 >

	Need			Allocation			Work		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	2	3	0
P_1	0	2	0	3	0	2	5	3	2
P_2	6	0	0	3	0	2	7	4	3
P_3	0	1	1	2	1	1			
P_4	4	3	1	0	0	2			

- Safe Sequence: < P_1, P_3 >

	Need			Allocation			Work		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	2	3	0
P_1	0	2	0	3	0	2	5	3	2
P_2	6	0	0	3	0	2	7	4	3
P_3	0	1	1	2	1	1	7	4	5
P_4	4	3	1	0	0	2			

- Safe Sequence: < P_1, P_3 , P_4>

	Need			Allocation			Work		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	2	3	0
P_1	0	2	0	3	0	2	5	3	2
P_2	6	0	0	3	0	2	7	4	3
P_3	0	1	1	2	1	1	7	4	5
P_4	4	3	1	0	0	2	7	4	5

- Safe Sequence: < P_1, P_3 , P_4, P_0>

Hence executing Safety algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies the safety requirement and so P_1's request may be granted immediately. Note that safe sequence is not necessarily a unique sequence. There are several safe sequences for the above example. See lecture slides for more details.

Here is another example. P_0 requests (0,2,0). Should this request be granted? In order to answer this question, we again follow Banker's algorithm as shown in the following sequence of steps.

1. Is Request0 ≤ Need0?
   $(0,2,0) \le (7,4,3) \Rightarrow$ true
2. Is Request1 ≤ Available?
   $(0,2,0) \le (3,3,2) \Rightarrow$ true

	Need			Allocation			Available		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	3	3	2
P_1	1	2	2	2	0	0			
P_2	6	0	0	3	0	2			
P_3	0	1	1	2	1	1			
P_4	4	3	1	0	0	2			

The following is the updated system state. We run the Safety algorithm on this state now and show the steps of executing the algorithm.

	Need			Allocation			Work		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	3	1	2
P_1	1	2	2	2	0	0			
P_2	6	0	0	3	0	2			
P_3	0	1	1	2	1	1			
P_4	4	3	1	0	0	2			

	Need			Allocation			Work		
	A	B	C	A	B	C	A	B	C
P_0	7	2	3	0	3	0	3	1	2
P_1	1	2	2	2	0	0	5	2	3
P_2	6	0	0	3	0	2			
P_3	0	1	1	2	1	1			
P_4	4	3	1	0	0	2			

- Safe Sequence: <P3 >

	Need			Allocation			Work		
	A	B	C	A	B	C	A	B	C
P_0	7	2	3	0	3	0	3	1	2
P_1	1	2	2	2	0	0	5	2	3
P_2	6	0	0	3	0	2	7	2	3
P_3	0	1	1	2	1	1			

	4	3	1	0	0	2			
P_4	4	3	1	0	0	2			

- Safe Sequence: $<P_3, P_1>$

	Need			Allocation			Work		
	A	B	C	A	B	C	A	B	C
P_0	7	2	3	0	3	0	3	1	2
P_1	1	2	2	2	0	0	5	2	3
P_2	6	0	0	3	0	2	7	2	3
P_3	0	1	1	2	1	1	10	2	5
P_4	4	3	1	0	0	2			

- Safe Sequence: $<P_3, P_1, P_2>$

	Need			Allocation			Work		
	A	B	C	A	B	C	A	B	C
P_0	7	2	3	0	3	0	3	1	2
P_1	1	2	2	2	0	0	5	2	3
P_2	6	0	0	3	0	2	7	2	3
P_3	0	1	1	2	1	1	10	2	5
P_4	4	3	1	0	0	2	10	5	5

- Safe Sequence: $<P_3, P_1, P_2, P_0, P_4>$

Hence executing the safety algorithm shows that sequence $<P_3, P_1, P_2, P_0, P_4>$ satisfies safety requirement. And so P_0's request may be granted immediately.

Suppose P_0 requests (0,2,0). Can this request be granted after granting P_1's request of (1,0,2)?

Operating Systems
Lecture No. 29

Reading Material
- Chapter 8 of the textbook
- Lecture 29 on Virtual TV

Summary
- Deadlock detection: resources with single and multiple instances
- Recovery from deadlocks
- Process termination
- Resource preemption

Deadlock Detection
If a system does not employ either a deadlock prevention or a deadlock avoidance algorithm then a deadlock may occur. In this environment, the system must provide:
- An algorithm that examines (perhaps periodically or after certain events) the state of the system to determine whether a deadlock has occurred
- A scheme to recover from deadlocks

Single Instance of Each Resource Type
If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource allocation graph, called a **wait-for graph**. We obtain this graph from the resource allocation graph by removing the nodes of type resource and collapsing the appropriate edges. More precisely, an edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait-for graph exists if and only if the corresponding resource allocation graph contains two edges for $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ some resource R_q. As before, a deadlock exists in the system if and only if the wait for graph contains a cycle. To detect deadlocks the system needs to maintain the wait-for graph and periodically to invoke an algorithm that searches for a cycle in the graph. The following diagram shows a resource allocation graph and the corresponding wait-for graph. The system represented by the given wait-for graph has a deadlock because the graph contains a cycle.
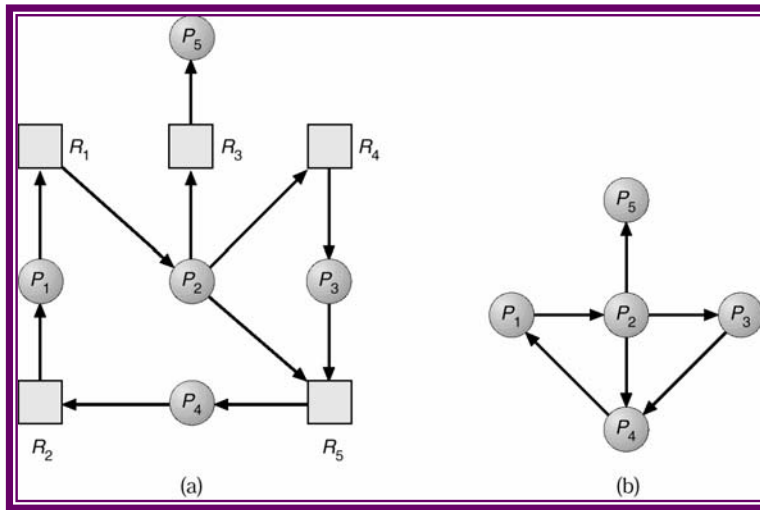
Resource-Allocation Graph   Corresponding wait-for graph

Several Instances of a Resource Type

The wait for graph scheme is not applicable to a resource allocation system with multiple instances of each resource type. The deadlock detection algorithm described next is applicable to such a system. It uses the following data structures:

- **Available:** A vector of length m indicates the number of available resources of each type.
- **Allocation:** An n x m matrix defines the number of resources of each type currently allocated to each process.
- **Request:** An n x m matrix indicates the current request of each process. If Request[i,j] == k, then process P_i is requesting k more instances of resource type R_j.

The algorithm is:
1) Let Work and Finish be vectors of length m and n respectively. Initialize Work=Available. For i=1, 2,… , n if Allocation[i] ≠ 0 the Finish[i]=false; otherwise Finish[i]=true
2) Find an index i such that both
   a. Finish[i] == false
   b. Request$_i$ ≤ Work
   c. If no such i exists go to step 4.
3) Work=Work + Allocation$_i$
   a. Finish[i]=true
   b. Go to step 2.
4) If Finish[i] == false, for some i, 1≤ i ≤n, then the system is in a deadlock state. Moreover, if Finish[i] == false, then P_i is deadlocked.

We show the working of this algorithm with an example. Consider the following system:

P = { P0, P1, P2, P3, P4 }
R = { A, B, C }
A: 7 instances
B: 2 instances
C: 6 instances

The system is currently in the following state. We want to know if the system has a deadlock. We find this out by running the above algorithm with the following state and construct a sequence in which requests for the processes may be granted.

	Allocation			Request			Work		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	2	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

	Allocation			Request			Work		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2	0	1	0
P2	3	0	2	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

- Finish Sequence: < P0>

	Allocation			Request			Work		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2	0	1	0
P2	3	0	2	0	0	0	3	1	2
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

- Finish Sequence: < P0, P2>

	Allocation			Request			Work		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2	0	1	0
P2	3	0	2	0	0	0	3	1	2
P3	2	1	1	1	0	0	5	2	3
P4	0	0	2	0	0	2			

- Finish Sequence: < P0, P2, P3>

146

	Allocation			Request			Work		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2	0	1	0
P_2	3	0	2	0	0	0	3	1	2
P_3	2	1	1	1	0	0	5	2	3
P_4	0	0	2	0	0	2	5	2	5

Here is the sequence in which requests of processes P_0 through P_4 may be satisfied:
$< P_0, P_2, P_3, P_4, P_1>$. This is not a unique sequence. A few other possible sequences are the following.

$< P_0, P_2, P_3, P_1, P_4,>$
$< P_0, P_2, P_4, P_1, P_3>$
$< P_0, P_2, P_4, P_3, P_1>$

Now let us assume that P_2 requests an additional instance of C. Do we have a finish sequence? The work below shows that if this request is granted, the system will enter a deadlock. P_0's request can be satisfied with currently available resources, but request for no other process can be satisfied after that. Thus, a deadlock exists, consisting of processes $P_1, P_2, P_3,$ and $P_4.$

Process	Request		
	A	B	C
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

	Allocation			Request			Work		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	2	0	0	1			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

	Allocation			Request			Work		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2	0	1	0
P_2	3	0	2	0	0	1			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

Detection Algorithm Usage

When should we invoke the deadlock detection algorithm? The answer depends n two factors:

1. How often is a deadlock likely to occur?
2. How many processes will be affected by deadlock when it happens?

Hence the options are:

- Every time a request for allocation cannot be granted immediately—expensive but process causing the deadlock is identified, along with processes involved in deadlock
- Periodically, or based on CPU utilization
- Arbitrarily—there may be many cycles in the resource graph and we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

Recovery from Deadlock

When a deadlock detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods the system reclaims all resources allocated to the terminated process.

- Abort all deadlocked processes: This method clearly will break the deadlock cycle, but at a great expense; these processes may have computed for a long time, and the results of these partial computations must be discarded and probably recomputed later.
- Abort one process at a time until the deadlock cycle is eliminated: This method incurs considerable overhead since after each process is aborted, a deadlock detection algorithm must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be so easy. If a process was in the midst of updating a file, terminating it will leave the system in an inconsistent state. If the partial termination method is used, then given a set of deadlocked processes, we must determine which process should be terminated in an attempt to break the deadlock. This determination is a policy decision similar to CPU scheduling problems. The question is basically an economic one, we should abort those processes the termination of which will incur the minimum cost.

Unfortunately, the term minimum cost is not a precise one. Many factors determine which process is chosen, including:

1. What the priority of the process is
2. How long the process has computed, and how much longer the process will compute before completing its designated task.
3. How many and what type of resources the process has used

4. How many resources the process needs in order to complete
5. How many processes will need to be terminated
6. Whether the process is interactive or batch

Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. **Selecting a victim:** Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlock process is holding, and the amount of time a deadlocked process has thus far consumed during its execution.

2. **Rollback:** If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state. Since, in general it is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it. However it is more effective to roll back the process only as far as necessary to break the deadlock. On the other hand, this method requires the system to keep more information about the state of all the running processes.

3. **Starvation:** In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as the victim. As a result this process never completes its designated task, a starvation situation that needs to be dealt with in any practical system. Clearly, we must ensure that a process is picked as a victim only a finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

Operating Systems
Lecture No. 30

Reading Material

- Chapter 9 of the textbook
- Lecture 30 on Virtual TV

Summary

- Basic concepts
- Logical to physical address translation
- Various techniques for memory management

Basic Concepts

Selection of memory-management method for a specific system depends on many factors especially on the *hardware* design of the system. Recent designs have integrated the hardware and operating system.

Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of its program counter and other memory management registers such as segment registers in Intel CPUs. These instructions may cause additional loading from and storing to specific memory addresses.

A typical instruction-execution cycle, e.g., first fetches an instruction from memory, which is then decoded and executed. Operands may have to be fetched from memory. After the instruction has been executed, the results are stored back in memory. The memory unit sees only a stream of memory addresses; it does not know how they are generated or what they are for (instructions or data).

Memory Hierarchy

The memory hierarchy includes:

- Very small, extremely fast, extremely expensive, and volatile CPU registers
- Small, very fast, expensive, and volatile cache
- Hundreds of megabytes of medium-speed, medium-price, volatile main memory
- Hundreds of gigabytes of slow, cheap, and non-volatile secondary storage
- Hundreds and thousands of terabytes of very slow, almost free, and non-volatile Internet storage (Web pages, Ftp repositories, etc.)

Memory Management

The purpose of memory management is to ensure fair, secure, orderly, and efficient use of memory. The task of memory management includes keeping track of used and free memory space, as well as when, where, and how much memory to allocate and deallocate. It is also responsible for swapping processes in and out of main memory

Source to Execution

Translation of a source program in a high-level or assembly language involves compilation and linking of the program. This process generates the machine language executable code (also known as a binary image) for the give source program. To execute the binary code, it is loaded into the main memory and the CPU state is set appropriately. The whole process is shown in the following diagram.

Compile/Assemble
↓
Link
↓
Load
↓
Execute
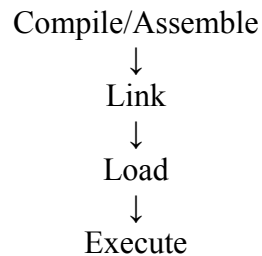
Address Binding

Usually a program resides on a disk as a binary executable or script file. The program must be brought into the memory it to be executed. The collection of processes that is waiting on the disk to be brought into the memory for execution forms the **input queue.**

The normal procedure is to select one of the processes in the input queue and to load that process into the memory. As the process is executed, it accesses instructions and data from memory. Eventually the process terminates and its memory space is become available for reuse.

In most cases, a user program will go through several steps–some of which may be optional–before being executed. These steps are shown in the following diagram. Addresses may be bound in different ways during these steps. Addresses in the source program are generally symbolic (such as an integer variable *count*). Address can be bound to instructions and data at different times, as discussed below briefly.

- **Compile time:** if you know at compile where the process will reside in memory, the **absolute addresses** can be assigned to instructions and data by the compiler.
- **Load time:** if it is not known at compile time where the process will reside in memory, then the compiler must generate **re-locatable code**. In this case the final binding is delayed until load time.
- **Execution time:** if the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this to work.

In case of compile and load time binding, a program may not be moved around in memory at run time.
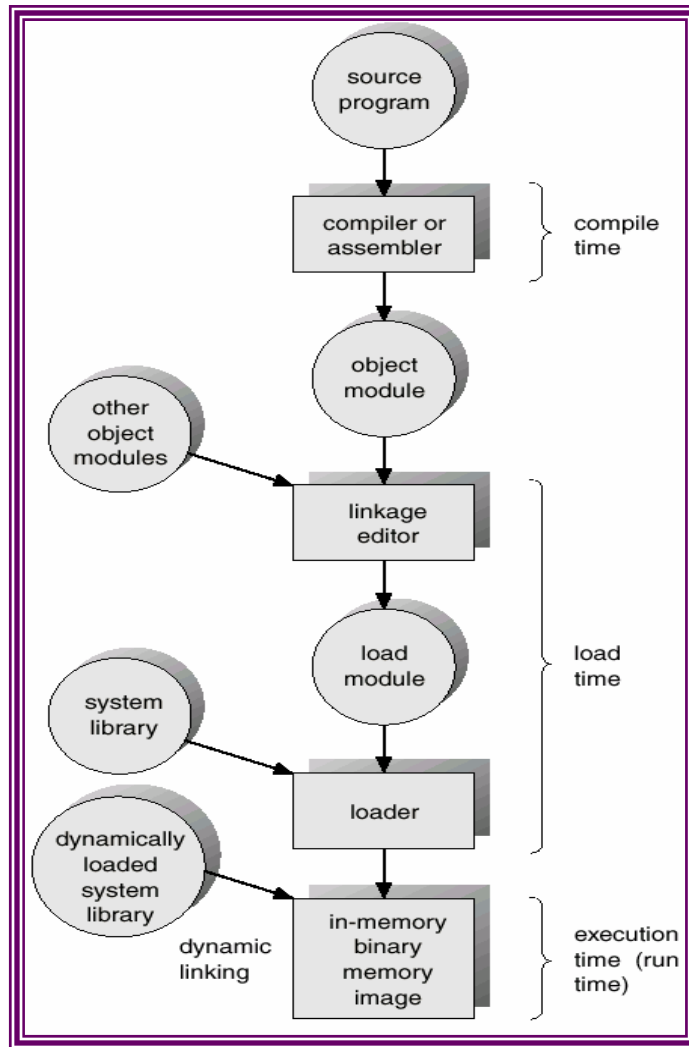
Logical- Versus Physical-Address Space

An address generated by the CPU is commonly referred to as a **logical address,** where as an address seen by the memory unit–that is, the one loaded into the **memory-address register** of the memory–is commonly referred to as the **physical address.** In essence, logical data refers to an instruction or data in the process address space where as the physical address refers to a main memory location where instruction or data resides.

The compile time and load time binding methods generate identical logical and physical addresses, where as the execution time binding method results in different physical and logical addresses. In this case we refer to the logical address as the **virtual address.** The set of all logical addresses generated by a program form the **logical address space** of a process; the set of all physical addresses corresponding to these logical addresses is a **physical address space** of the process. The total size of physical address space in a system is equal to the size of its main memory.

The run-time mapping from virtual to physical addresses is done by a piece of hardware in the CPU, called the **memory management unit (MMU).**

Translation Examples

In the following two diagrams, we show two simple ways of translating logical addresses into physical address. In both case, there is a "base" register which is loaded with the address of the first byte in the program (instruction or data—in case of the second example, separate registers are used to point to the beginning of code, data, and stack portions of a program). In the first case, the base register is called the **relocation register**. The logical address is translated into the corresponding physical address by adding the logical address to the value of the relocation register, as shown below.
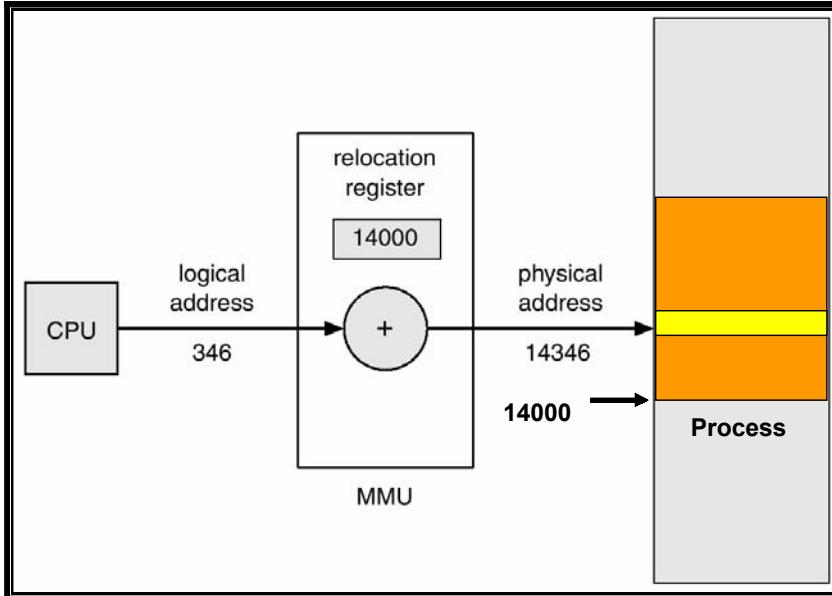
In i8086, the logical address of the next instruction is specified by the value of **instruction pointer** (IP). The physical address for the instruction is computed by shifting the **code segment register** (CS) left by four bits and adding IP to it, as shown below.
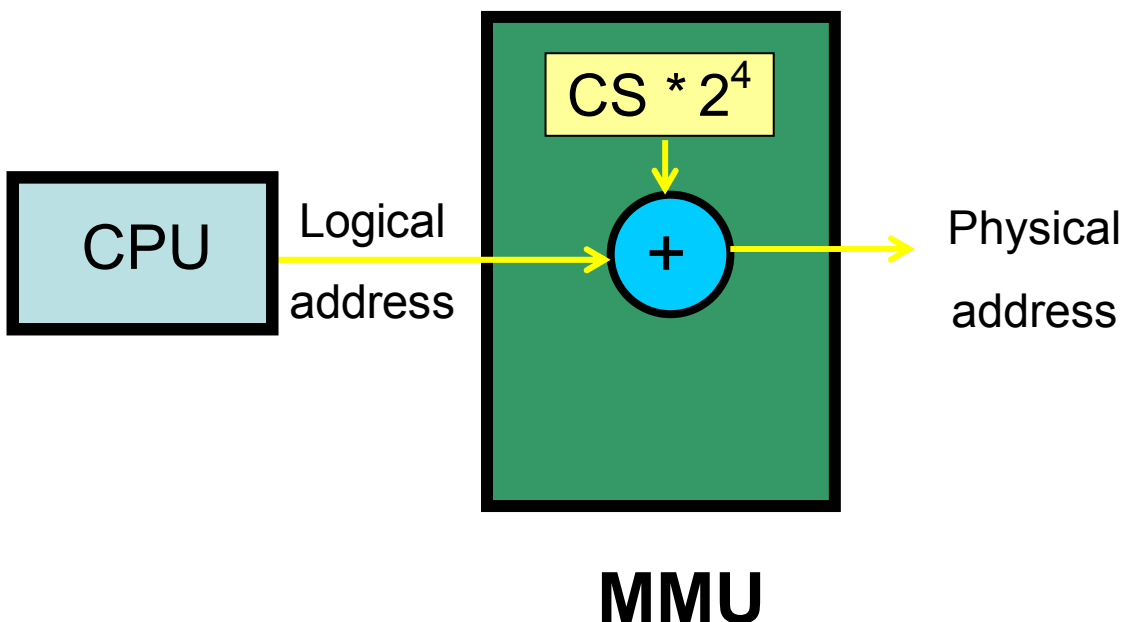
In the following example, we show the logical address for a program instruction and computation of physical address for the given logical address.

- Logical address (16-bit)
  IP = 0B10h
  CS = D000h
- Physical address (20-bit)
  CS * 24 + IP = D0B10h

Various techniques for memory management

Here are some techniques of memory management, which are used in addition to the main techniques of memory management such as paging and segmentation discussed later in the course.

Dynamic Loading

The size of a process is limited to the size of physical memory. To obtain better memory-space utilization, we can use **dynamic loading**. With dynamic loading, a routine is not loaded until it is called. All routines are kept on a disk in a re-locatable format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded or not. If not, the re-locatable linking loader is called to load the desired routine into the memory and to update the program's address tables to reflect this change. The control is then passed to the newly loaded routine.

The advantage of dynamic loading is that an unused routine is never loaded. This means that potentially less time is needed to load a program and less memory space is required. However the run time activity involved in dynamic loading is a disadvantage. Dynamic programming does not require special support from the operating system.

Dynamic Linking and Shared Libraries

Some operating systems support only **static linking** in which system language libraries are treated like any other object module and are combined by the loader into the binary proper image. The concept of dynamic linking is similar to that of dynamic loading. Rather than the loading being postponed until execution time, linking is postponed until run-time. This feature is usually used with system libraries. Without this facility, all programs on a system need to have a copy of their language library included in the executable image. This requirement wastes both disk space and main memory. With dynamic linking, a stub is included in the image for each library-routine reference. This *stub* is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present. During execution of a process, stub is replaced by the address of the relevant library code and the code is executed .If library code is not in memory, it is loaded at this time

This feature can be extended to update libraries. A library may be replaced by a new version and all programs that reference the library will automatically use the new version without any need to be re-linked. More than one version of a library may be loaded into the memory and each program uses its version information to decide which copy of the library to use. Only major changes increment the version number. Only programs that are

compiled with the new library version are affected by the incompatible changes incorporated in it. Programs linked before the new library was installed will continue using the older library. This system is also known as **shared libraries**.

Dynamic linking requires potentially less time to load a program. Less disk space is needed to store binaries. However it is a time-consuming run-time activity, resulting in slower program execution. Dynamic linking requires help from the operating system. The `gcc` compiler invokes dynamic linking by default. The `-static` option allows static linking.

Operating Systems
Lecture No. 31

Reading Material
- Chapter 9 of the textbook
- Lecture 31 on Virtual TV

Summary
- Overlays
- Swapping
- Contiguous memory allocation
- MFT

Overlays

To enable a process to be larger than the amount of memory allocated to it, we can use **overlays.** The idea of overlays is to keep in memory only those instructions and data that are needed at any given time. When other instructions are needed, they are loaded into space occupied previously by instructions that are no longer needed. We illustrate the concept of overlays with the example of a two-pass compiler. Here are the various specifications:

- 2-Pass assembler/compiler
- Available main memory: 150k
- Code size: 200k
  - Pass 1 ……………….. 70k
  - Pass 2 ……………….. 80k
  - Common routines …... 30k
  - Symbol table ………… 20k

Common routines, symbol table, overlay driver, and Pass 1 code are loaded into the main memory for the program execution to start. When Pass 1 has finished its work, Pass 2 code is loaded on top of the Pass 1 code (because this code is not needed anymore). This way, we can execute a 200K process in a 150K memory. The diagram below shows this pictorially.

The problems with overlays are that a) you may not be able to partition all problems into overlays, and b) programmer is responsible of writing the overlays driver.
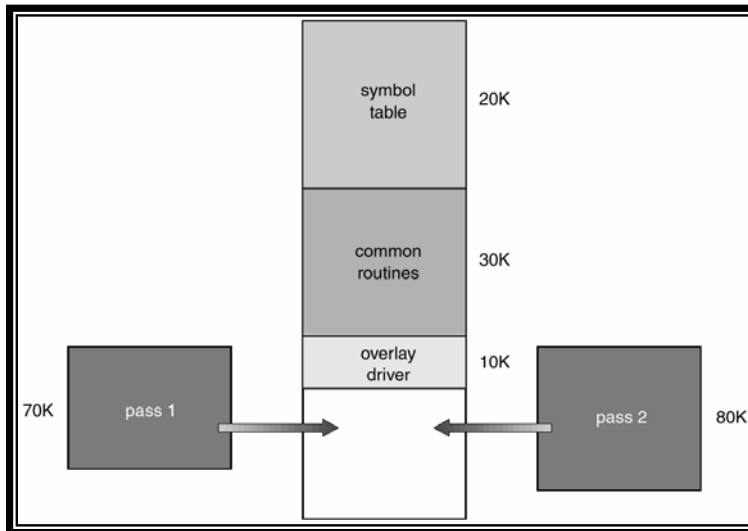
Overlays Example

Swapping

A process needs to be in the memory to be executed. A process, however, can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution. Backing store is a fast disk large enough to accommodate copies of all memory images for all users; it must provide direct access to these memory images. The system maintains a *ready queue* of all processes whose memory images are on the backing store or in memory and are ready to run.

For example, assume a multiprogramming environment with a round robin CPU scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been freed. A variant of this swapping policy can be used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manger can swap out the lower-priority process so that it can load and execute the higher--priority process. When the higher--priority process finishes, the lower--priority process can be swapped back in and continued. This technique is called **roll out, roll in**.

The major part of swap time is transfer time; the total transfer time is directly proportional to the *amount* of memory swapped.

Swapping is constrained by factors like quantum for RR scheduler and pending I/O for swapped out process. Assume that I/O operation was queued because the device was busy. Then if we were to swap out P1, and swap in process P2, the I/O operation might attempt to access memory that now belongs to P2.The solution to this problem are never to swap out processes with pending I/O or to execute I/O in kernel space

157
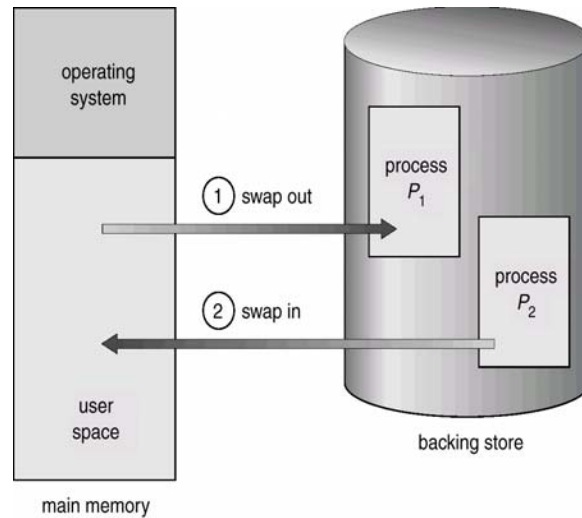
Schematic View of Swapping

Cost of Swapping
Process size   = 1 MB
Transfer rate   = 5 MB/sec
Swap out time   = 1/5 sec
                           = 200 ms
Average latency   = 8 ms
Net swap out time   = 208 ms
Swap out + swap in   = 416 ms

Contiguous memory allocation

The main memory must accommodate both operating system and the various user spaces. Thus memory allocation should be done efficiently.

The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. The operating system may be placed in the high memory or the low memory. The position of the interrupt vector usually affects this decision. Since the interrupt vector is often in the low memory, programmers place the OS in low memory too.

- It is desirable to have several user processes residing in the memory at the same time. In contiguous memory allocation, each process is contained in a single contiguous section of memory. The **base** (re-location) and **limit** registers are used to point to the smallest memory address of a process and its size, respectively.
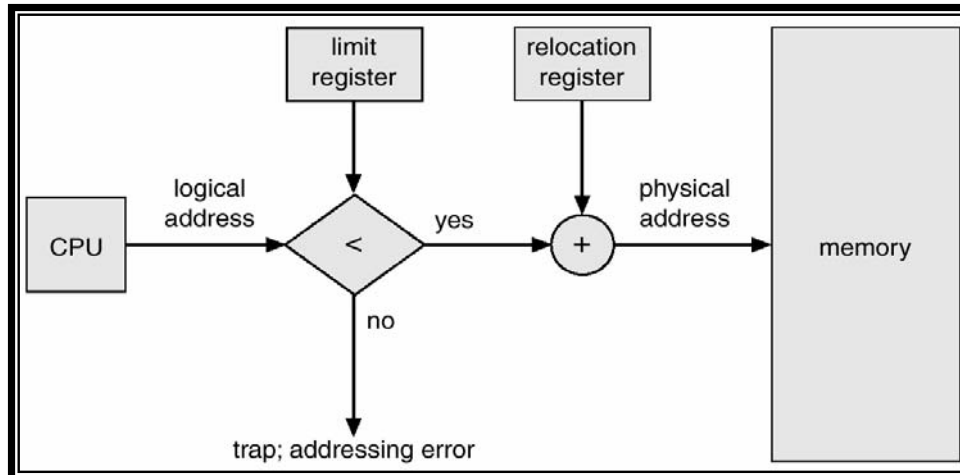
Contiguous Allocation

Multiprogramming with Fixed Tasks (MFT)

In this technique, memory is divided into several fixed-size partitions. Each partition may contain exactly one process. Thus the degree of multiprogramming is bound by the number of partitions. In this multiple partition method, when a partition is free, a process is selected from the input queue and is loaded in the free partition. When the process terminates, the partition becomes available for another process.

- This was used by IBM for system 360 OS/MFT (multiprogramming with a fixed number of tasks).
- Can have a single input queue instead of one for each partition.
  - So that if there are no big jobs can use big partition for little jobs.
  - Can think of the input queue(s) as the ready list(s) with a scheduling policy of FCFS in each partition.
- The partition boundaries are *not* movable and are set at boot time (must reboot to move a job).
  - MFT can have large **internal fragmentation**, i.e., wasted space *inside* a region
- Each process has a single ``segment'' (we will discuss segments later)
  - No sharing between processes.
  - No dynamic address translation.
  - At load time must ``establish addressability''.
  - Must set a base register to the location at which the process was loaded (the bottom of the partition).
  - The base register is part of the programmer visible register set.
  - This is an example of address translation during load time.
  - Also called **relocation**.
- Storage keys are adequate for protection (IBM method).
- Alternative protection method is base/limit registers.
- An advantage of base/limit is that it is easier to move a job.
- But MFT didn't move jobs so this disadvantage of storage keys is moot.
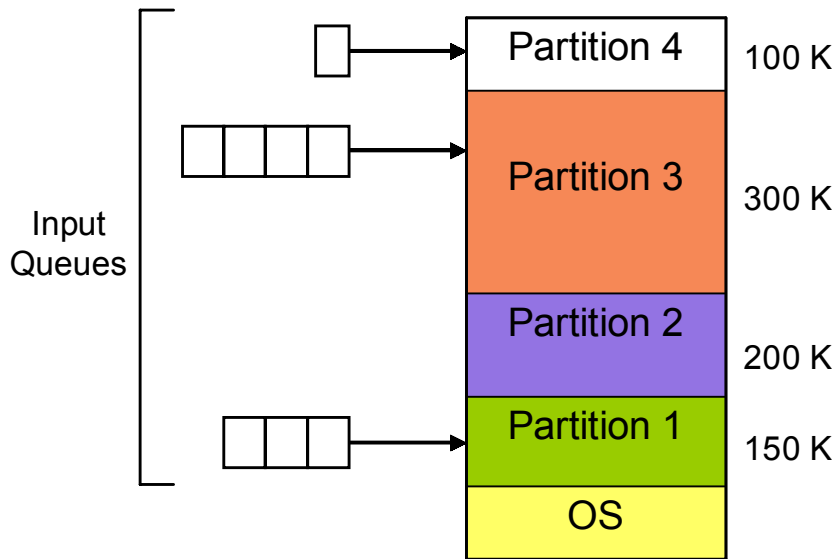
Multiprogramming with Fixed Tasks (MFT) with a queue per partition

MFT with multiple queues involves load-time address binding. In this technique, there is a potential for wasted memory space i.e. an empty partition but no process in the associated queue. However in MFT with single queue there is a single queue for each partition. The queue is searched for a process when a partition becomes empty. *First-fit*, *best-fit*, *worst-fit* space allocation algorithms can be applied here. The following diagram shows MFT with single input queue.
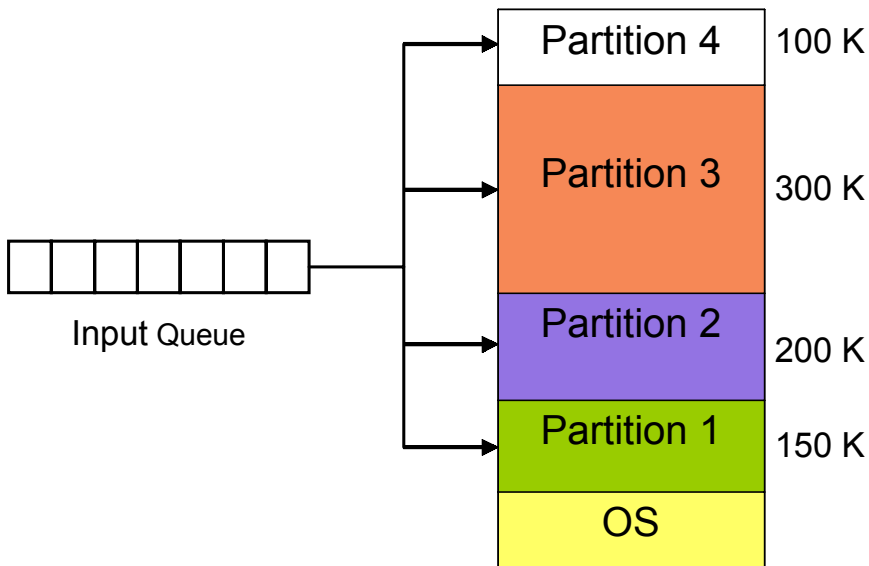
Multiprogramming with Fixed Tasks (MFT) with one input queue

Operating Systems
Lecture No. 32

Reading Material
- Chapter 9 of the textbook
- Lecture 32 on Virtual TV

Summary
- MVT
- Paging
- Logical to physical address translation

Multiprogramming with Variable Tasks (MVT)

This is the generalization of the fixed partition scheme. It is used primarily in a batch environment. This scheme of memory management was first introduced in IBM OS/MVT (multiprogramming with a varying number of tasks). Here are the main characteristics of MVT.

- Both the number and size of the partitions change with time.
- Job still has only one segment (as with MFT) but now can be of any size up to the size of the machine and can change with time.
- A single ready list.
- Job can move (might be swapped back in a different place).
- This is dynamic address translation (during run time).
- Must perform an addition on every memory reference (i.e. on every address translation) to add the start address of the partition.
- **Eliminates internal fragmentation**.
  - ➢ Find a region the exact right size (leave a hole for the remainder).
  - ➢ Not quite true, can't get a piece with 10A755 bytes. Would get say 10A760. But internal fragmentation is *much* reduced compared to MFT. Indeed, we say that internal fragmentation has been eliminated.
- **Introduces external fragmentation**, i.e., holes *outside* any region.
- What do you do *if no hole is big enough* for the request?
  - ➢ Can compact memory
    - Transition from bar 3 to bar 4 in diagram below.
    - This is expensive.
    - Not suitable for real time systems.
  - ➢ Can swap out one process to bring in another
    - Bars 5-6 and 6-7 in the following diagram
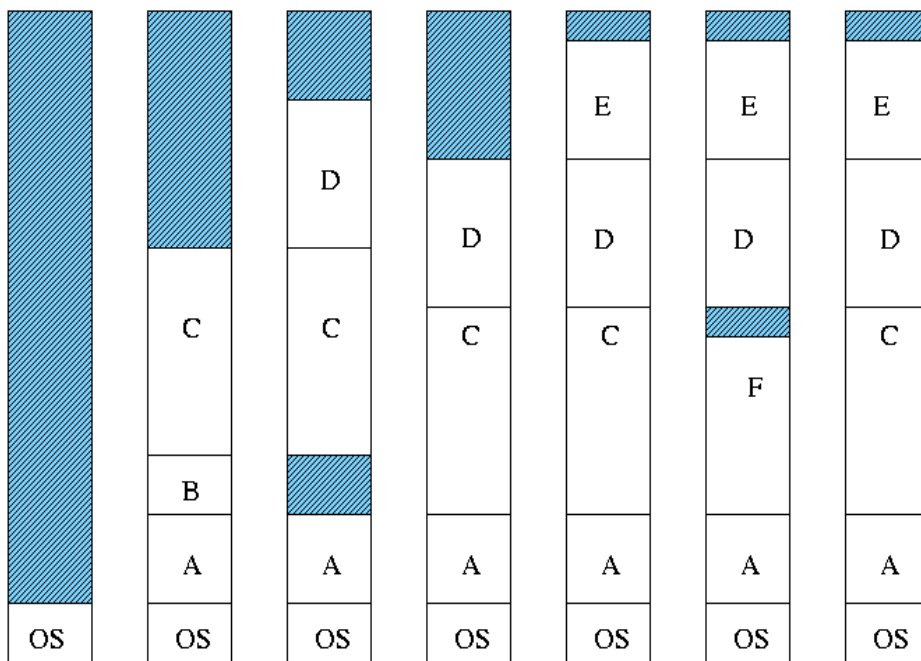
Multiprorgamming with Variable Tasks (MVT), external fragmentation, and compaction

External fragmentation

As processes come and go, *holes* of free space are created in the main memory. External Fragmentation refers to the situation when free memory space exists to load a process in the memory but the space is not contiguous. Compaction eliminates external fragmentation by shuffling memory contents (processes) to place all free memory into one large block. The cost of compaction is slower execution of processes as compaction takes place.

Paging

In the memory management techniques discussed so far, two Paging is a memory management scheme that permits the physical address space of a process to be non-contiguous. It avoids the considerable problem of fitting the various sized memory chunks onto the backing store, from which most of the previous memory-management schemes suffered. When some code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. The fragmentation problems discussed in connection with main memory are also prevalent with backing store, except that access is much slower so compaction is impossible.

Physical memory is broken down into fixed-sized blocks, called **frames**, and logical memory is divided into blocks of the same size, called **pages**. The size of a page is a power of 2, the typical page table size lying between 1K and16K. It is important to keep track of all free frames. In order to run a program of size *n* pages, we find *n* free frames and load program pages into these frames. In order to keep track of a program's pages in the main memory a **page table** is used.

Thus when a process is to be executed, its pages are loaded into any available memory frames from the backing store. The following snapshots show process address space with pages (i.e., logical address space), physical address space with frames, loading of paging into frames, and storing mapping of pages into frames in a page table.

Process Address Space

Physical Address Space

a) Logical and physical address spaces

Process Address Space
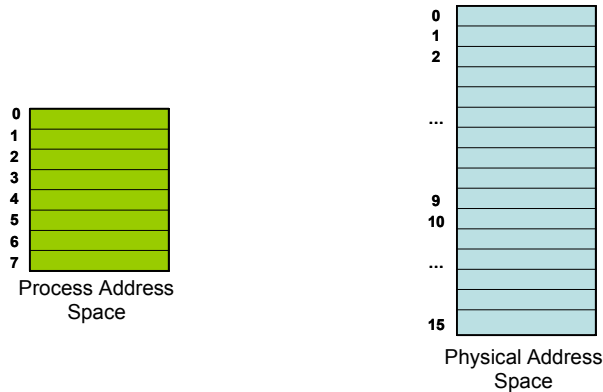
Page

Page Table

Physical Address Space
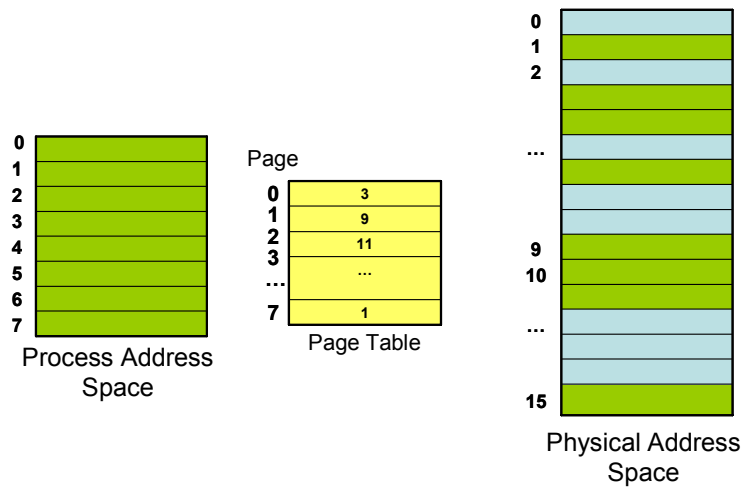
b) Mapping paging in the logical into the frames in the physical address space and keeping this mapping in the page table

Every **logical address** generated by the CPU is divided into two parts: a page number (p) and a page offset (d). The page table contains the base address (frame number) of each page in physical memory. The frame number is combined with the page offset to obtain the physical memory address of the memory location that contains the object addressed by the corresponding logical address. Here p is used to index the process page table; page table entry contains the frame number, f, where page p is loaded. The **physical address** of the location referenced by (p,d) is computed by appending d at the end of f, as shown below:

f	d
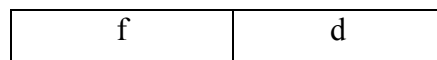
The hardware support needed for this address translation is shown below.
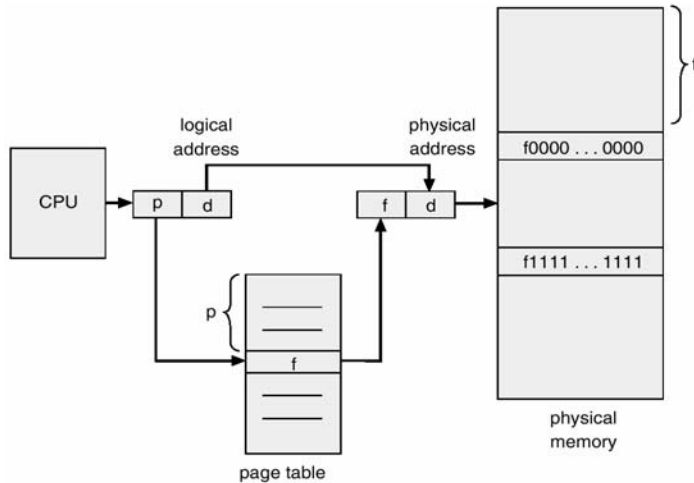
Hardware support for paging

Paging itself is a form of dynamic relocation. When we use a paging scheme, we have no external fragmentation; however we may have **internal fragmentation**. An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory. The user views that memory as one single contiguous space, containing only this program. In fact, the user program is scattered throughout the physical memory, which also holds other programs.

Paging Example

- Page size   = 4 bytes
- Process address space   = 4 pages
- Physical address space   = 8 frames
- Logical address: (1,3)   = 0111
- Physical address: (6,3)   = 1011
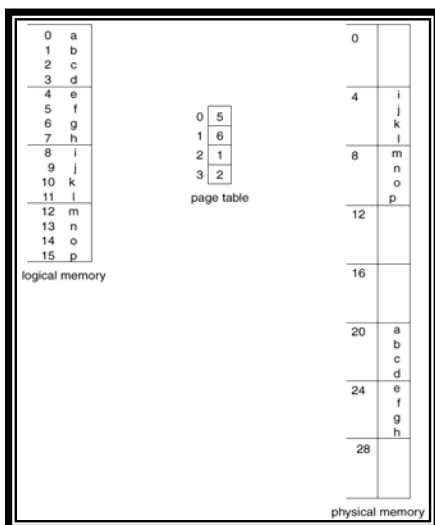
Operating Systems
Lecture No. 33

Reading Material
- Chapter 9 of the textbook
- Lecture 33 on Virtual TV

Summary
- Addressing and logical to physical address translation
- Examples: Intel P4 and PDP-11
- Page table implementation
- Performance of paging

Addressing in Paging

The page size is defined by the CPU hardware. If the size of logical address space is 2^m and a page size is 2^n addressing units (bytes or words) , then the high-order *m-n* bits of a logical address designate the page number and the *n* low order bits designate offset within the page. Thus, the logical address is as follows:

page number	page offset
p	d
m-n bits	n bits
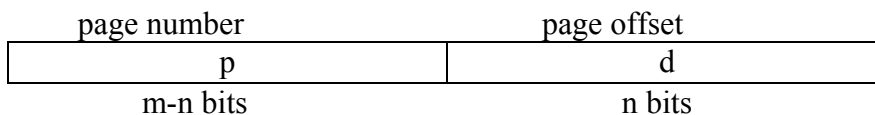
Example:

Assume a **logical address space** of 16 pages of 1024 words, each mapped into a physical memory of 32 frames. Here is how you calculate the various parameters related to paging.

No. of bits needed for **p** = ceiling [$\log_2$ 16] bits = 4 bits
No. of bits neede for **f** = ceiling [$\log_2$ 32] bits = 5 bits
No. of bits needed for **d** = ceiling [$\log_2$ 2048] bits = 11 bits
Logical address size   = |p| + |d| = 4+11 bits = 15 bits
Physical address size = |f| + |d| = 5+11 bits = 16 bits

Page Table Size

Page table size= NP * PTES , where NP is the number of pages in the process address space and PTES is the page table entry size (equal to |f| based on our discussion so far).

Page table size = 16 * 5 bits (for the above example; assuming a byte size page table entry)

Paging in Intel P4

32-bit *linear* address

4K page size

Maximum pages in a process address space $= 2^{32} / 4K$
Number of bits needed for **d** $= \log_2 4K$ bits $= 12$ bits
Number of bits needed for **p** $= 32 - 12$ bits $= 20$

Paging in PDP-11
16-bit logical address
8K page size

Maximum pages in a process address space $= 2^{16} / 8K = 8$
$|\mathbf{d}| = \log_2 8K = 13$ bits
$|\mathbf{p}| = 16 - 13 = 3$ bits

Another Example
Logical address = 32-bit

Process address space $= 2^{32}$ B $= 4$ GB
Main memory = RAM = 512 MB
Page size = 4K

Maximum pages in a process address space $= 2^{32} / 4K = 1M$
$|\mathbf{d}| = \log_2 4K = 12$ bits
$|\mathbf{p}| = 32 - 12 = 20$ bits
No. of frames $= 512$ M $/ 4$ K $= 128$ K
$|\mathbf{f}| =$ ceiling $[\log_2 128$ K] bits $= 17$ bits ≈ 4 bytes (rounding to next even-numbered byte)
Physical address $= 17+12$ bits

Implementation of Page table

▪ In the CPU registers
This is OK for small process address spaces and large page sizes. It has the advantage of having *effective memory access time* ($T_{effective}$) about the same as memory access time (T_{mem}). An example of this implementation is in PDP-11.

▪ In the main memory
A page table base register (PTBR) is needed to point to the page table. With page table in main memory, the effective memory access time, $T_{effective}$, is $2T_{mem}$, which is <u>not</u> acceptable because it would slow down program execution by a factor of two.

▪ In the translation look-aside buffer (TLB)
A solution to this problem is to use special, small, fast lookup hardware, called translation look-aside buffer (TLB), which typically has 64–1024 entries. Each entry is (key, value). The key is searched for in parallel; on a hit, value is returned. The (key,value) pair is (p,f) for paging. For a logical address, (p,d), TLB is searched for p. If an entry with a key p is found, we have a <u>hit</u> and f is used to form the physical address. Else, page table in the main memory is searched.
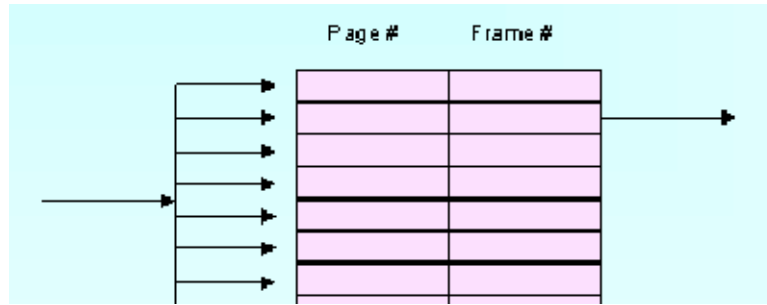
TLB –Logical address: (p,d)

The TLB is loaded with the (p,f) pair so that future references to p are found in the TLB, resulting in improved hit ratio. On a context switch, the TLB is flushed and is loaded with values for the scheduled process. Here is the hardware support needed for paging with part of the page table stored in TLB.
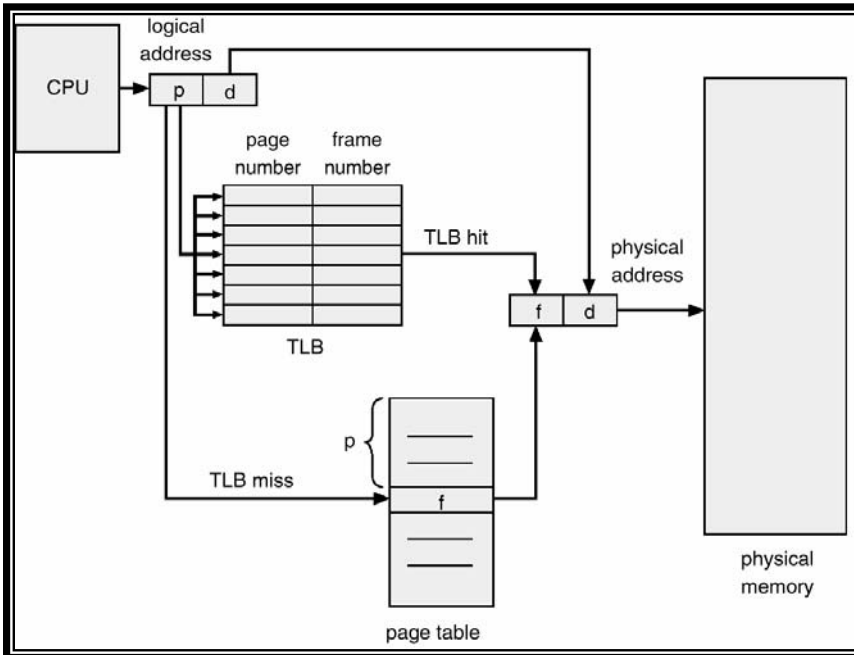
Paging Hardware with TLB

Performance of Paging

We discuss performance of paging in this section. The performance measure is the effective memory access time. With part of the page table in the TLB and the rest in the main memory, the effective memory access time on a hit is $T_{mem} + T_{TLB}$ and on a miss is $2T_{mem} + T_{TLB}$.

If HR is hit ratio and MR is miss ratio, the effective access time is given by the following equation

$$T_{effective} = HR\ (T_{TLB} + T_{mem}) + MR\ (T_{TLB} + 2T_{mem})$$

We give a few examples to help you better understand this equation.

Example 1

T_{mem} = 100 nsec
T_{TLB} = 20 nsec
Hit ratio is 80%
$T_{effective}$ = 0.8 (20 + 100) + 0.2 (20 + 2*100) nanoseconds = 140 nanoseconds

This means that with 80% chances of finding a page table entry in the TLB, the effective access time becomes 40% worse than memory access time without paging.

Example 2

T_{mem} = 100 nsec
T_{TLB} = 20 nsec
Hit ratio is 98%
$T_{effective}$ = 0.98 (20 + 100) + 0.02 (20 + 2*100) nanoseconds = 122 nanoseconds

This means that with 98% chances of finding a page table entry in the TLB, the effective access time becomes 22% worse than memory access time without paging. This means that with a small cache and good hit ratio, we can maintain most of the page table in the main memory and get much better performance than keeping the page table in the main memory and not using any cache.

Operating Systems
Lecture No. 34

Reading Material
- Chapter 9 of the textbook
- Lecture 34 on Virtual TV

Summary
- Protection under paging
- Structure of the page table
  - Multi-level paging
  - Hashed page tables
  - Inverted page table

Protection under Paging

Memory protection in paging is achieved by associating protection bits with each page. These bits are associated with each page table entry and specify protection on the corresponding page. The primary protection scheme guards against a process trying to access a page that does not belong to its address space. This is achieved by using a valid/invalid (v) bit. This bit indicates whether the page is in the process address space or not. If the bit is set to invalid, it indicates that the page is not in the process's logical address space. Illegal addresses are trapped by using the valid-invalid bit and control is passed to the operating system for appropriate action. The following diagram shows the use of v bit in the page table. In this case, logical address space is six page and any access to pages 6 and 7 will be trapped because the v bits for these pages is set to invalid.
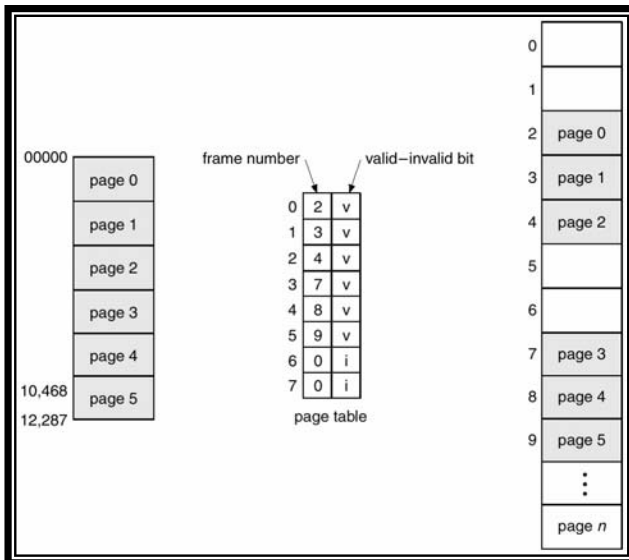
Use of valid/invalid (v) bit for protection under paging

One bit can define the page table to be read and write or read only. Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read only page. An attempt to write to a read-only page causes a hardware trap to the operating system (memory-protection violation).

This approach can be expanded to provide a finer level of protection. Read, write, and execute bits (r, w, x) can be used to allow a combination of these accesses, similar to the file protection scheme used in the UNIX operating system. Illegal attempts will be trapped to the operating system.

Structure of the Page Table

As logical address spaces become large (32-bit or 64-bit), depending on the page size, page table sizes can become larger than a page and it becomes necessary to page the page the page table. Additionally, large amount of memory space is used for page table. The following schemes allow efficient implementations of page tables.

- Hierarchical / Multilevel Paging
- Hashed Page Table
- Inverted Page Table

Hierarchical/Multilevel Paging

Most modern computers support a large logical address space: (2^{32} to 2^{64}). In such an environment, the page table itself becomes excessively large. Consider the following example:

- Logical address = 32-bit
- Page size = 4K bytes (212 bytes)
- Page table entry = 4 bytes
- Maximum pages in a process address space = 232 / 4K = 1M
- Maximum pages in a process address space = 232 / 4K = 1M
- Page table size = 4M bytes

This page table cannot fit in one page. One solution is to page the page table, resulting in a 2-level paging. A page table needed for keeping track of pages of the page table— called the <u>outer page table</u> or <u>page directory</u>. In the above example:

- No. of pages in the page table is 4M / 4K = 1K
- Size of the outer page table is 1K * 4 bytes = 4K bytes $\Rightarrow$ outer page will fit in one page

In the 32-bit machine described above, we need to partition p into two parts, p1 and p2. p1 is used to index the outer page table and p2 to index the inner page table. Thus the logical address is divided into a page number consisting of 20 bits and a page offset of 12 bits. Since we page the page table, the page number is further divided into a 10-bit page number, and a 10-bit page offset. This is known as **two-level paging**. The following diagram shows division of the logical address in 2-level paging and hierarchical views of the page table.

Outer page table index	Inner page table index	Page offset
p1	**p2**	**d**
10 bits	10 bits	12 bits

a) Logical address
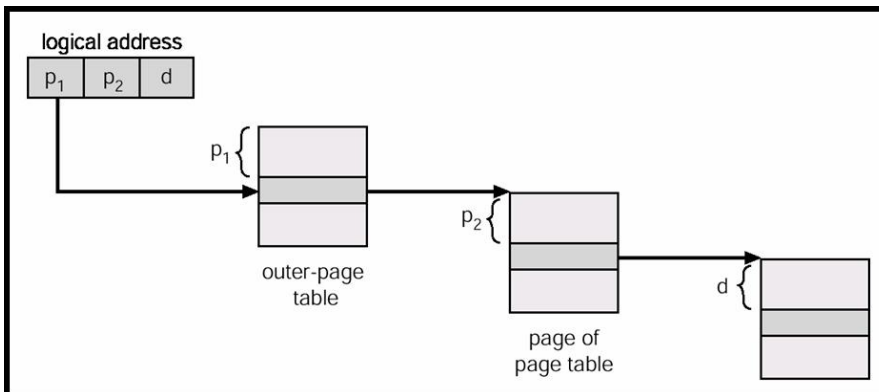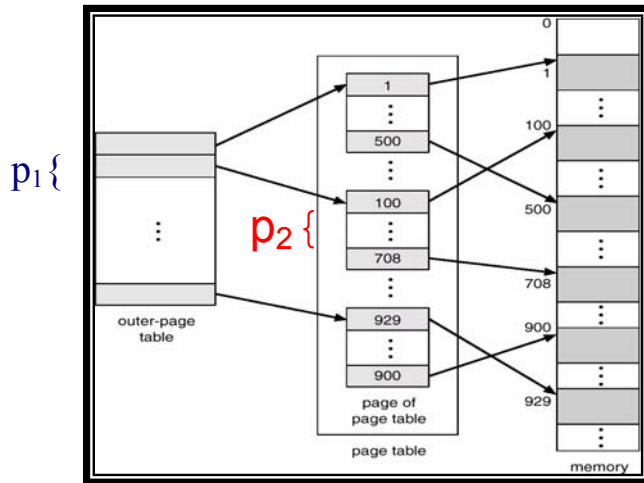
b) Two views of address translation for a two-level paging architecture

Another Example: DEC VAX
- Logical address = 32 bits
- Page size = 512 bytes = 2^9 bytes
- Process address space divided into <u>four</u> equal sections
- Pages per section = $2^{30} / 2^9 = 2^{21} = 2M$
- Size of a page table entry = 4 bytes
- Bits needed for <u>page offset</u> = $\log_2 512 = 9$ bits
- Bits to <u>specify a section</u> = $\log_2 4 = 2$ bits
- Bits needed to <u>index page table for a section</u> = $\log_2 2^{21} = 21$ bits
- <u>Size of a page table</u> = $2^{21} * 4 = 8$ MB
- 8 MB page table is paged into 8MB / 512 = 2 K pages

- Size of the outer page table (2K * 4 = 8 KB) is further paged, resulting in 3-level paging per section

Section	Page number	Page offset
s	**p**	**d**
2	21	9

More Examples

- 32-bit Sun SPARC supports 3-level paging
- 32-bit Motorola 68030 supports 4-level paging
- 64-bit Sun UltraSPARC supports 7-level paging – too many memory references needed for address translation

Hashed Page Table

This is a common approach to handle address spaces larger then 32 bits .Usually open hashing is used. Each entry in the linked list has three fields: page number, frame number for the page, and pointer to the next element—(p, f, next). The page number in the logical address (specified by p) is hashed to get index of an entry in the hash table. This index is used to search the linked list associated with this entry to locate the frame number corresponding to the given page number. The advantage of hashed page tables is smaller page tables.
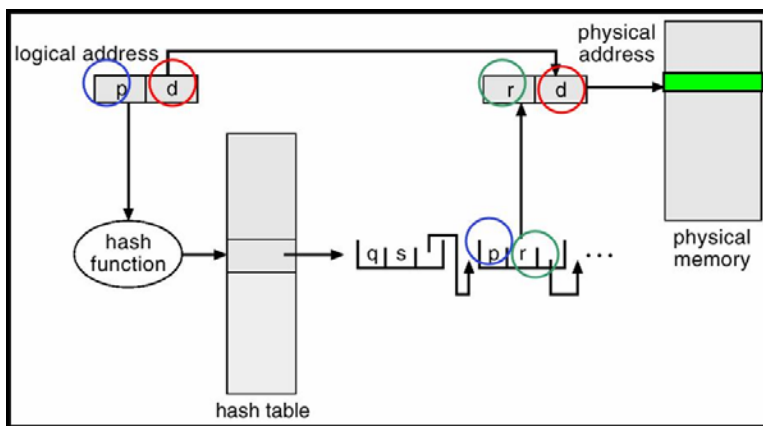
Inverted Page Table

Usually each process has a page table associated with it. The page table has one entry for each page in the address space of the process. For large address spaces (32-bit and above), each page table may consist of millions of entries. These tables may consume large amounts of physical memory, which is required just to keep track of how the mapping of logical address spaces of processes onto the physical memory.

A solution is to use an inverted page table. An **inverted page table** has one entry for each real page (frame) of memory. Each entry consists of the virtual address of the page stored in the in that real memory location, with information about the process that own the page.

Page table size is limited by the number of frames (i.e., the physical memory) and not process address space. Each entry in the page table contains (pid, p). If a page 'p' for a process is loaded in frame 'f', its entry is stored at index 'f' in the page table. We effectively index the page table with frame number; hence the name inverted page table. Examples of CPUs that support inverted pages tables are 64-bit UltraSPARC and PowerPC. The following diagram shows how logical addresses are translated into physical addresses.
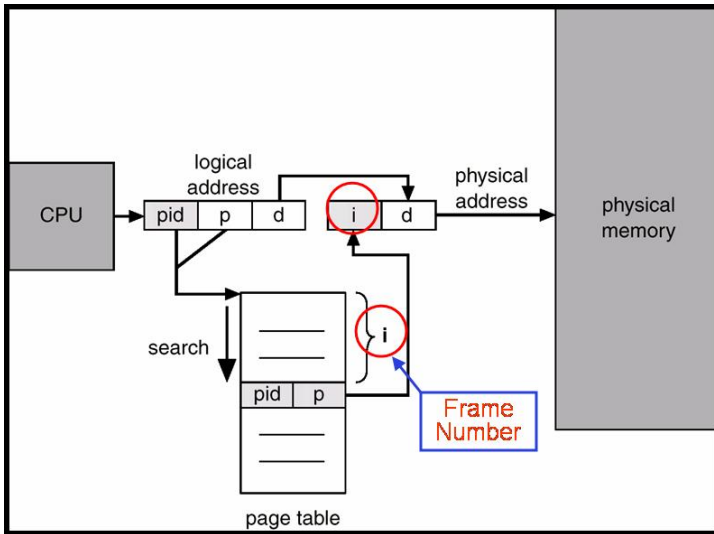
Address translation with inverted page table

Operating Systems
Lecture No. 35

Reading Material
- Chapter 9 of the textbook
- Lecture 35 on Virtual TV

Summary
- Sharing in paging
- Segmentation
- Logical to physical address translation
- Hardware support needed
- Protection and sharing

Sharing in Paging

Another advantage of paging is the possibility of *sharing* common code. Reentrant (read-only) code pages of a process address can be shared. If the code is reentrant, it never changes during execution. Thus two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process' execution. The data for two different processes will, of course, vary for each process. Consider the case when multiple instances of a text editor are invoked. Only one copy of the editor needs to be kept in the physical memory. Each user's page table maps on to the same physical copy of the editor, but data pages are mapped onto different frames. Thus to support 40 users, we need only one copy of the editor, which results in saving total space.
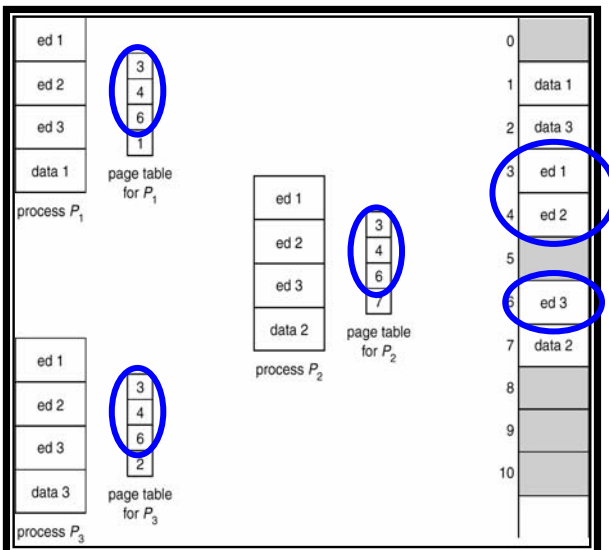
Sharing in paging

Segmentation

Segmentation is a memory management scheme that supports programmer's view of memory. A logical-address space is a collection of segments. A segment is a logical unit such as: main program, procedure, function, method, object, global variables, stack, and symbol table. Each segment has a name and length. The addresses specify both the segment name and the offset within the segment. An example of the logical address space of a process with segmentation is shown below.
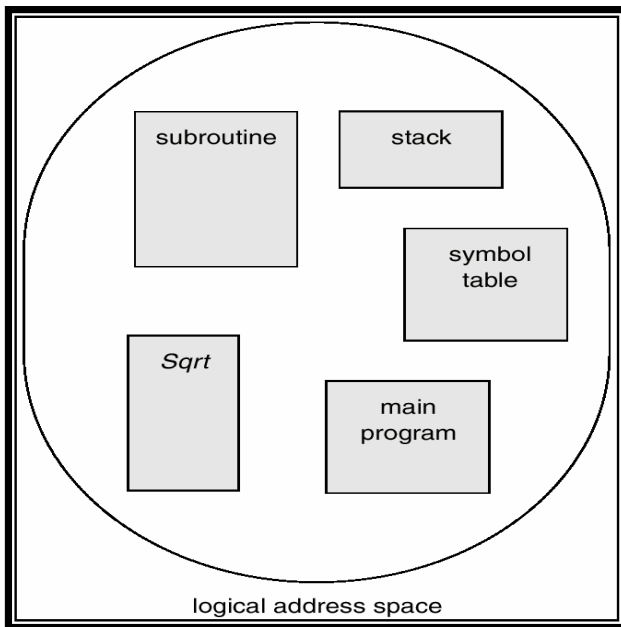
Logical address space with segmentation

For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus a logical address consists of a two tuple:

<segment-number, offset> or <s,d>

The segment table maps the two-dimensional logical addresses to physical addresses. Each entry of a segment table has a *base* and a *segment limit*. The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment.

There are two more registers, relevant to the concept of segmentation:

- **Segment-table base register** (STBR) points to the segment table's location in memory.
- **Segment-table length register** (STLR) indicates number of segments used by a program

Segment number s is legal if $s < STLR$, and offset, d, is legal if $d < limit$. The following diagram shows the hardware support needed for translating a logical address into the physical address when segmentation is used. This hardware is part of the MMU in a CPU.
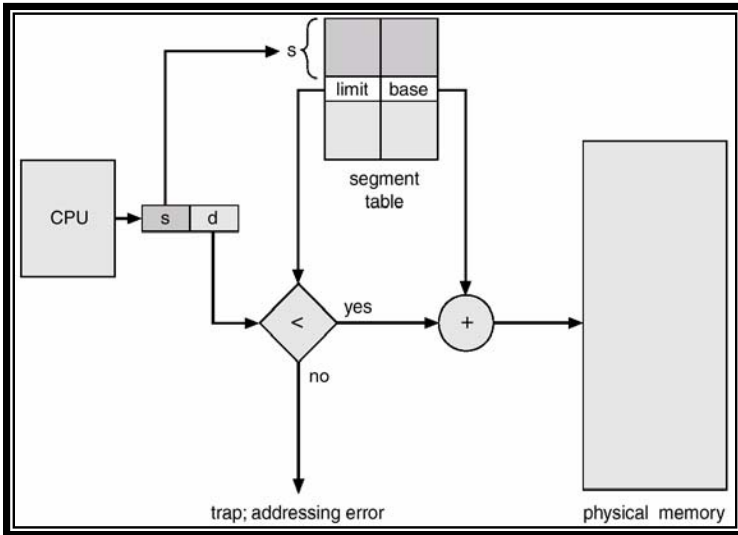
Hardware support for segmentation

For logical to physical address conversion, segment number, s, is used to index the segment table for the process. If d < limit, it is added to the base value to compute the physical address for the given logical address. The segment base and limit values are used to relocate and bound check the reference at runtime.

Sharing of Segments

Another advantage of segmentation is sharing of code or data. Each process has a segment table associated with it, which the dispatcher uses to define the hardware segment table when this process is given the CPU. Segments are shared when entries in the segment tables of two different processes point o the same physical location. The sharing occurs at segment level, thus, any information defined as a segment can be shared.
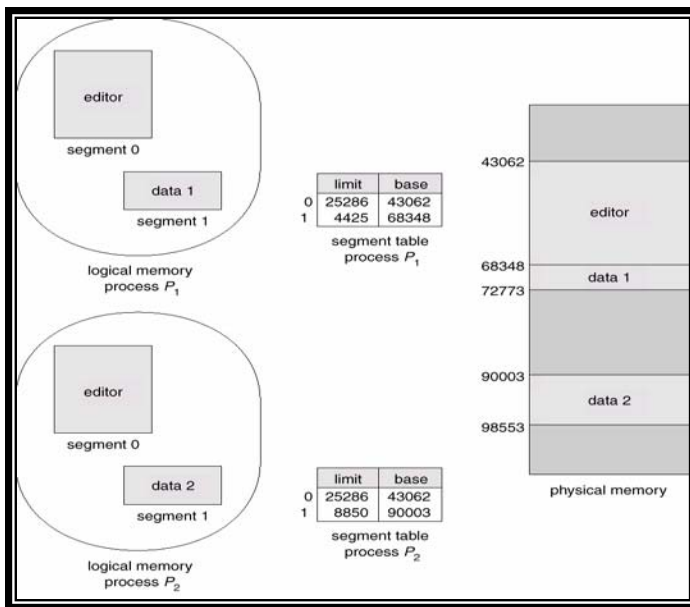
Sharing in segmentation

The long-term scheduler must find and allocate memory for all the segments of a user program. This situation is similar to paging except that the segments are of *variable* length; pages are all the same size. Thus memory allocation is a dynamic storage allocation problem, usually solved with a best fit or worst fit algorithm.

Protection

A particular advantage of segmentation is the association of protection with segments. Because the segments represent a semantically defined portion of the program, it is likely that ll the entries will be used the same way. Hence, some segments are instructions, whereas other segments are data. In a modern architecture, instructions are non-self-modifying so they can be defined as read only. Or execute only. The memory mapping hardware will check the protection bits associated with each segment-table entry top prevent illegal access to memory, such as attempts to write into a read only segment. By placing an array in its own segment, the memory management hardware will automatically check that array indexes are legal and do not stray outside array boundaries.

The bits associated with each entry in the segment table, for the purpose of protection are:
- Validation bit : if the validation bit is 0, it indicates an illegal segment
- Read, write, execute bits
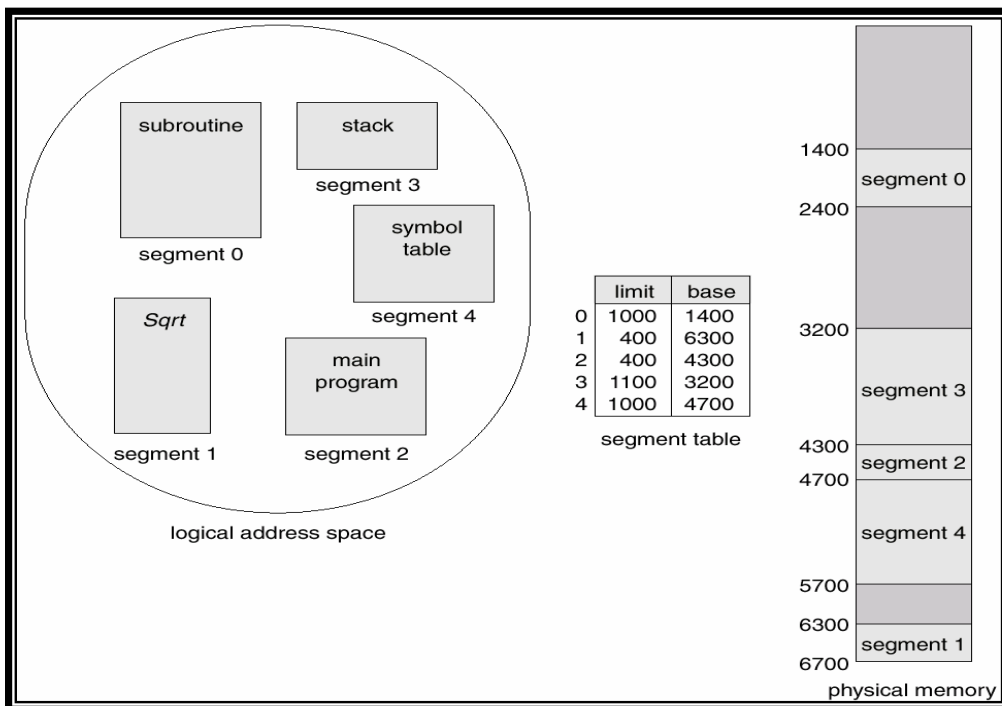
Issues with Segmentation

Segmentation may then cause external fragmentation (i.e. total memory space exists to satisfy a space allocation request for a segment, but memory space is not contiguous), when all blocks of memory are too small to accommodate a segment. In this case, the process may simply have to wait until more memory (or at least a larger hole) becomes

available or until compaction creates a larger hole. Since segmentation is by nature a dynamic relocation algorithm, we can compact memory whenever we want.

If we define each process to be one segment, this approach reduces to the variable sized partition scheme. T the other extreme, every byte could be put in its own segment and relocated separately. This eliminates external fragmentation altogether, however every byte would need a base register for its relocation, doubling memory use. The next logical step- fixed sized, small segments, is paging i.e. paged segmentation.

Also it might latch a job in memory while it is involved in I/O. To prevent this I/O should be done only into OS buffers.

Operating Systems
Lecture No. 36

Reading Material
- Chapter 9 of the textbook
- Lecture 36 on Virtual TV

Summary
- Paged segmentation
- Examples of paged segmentation: MULTICS under GE 345 and OS/2, Windows, and Linux under Intel CPUs

Paged Segmentation
In paged segmentation, we divide every segment in a process into fixed size pages. We need to maintain a page table per segment CPU's memory management unit must support both segmentation and paging. The following snapshots illustrate these points.


logical memory   physical memory

logical memory   physical memory
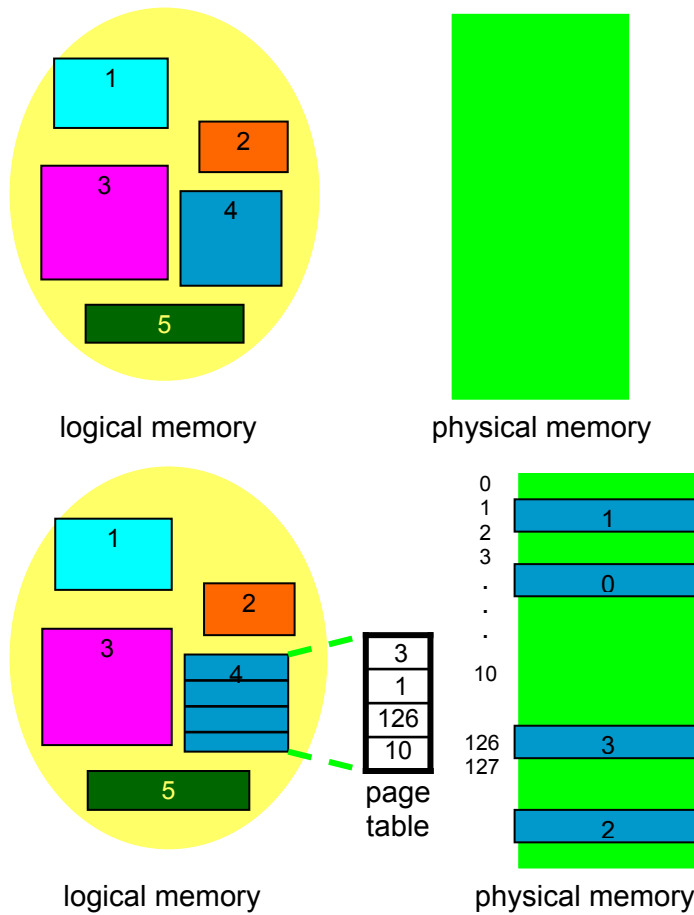
Paged Segmentation

The logical address is still <s,d>, with s used to index the segment table. Each segment table entry consist of the tuple

<center><segment-length, page-table-base></center>

The logical address is legal if d < segment-length. The segment offset, d, is partitioned into two parts: p and d', where p is used to index the page table associated with the segment s and d' is used as offset within the page p. p indexes the page table to retrieve frame, f, and physical address (f,d') is formed. The following diagrams show the format of logical address and its division, and the hardware support needed for logical to physical address translation.
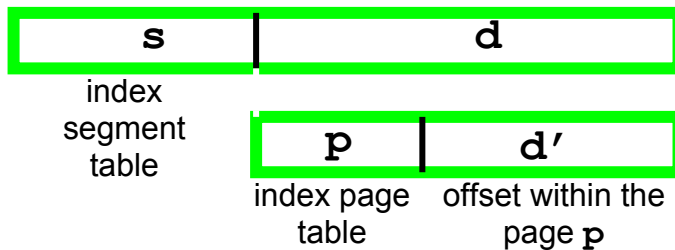
a) Logical address and its partition
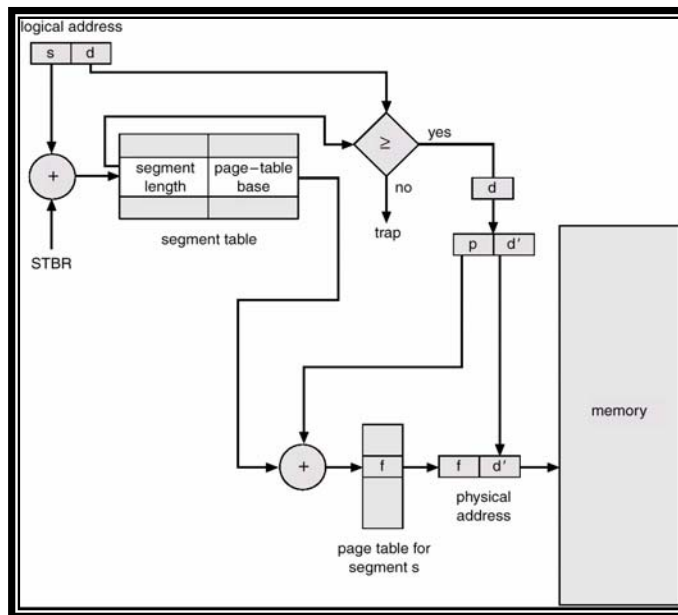
b) Hardware support needed for logical to physical address translation

MULTICS Example

We now take the example of one of the finest operating systems of late 1960s and early 1970s, known as the MULTICS operating system. Here are the specifications of the CPU supported by MULTICS and calculation of its various parameters such as the largest segment size supported by MULTICS.

- GE 345 processor
- Logical address = 34 bits

- Page size = 1 KB
- **s** is 18 bits and **d** is 16 bits
- Size of **p** and **d'**, <u>largest segment size</u>, and <u>max. number of segments per process</u>?
- Largest segment $= 2^d$ bytes $= 2^{16}$ bytes
- Maximum number of pages per segment $= 2^{16} / 1 K = 64$
- $|p| = \log2\ 64$ bits $= 6$ bits
- $|d'| = \log2\ 1 K$ $= 10$ bits
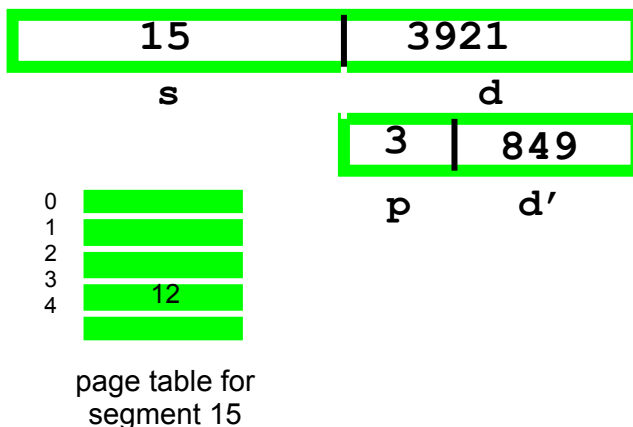- Maximum number of segments per process $= 2^s = 2^{18}$

s	d

18 bits

p	d'

6 bits   10 bits
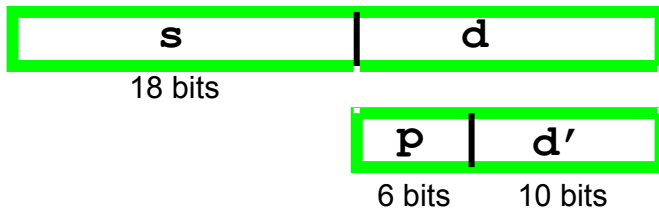
Logical address and its partition for GE645 on which MULTICS ran

    Consider a process with its segment 15 having 5096 bytes. The process generates a logical address (15,3921). Is it a legal address? How many pages does the segment have? What page does the logical address refer to? Is it a legal address? Yes
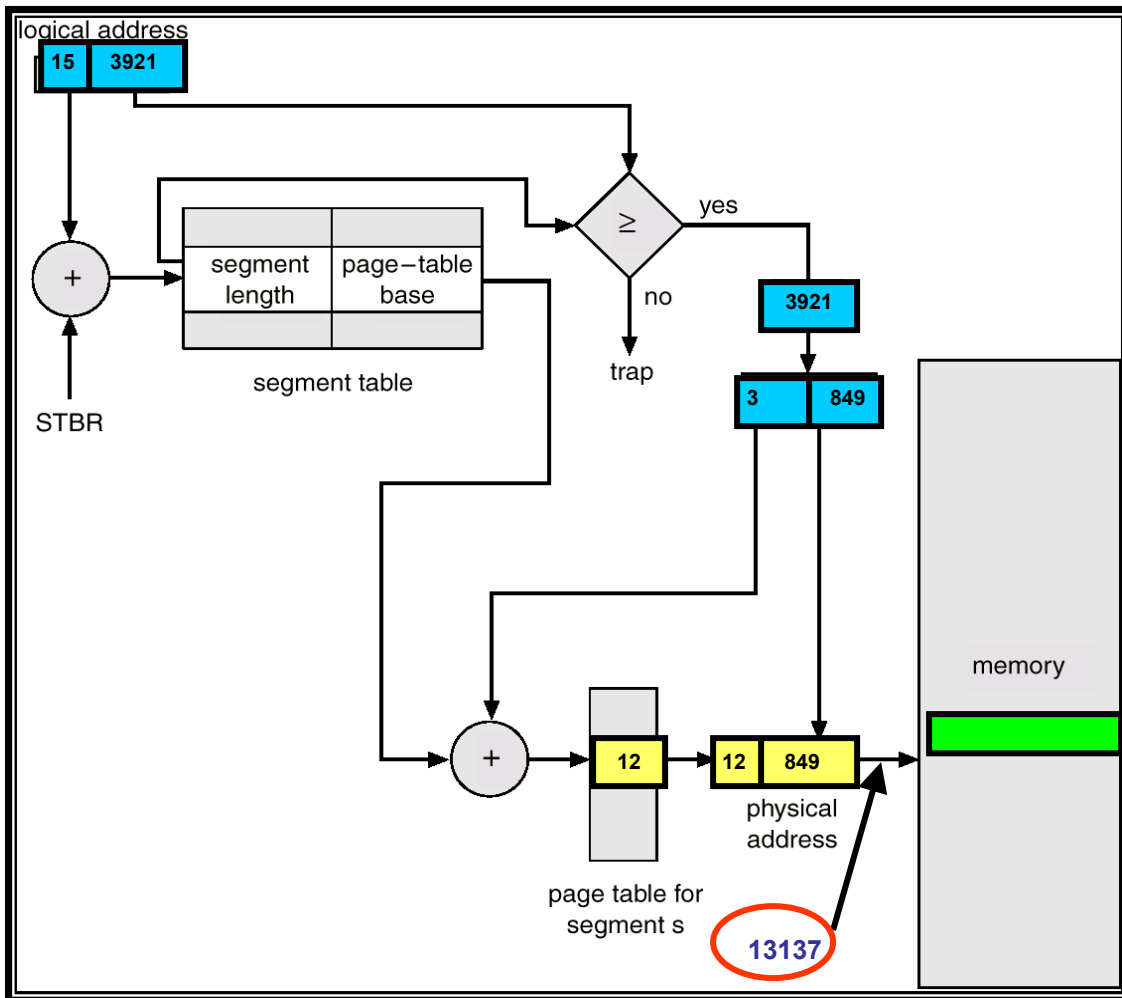
- How many pages does the segment have?
  ceiling[5096/1024]= 5
- What page does the logical address refers to?
  ceiling[3921/1024]= 4 (i.e., page number 3)
- What are the value of d' and the physical address if page number 3 (i.e., the fourth page) is in frame 12? Here is how we compute these parameters, along with the graphical representation of the various parameters. Logical to physical address translation is shown in the figure on the next page.

> **d' = 3921 − 3*1K**   **= 849**
> <u>**Physical address**</u> **= 12*1K + 849 = 13137**

15	3921

s   d

3	849

p   d'

```
0
1
2
3
4    12
```

page table for
segment 15

An example related to MULTICS

logical address

15 3921

segment length | page−table base

segment table

STBR

+

≥ yes

no

trap

3921

3 849

+

12

12 849

physical address

page table for segment s

13137

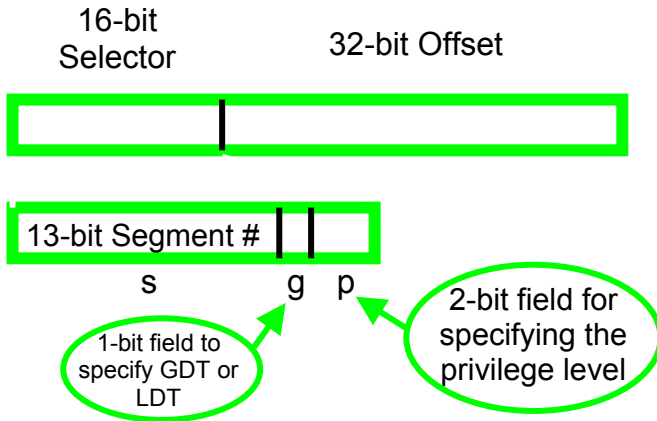memory

Intel 80386 Example
- IBM OS/2, Microsoft Windows, and Linux run on it
- Paged segmentation with two-level paging
- Logical address = 48 bits
- 16-bit selector and 32-bit offset
- Page size = 4 KB
- 4-byte page table entry
- 32-entry TLB, covering 32*4K (128 KB) memory … TLB Reach

Logical/virtual address and its division for Intel 80386 and higher

Real Mode
20-bit <u>physical address</u> is obtained by shifting left the <u>Selector</u> value by four bits and adding to it the 16-bit <u>effective address.</u>

Operating Systems
Lecture No. 37

Reading Material
- Chapters 9 and 10 of the textbook
- Lecture 37 on Virtual TV

Summary
- Intel 80386 Virtual Memory Support
- Virtual Memory Basic Concept
- Demand Paging
- Page Fault
- Performance of Demand Paging

Intel 80386 Virtual Memory Support

We discussed logical to physical address translation in the real mode operation of the Intel 80386 processor in the last lecture. Here we discuss address translation in the protected mode.

Protected Mode
- 248 bytes <u>virtual address space</u>
- 232 bytes <u>linear address space</u>
- Max segment size = 4 GB
- Max segments / process = 16K
- Six CPU registers allow access to six segments at a time
- <u>Selector</u> is used to index a segment descriptor table to obtain an 8-byte segment descriptor entry. Base address and offset are added to get a <u>32-bit linear address</u>, which is partitioned into p1, p2, and d for supporting 2-level paging.

The following figure shows the hardware support needed for this translation.
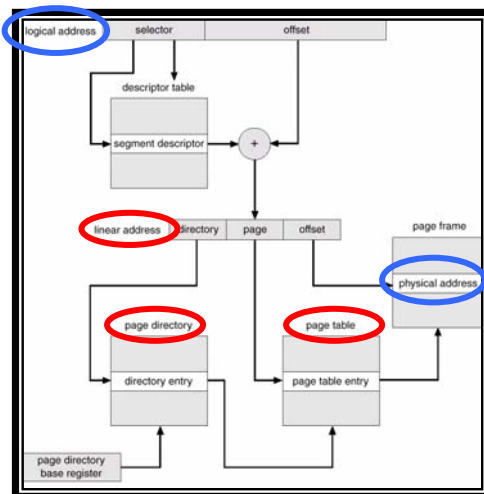
Intel 80386 address translation in protected mode

Virtual Memory Basic Concept

An examination of real programs shows that in many cases the existence of the entire program in memory is not necessary:

- Programs often have code to handle unusual error conditions. Since these errors seldom occur in practice, this code is almost never executed.
- Arrays, lists and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements even though it is seldom larger than 10 by 10 elements.
- Certain options of a program may be used rarely.

Even in cases where the entire program is needed, it may not be all needed at the same time. The ability to execute a program that is only partially in memory confers many benefits.

- A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large virtual address space simplifying the programming task.
- Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput with no increase in response time or turnaround time.
- Less I/O would be needed to load or swap each user program into memory, so each user program would run faster.

Thus running a program that is not entirely in memory would benefit both the system and the user.

Virtual Memory is the separation of user logical memory from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available. Virtual memory makes the task of programming easier because the programmer need not worry about the amount of physical memory, or about what code can be placed in overlays; she can concentrate instead on the problem to be programmed.

In addition to separating logical memory from physical memory, virtual memory also allows files and memory to be shared by several different processes through page sharing. The sharing of pages further allows performance improvements during process creation. Virtual memory is commonly implemented as demand paging. It can also be implemented in a segmentation system. One benefit of virtual memory is efficient process creation. Yet another is the concept of memory mapped files. We will discuss these topics in subsequent lectures.
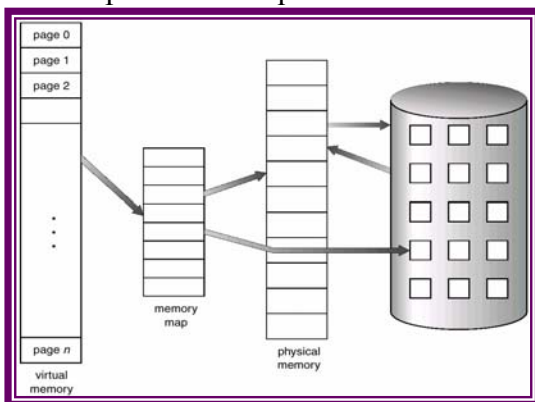

Mapping of logical memory onto physical memory under paging

Demand Paging

A demand paging system is similar to a paging system with swapping. Processes reside on secondary memory (which is usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however we use a lazy swapper. A lazy swapper never swaps a page into memory unless that page will be needed. Since we are now viewing a process as a sequence of pages rather than as one large contiguous address space, use of swap is technically incorrect. A swapper manipulates entire processes, whereas a pager is concerned with the individual pages of a process. Thus the term pager is used in connection with demand paging.

Basic Concepts

When a process is to be swapped in, the paging software guesses which pages would be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.
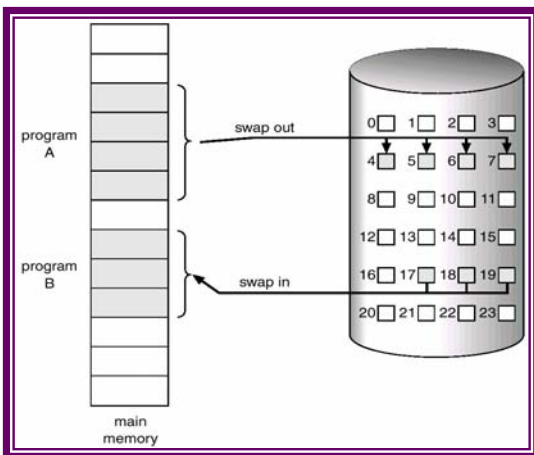
Swapping in and out of pages

With this scheme, we need some form of hardware support to distinguish which pages are in memory and which are on disk. The valid-invalid bit scheme described in previous lectures can be used. This time however when the bit is set to valid, this value indicates that the associated page is both legal and in memory. If the bit is set to invalid this value indicates that the page either is invalid or valid but currently on the disk. The page table entry for a page that is brought into memory is set as usual but the page table entry for a page that is currently not in memory is simply marked invalid or contains the address of the page on disk.

Notice that marking a page invalid will have no effect if the process never attempts to access that page. Hence if we guess right and page in all and only those pages that are actually needed, the process will run exactly as though we had brought in all pages. Wile the process executes and accesses pages that are memory resident, execution proceeds normally.
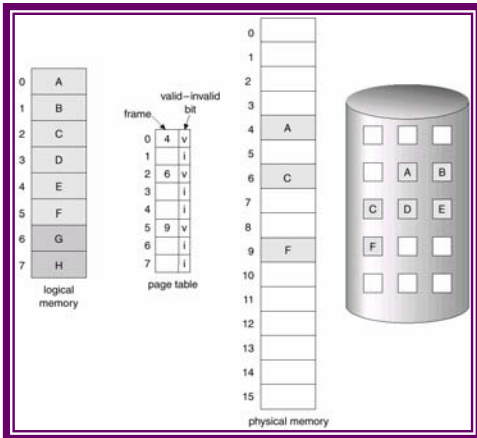
Protection under paging

Page Fault

But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a page fault trap. The paging hardware in translating the address through the page table will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory (in an attempt to minimize disk transfer overhead and memory requirements) rather than an invalid address error as a result of an attempt to use an illegal memory address. The procedure for handling a page fault is straightforward:

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was valid or invalid memory access.
2. If the reference was invalid we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example)
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been in memory.
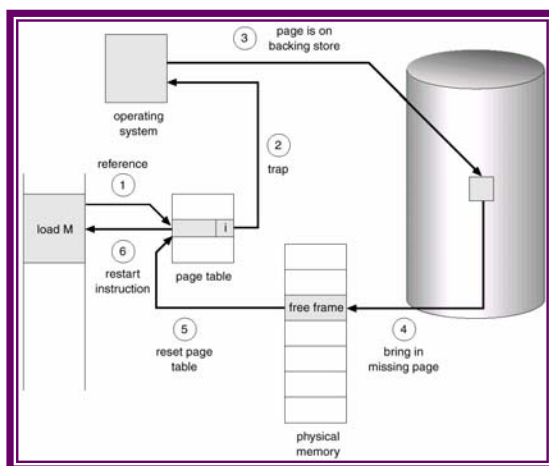
Steps needed for servicing a page fault

Since we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we can restart the process in exactly the same place and state except that the desired page is now in memory and is accessible. In this way we are able to execute a process even though portions of it are not yet in memory. When the process tries to access locations that are not in memory, the hardware traps the operating system (page fault). The operating system reads the desired into memory and restarts the process as though the page had always been in memory.

In the extreme case, we could start executing a process with no pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a non memory resident page, the process immediately faults for the page. After this page is brought into memory, the process continues to execute faulting as necessary until every page that it needs is in memory. At that point, it can execute with no more faults. This scheme is called pure demand paging: never bring a page into memory until it is required.

The hardware needed to support demand paging is the same as the hardware for paging and swapping:

- Page table: This table has the ability to mark an entry invalid through a valid-invalid bit or special value of protection bits.
- Secondary memory: This memory holds those pages that are not present in main memory. The secondary memory is usually a high speed disk. It is known as the swap device, and the section of disk used for this purpose is called the swap space.

In addition to this hardware, additional architectural constraints must be imposed. A crucial one is the need to be able to restart any instruction after a page fault. In most cases this is easy to meet, a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again, and then fetch the operand. A similar problem occurs in machines that use special addressing modes, including auto increment and auto decrement modes. These addressing modes use a register as a pointer and automatically increment or decrement the register. Auto decrement automatically decrements the register before using its contents as the operand address; auto increment increments the register after using its contents. Thus the instruction

MOV (R2) +, -(R3)

Copies the contents of the location pointed to by register2 into that pointed to by register3. Now consider what will happen if we get a fault when trying to store into the location pointed to by register3. To restart the instruction we must reset the two registers to the values they had before we started the execution of the instruction.
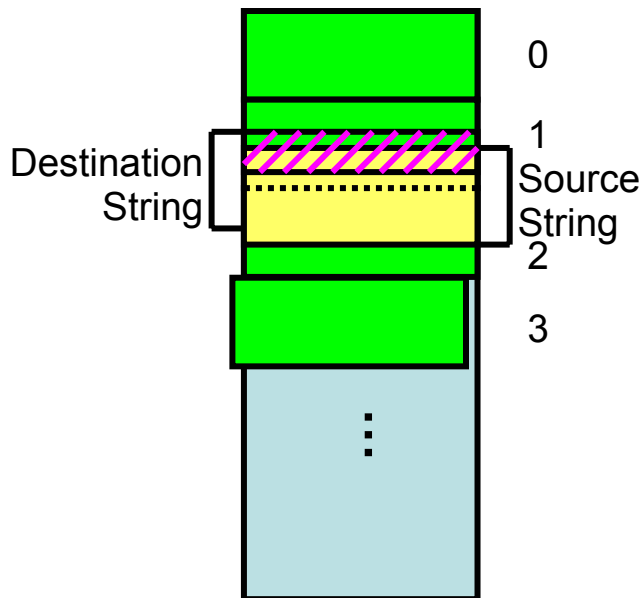
Execution of a block (string) move instruction causing part of the source to be overwritten before a page fault occurs

Another problem occurs during the execution of a block (string) move instruction. If either source or destination straddles a page boundary a page fault might occur after the move is partially done. In addition if the source and destination blocks overlap the source block may have been modified in which case we cannot simply restart the instruction, as shown in the diagram on the previous page.

Performance of demand paging
Demand paging can have a significant effect on the performance of a computer system. To see why, let us compute the effective access time for a demand paged memory. For most computer systems, the memory access time, denoted ma now ranges from 10 to 200 nanoseconds. As long as we have no page faults, the effective access time is equal to the memory access time. If, however a page fault occurs, we must first read the relevant page from disk, and then access the desired word.

Let p be the probability of a page fault ($0 \leq p \leq 1$). We would expect p to be close to zero, that is, there will be only a few page faults. The effective access time is then:

Effective access time = $(1-p) * ma + p *$ page fault time

To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

1. Trap to the operating system
2. Save the user registers and process states
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on disk
5. Issue a read from the disk to a free frame:
   a. Wait in a queue for this device until the read request is serviced
   b. Wait for the device seek and/or latency time
   c. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user (CU scheduling; optimal)

189

7. Interrupt from the disk (I/O completed)
8. Save the registers and process state for the other user (if step 6 is executed)
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show that the desired page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state and new page table

Not all these steps are necessary in every case. For example we are assuming that in step 6, the CPU is allocated to another process while the I/O occurs. This arrangement allows multiprogramming to maintain CPU utilization, but requires additional time to resume the page fault service routine when the I/O transfer is complete.

In any case we are faced with three major components of the page fault service time:

1. Service the page fault interrupt
2. Read in the page
3. Restart the process

The first and third tasks may be reduced, with careful coding, to several hundred instructions. These tasks may take from 1 to 100 microseconds each. The page switch time, on the other hand, will probably be close to 24 milliseconds. A typical hard disk has an average latency of 8 milliseconds, a seek of 15 milliseconds, and a transfer time of 1 millisecond. Thus, the total paging time would be close to 25 milliseconds, including hardware and software time. Remember that we are looking at only the device service time. If a queue of processes is waiting for the device we have to add device queuing time as we wait for the paging device to be free to service our request, increasing even more the time to swap.

If we take an average page fault service time of 25 milliseconds and a memory access time of 100 nanoseconds, then the effective access time in nanoseconds is

Effective access time $= (1-p) * (100) + p (25 \text{ milliseconds})$
$= (1-p) * 100 + p * 25,000,000$
$= 100 + 24,999,900 * p$

We see then that the effective access time is directly proportional to the page fault rate. If one access out of 1,000 causes a page fault, the effective access time is 25 microseconds. The computer would be slowed down by a factor of 250 because of demand paging! If we want less than 10 percent degradation, we need:

$110 > 100 + 25,000,000 * p$
$10 > 25,000,000 * p$
$p < 0.0000004$

That is, to keep the slowdown due to paging to a reasonable level, we can allow only less than one memory access out of 2,500,000 to page fault.

It is important to keep the slowdown due to paging to a reasonable level, we can allow only less than one memory access out of 2,500,000 to page fault.

It is important to keep the page fault rate low in a demand-paging system. Otherwise the effective access time increases, slowing process execution dramatically.

One additional aspect of demand paging is the handling and overall use of swap space. Disk I/O to swap space is generally faster than that to the file system. It is faster because swap space is allocated in much larger blocks, and file lookups and indirect allocation methods are not used. It is therefore possible for the system to gain better paging throughput by copying an entire file image into the swap space at process startup,

and then performing demand paging from the swap space. Another option is to demand pages from the from the file system initially, but to write the pages to swap space as they are replaced. This approach will ensure that only needed pages are ever read from the file system, but all subsequent paging is done from swap space.

Some systems attempt to limit the amount of swap space when binary files are used. Demand pages for such files are brought directly from the file system. However, when page replacement is called for, these pages can simply be overwritten and read in from the file system again if ever needed. Using this approach, the file system itself serves as the backing store. However swap space must still be used for pages not associated with a file; these pages include the stack and heap for a process. This technique is used in several systems including Solaris.

Operating Systems
Lecture No. 38

Reading Material
- Chapter 10 of the textbook
- Lecture 38 on Virtual TV

Summary
- Performance of Demand Paging
- Process Creation
- Memory Mapped Files

Performance of Demand Paging with Page Replacement
When there is no free frame available, page replacement is required, and we must select the pages to be replaced. This can be done via several replacement algorithms, and the major criterion in the selection of a particular algorithm is that we want to minimize the number of page faults. The victim page that is selected depends on the algorithm used, it might be the least recently used page, or the most frequently used etc depending on the algorithm.

Another Example
- Effective memory access is 100 ns
- Page fault overhead is 100 microseconds = 105 ns
- Page swap time is10 milliseconds = 107 ns
- 50% of the time the page to be replaced is "dirty"
- Restart overhead is 20 microseconds = 2 x 104 ns

Effective access time = 100 * (1-p) + (105 + 2 * 104 + 0.5 * 107 + 0.5 * 2 * 107) * p
                      = 100 * (1-p) + 15,120,000 * p

What is a Good Page Fault Rate?
For the previous example suppose p is 1%, then EAT is
            = 100 * (1-p) + 15,120,000 * p
            = 151299 ns
Thus a slowdown of 151299 / 100 = 1513 occurs.
      For the luxury of virtual memory to cost only 20% overhead, we need
            120 > 100 * (1-p) + 15,120,000 * p
            120 > 100 -100 p + 15,120,000 p
            p < 0.00000132

      $\Rightarrow$ Less than one page fault for every 755995 memory accesses!

Process Creation and Virtual Memory

Paging and virtual memory provide other benefits during process creation, such as copy on write and memory mapped files.

Copy on Write `fork()`

Demand paging is used when reading a file from disk into memory and such files may include binary executables. However, process creation using fork() may bypass initially the need for demand paging by using a technique similar to page sharing. This technique provides for rapid process creation and minimizes the number of new pages that must be allocated to newly created processes.

Recall the fork() system call creates a child process as a duplicate of its parent. Traditionally fork() worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent. However, considering that many child processes invoke the exec() system call immediately after creation, the copying of the parent's address space may be unnecessary. Alternatively we can use a technique known as copy on write. This works by allowing the parent and child processes to initially share the same pages. These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created. For example assume a child process attempts to modify a page containing portions of the stack; the operating system recognizes this as a copy-on-write page. The operating system will then create a copy of this page mapping it to the address space of the child process. Therefore the child page will modify its copied page, and not the page belonging to the parent process. Using the copy-on-write technique it is obvious that only the pages that are modified by either process are copied; all non modified pages may be shared by the parent and the child processes. Note that only pages that may be modified are marked as copy-on-write. Pages that cannot be modified (i.e. pages containing executable code) may be shared by the parent and the child. Copy-on-write is a common technique used by several operating systems such as Linux, Solaris 2 and Windows 2000.

When it is determined a page is gong to be duplicated using copy-on-write it is important to note where the free page will be allocated from. Many operating systems provide a pool of free pages for such requests. These free pages are typically allocated when the stack or heap for a process must expand or for managing copy-on-write pages. Operating systems typically allocate these pages using a technique known as zero-fill-on-demand. Zero-fill-on-demand pages have been zeroed out before allocating, thus deleting the previous contents on the page. With copy-on-write the page being copied will be copied to a zero-filled page. Pages allocated for the stack or heap are similarly assigned zero-filled pages.

`vfork()`

Several versions of UNIX provide a variation of the fork() system call—vfork() (for virtual memory fork). vfork() operates differently than fork() with copy on write. With vfork() the parent process is suspended and the child process uses the address space of the parent. Because vfork() does not use copy-on-write, if the child process changes any pages of the parent's address space, the altered pages will be visible to the parent once it resumes. Therefore, vfork() must be used with caution, ensuring that the child process does not modify the address space of the parent. vfork() is intended to be used when the

child process calls exec() immediately after creation. Because no copying of pages takes place, vfork() is an extremely efficient method of process creation and is sometimes used to implement UNIX command-line shell interfaces.

Linux Implementation
In Linux, shared pages are marked read-only after fork(). If either process tries to modify a shared page, a page fault occurs and the page is copied. The other process (who later faults on write) discovers it is the only owner; so no copying takes place. In other words, Linux implementation of fork() is based on the "copy-on-write" semantics.

Memory Mapped files

Consider a sequential read of a file on disk using the standard system calls open(), read(), write(). Every time the file is accessed requires a system call and disk access. Alternatively we can use the virtual memory techniques discussed so far to treat file I/O as routine memory accesses. This approach is known as memory mapping a file, allowing a part of the virtual address space to be logically associated with a file. Memory mapping a file is possible by mapping a disk block to a page (or pages) in memory. Initial access to the file proceeds using ordinary demand paging resulting in a page fault. However, a page sized portion of the file is read from the file system into a physical page. Subsequent reads and writes to the file are handled as routine memory accesses, thereby simplifying file access and usage by allowing file manipulation through memory rather than the overhead of using the read() and write() system calls. Note that writes to the file mapped in memory may not be immediate writes to the file on disk. Some systems may choose to update the physical file when the operating system periodically checks if the page in memory mapping the file has been modified. Closing the file results in all the memory mapped data being written back to disk and removed from the virtual memory of the process. The concept of memory mapped files is shown pictorially in the following diagram.
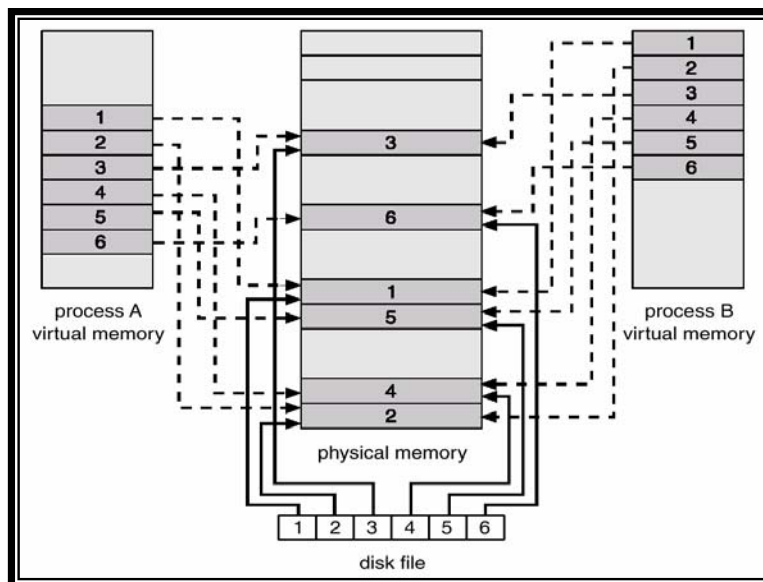
Memory mapped files

Memory-Mapped Files in Solaris 2

Some operating systems provide memory mapping only through a specific system call and treat all other file I/O using the standard system calls. However, some systems choose to memory map a file regardless of whether a file was specified as a memory map or not. For example: Solaris 2 treats all file I/O as memory mapped, allowing file access to take place in memory, whether a file has been specified as memory mapped using mmap() system call or not.

Multiple processes may be allowed to map the same file into the virtual memory of each to allow sharing of data. Writes by any of the processes modify the data in virtual memory and can be seen by all others that map the same section of the file. Given our knowledge of virtual memory it should be clear how the sharing of memory mapped sections of memory is implemented. The virtual memory map of each sharing process points to the same page of physical memory – the page that holds a copy of the disk block. This memory mapping is illustrated as:

The memory mapping system calls can only support copy-on-write functionality allowing processes to share a file in read-only mode, but to have their own copies of any data they modify. So that access to the shared data is coordinated, the processes involved might use one of the mechanisms for achieving mutual exclusion.

`mmap()` System Call

In a UNIX system, mmap() system call can be used to request the operating system to memory map an opened file. The following code snippets show "normal" way of doing file I/O and file I/O with memory mapped files.

"Normal" File I/O

```
fildes = open(...);
lseek(...);
read(fildes, buf, len);
/* use data in buf */
```

File I/O with mmap()

```
fildes = open(...)
address = mmap((caddr_t) 0, len,(PROT_READ | PROT_WRITE), MAP_PRIVATE, fildes, offset);
/* use data at address */
```

Operating Systems
Lecture No. 39

Reading Material
- Chapter 10 of the textbook
- Lecture 39 on Virtual TV

Summary
- Page replacement (basic concept and replacement algorithms)
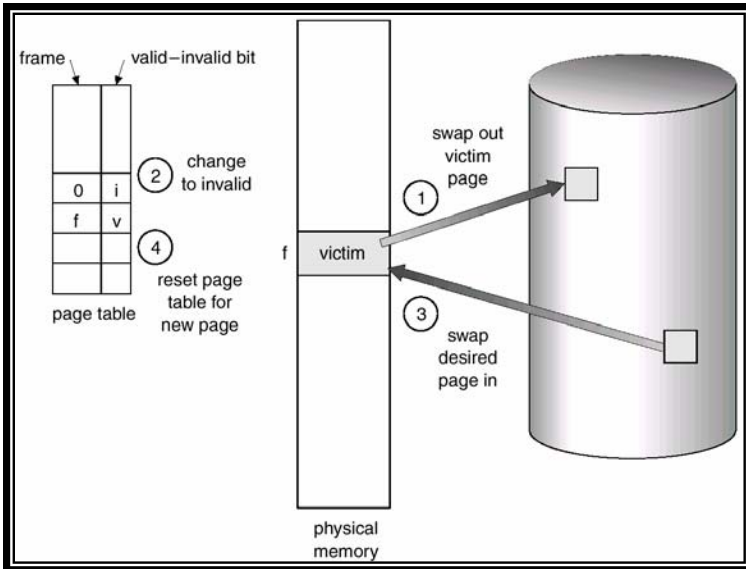
Page replacement
While a user process is executing, a page fault occurs. The hardware traps to the operating system, which checks its internal tables to see that this page is a genuine one rather than an illegal memory access. The operating system determines where the desired page is residing on the disk, but then finds that there are no free frames on the free frame list: All memory is in use.

The operating system has several options at his point. It could terminate the user process. However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput. Users' should not be aware that their processes are running on a paged system – paging should be logically transparent to the user. So this option is not the best choice. The operating system could swap out a process, but that would reduce the level of multiprogramming. So we explore page replacement. This means that if no free frame is available on a page fault, we replace a page in memory to load the desired page. The page-fault service routine is modified to include page replacement. We can free a frame by writing its contents to swap space, and changing the page table to indicate that the page is no longer in memory. The modified page fault service routine is:

1. Find the location of the desired page on the disk
2. Find a free frame
   a) If there is a free frame use it.
   b) If there is no free frame, use a page replacement algorithm to select a victim frame.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Restart the user process.

The following diagram shows theses steps pictorially.

Steps needed for page replacement

We can reduce overhead by using a *modify* bit (or *dirty* bit). Each page or frame may have a modify bit associated with it in hardware. The modify bit is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified. When we select a page for replacement we examine it's modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case we must write that page to the disk. If the modify bit is not set however, the page has not been modified since it was read into memory, and hence we can avoid writing that page to disk. In the following figure we show two processes with four pages each, main memory having eight frames, with two used for resident part of operating system (leaving six frames for user processes). Both processes have three of their pages in memory and therefore there is no free frame. When the upper process (user 1) tries to access its fourth page (page number 3), a page fault is caused and page replacement is needed.
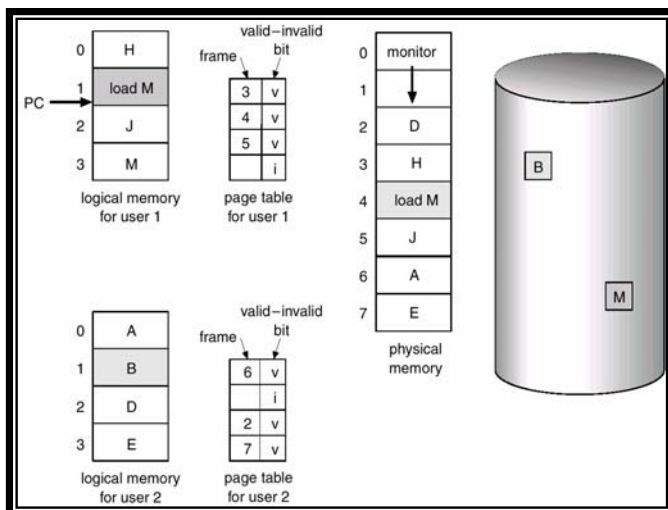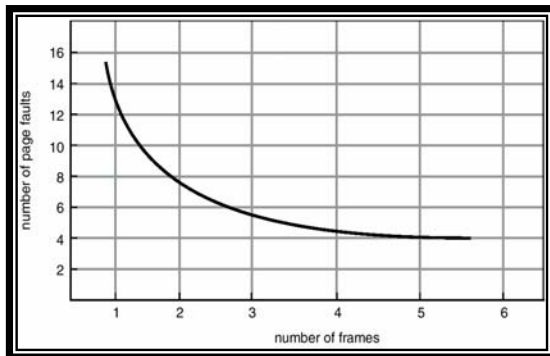

Page fault and page replacement

Page Replacement Algorithms

In general we want a page replacement algorithm with the lowest page-fault rate. We evaluate an algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.

To determine the number of page faults for a particular reference string and page replacement algorithm, we also need to know the number of page frames available. Obviously as the number of frames available increases, the number of page faults decreases.

Expected relationship between number of free frames allocated to a process and the number of page faults caused by it

FIFO Page Replacement

The simplest page-replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory we insert t at the tail of the queue.

Consider the following example, in which the number of frames allocated is 4, and the reference string is 1, 2, 3, 4, 5, 1, 6, 7, 8, 7, 8, 9, 5, 4, 5, 4, 4. The number of page faults caused by the process is nine, as shown below.

1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 5, 4, 5, 4, 4

Example for the FIFO page replacement algorithm

The problem with this algorithm is that it suffers from Belady's anomaly: For some page replacement algorithms the page fault rate may increase as the number of allocated

frames increases, whereas we would expect that giving more memory to a process would improve its performance.

Optimal Algorithm

An optimal page-replacement algorithm has the lowest page fault rate of all algorithms, and will never suffer from the Belay's algorithm. This algorithm is simply to replace the page that will not be used for the longest period of time. Use of this algorithm guarantees the lowest possible page-fault rate for a fixed number of frames. In case of the following example (which uses the same replacement string as the example for the FIFO algorithm), the number of page faults caused by the process is seven.

1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 5, 4, 5, 4, 4

1	1	6	6	8	9	9
2	5	5	5	5	5	5
3	3	3	7	7	7	7
4	4	4	4	4	4	4

Example for the optimal page replacement algorithm

Unfortunately this algorithm is difficult to implement because it requires future knowledge of the reference string. As a result this algorithm is used mainly for comparison.

LRU Page Replacement

If we use the recent past as an approximation of the near future, then we will replace the page that has not been used for the longest period of time. This approach is the least recently used algorithms. The following example illustrates the working of LRU algorithm.
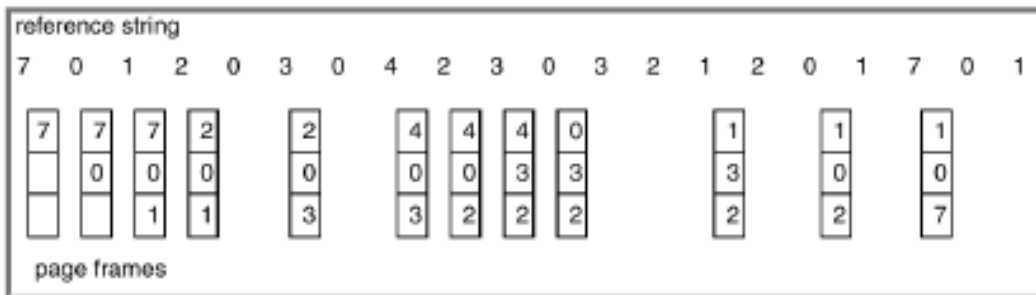
Example for the LRU page replacement algorithm

Here is another example, which uses the same reference string as used in the examples for the FIFO and optimal replacement algorithms. The number of page faults in this case is nine.

1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 5, 4, 5, 4, 4

Another example for the LRU algorithm

An LRU page replacement may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

Counter-based Implementation of LRU
In the simplest case we associate with each page table entry a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page entry for that page. In that way we always have the time of the last reference to each page. We replace the page that has the smallest time value. This scheme requires a search of the page table to find the LRU page and a write to memory for each memory access. The times must also be maintained when page tables are changed. Overflow of the clock must be considered.

Stack-based Implementation of LRU
Another approach to implementing the LRU algorithm is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on top. In this way, the top of the stack is always the most recently used page and the bottom is the LRU page. Because entities must be removed from the middle of the stack, it is best implementing by a doubly linked list with a head and tail pointer. Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement the tail pointer points to the bottom of the stack which is the LRU page. The following diagram shows the working of stack-based implementation of the LRU algorithm.
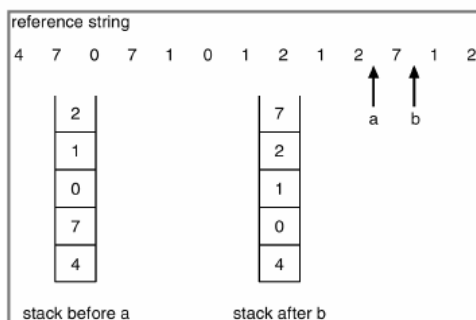
Stack based implementation of the LRU page replacement algorithm

Operating Systems
Lecture No. 40

Reading Material
- Chapter 10 of the textbook
- Lecture 40 on Virtual TV

Summary
- Belady's Anomaly
- Page Replacement Algorithms
  - Least Frequently Used (LFU)
  - Most Frequently Used (MFU)
  - Page Buffering Algorithm
- Allocation of Frames
- Minimum Number of Frames
- Thrashing

Belady's Anomaly
Consider the following example of the FIFO algorithm.
- Number of frames allocated = 3
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Number of page faults = 9

1	1	1	4	4	4	5	5	5
	2	2	2	1	1	1	3	3
		3	3	3	2	2	2	4

Now an intuitive idea is that if we increase the number of frames allocated to 4 from 3, the page faults should decrease, but the following example demonstrates otherwise.
- Number of frames allocated = 4
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Number of page faults = 10

1	1	1	1	5	5	5	5	4	4
	2	2	2	2	1	1	1	1	5
		3	3	3	3	2	2	2	2
			4	4	4	4	3	3	3

This is due to the **Belady's Anomaly** which states that "For some page replacement algorithms, the page fault rate may increase as the number of allocated frames increases."
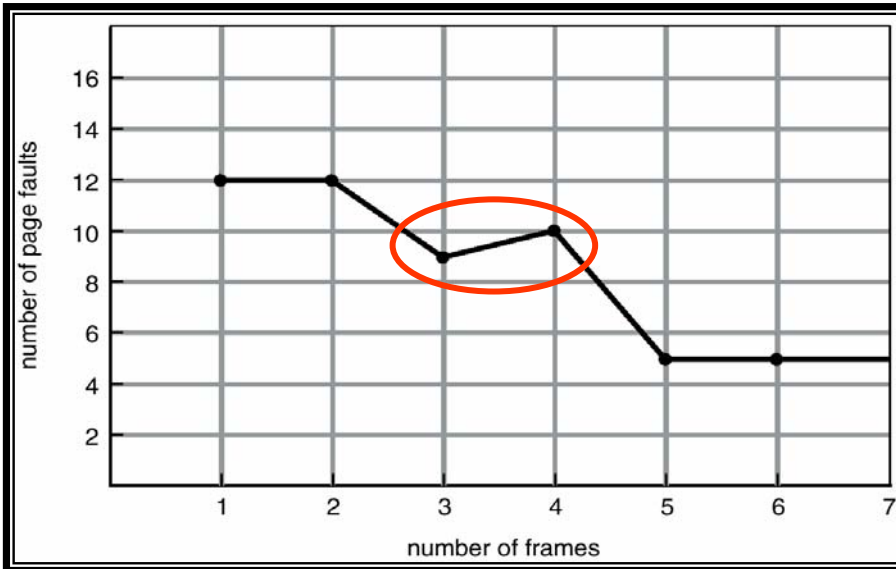
Belady's anomaly

Stack Replacement Algorithms

These are a class of page replacement algorithms with the following property:

Set of pages in the main memory with n frames is a subset of the set of pages in memory with n+1 frames.

These algorithms do not suffer from Belady's Anomaly. An example is the LRU algorithm.

Consider the following example which shows that LRU does not suffer from Belady's anomaly for the given reference string.

- Number of frames allocated = 3
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Number of page faults = 10

1	1	1	4	4	4	5	3	3	3
	2	2	2	1	1	1	1	4	4
		3	3	3	2	2	2	2	5

- Number of frames allocated = 4
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Number of page faults = 8

1	1	1	1	1	1	1	5
	2	2	2	2	2	2	2
		3	3	5	5	4	4
			4	4	3	3	3

LRU Approximation Algorithm

Few computer systems provide sufficient hardware support for true LRU page replacement. Some systems provide no hardware support and other page replacement algorithms must be used. Many systems provide some help however, in the form of a reference bit. The reference bit for a page is set by the hardware whenever that page is referenced. Reference bits are associated with each entry in the page table.

Initially all bits are cleared by the operating system. As a user process executes the bit associated with each page referenced is set to 1 by the hardware. After some time we can determine which pages have been used and which have not been used by examining the reference bits. We do not know the order of use however, but we know which pages were used and which were not used.

Least frequently used algorithm

This algorithm is based on the locality of reference concept— the least frequently used page is not in the current locality. LFU requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again. Since it was used heavily it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average user count.

Most Frequently Used

 The MFU page replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used; it will be in the **locality** that has just started.

Page Buffering Algorithm

The OS may keep a pool of free frames. When a page fault occurs a victim page is chosen as before. However the desired page is read into a free frame from the pool before the victim is written out. This allows the process to restart as soon as possible, without waiting for the victim to be written out. When the victim is later written out, its frame is added to the free frame pool. Thus a process in need can be given a frame quickly and while victims are selected, free frames are added to the pool in the background

An expansion of this idea is to maintain a list of modified pages. Whenever the paging device is idle, a modified page is selected and is written to disk. Its modify bit is then reset. This scheme increases the probability that a page will be clean when it is selected for replacement and will not need to be written out.

Another modification is to keep a pool of free frames, but to remember which page was in which frame. Since the frame contents are not modified when a frame is written to disk, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused. No I/O is needed in this case. When a page fault occurs we check whether the desired page is in the free-frame pool. If it is not we must select a free frame and read into it. This method is used together with FIFO replacement in the VAX/VMS operating system.

Local vs Global Replacement

If process P generates a page fault, page can be selected in two ways:

- Select for replacement one of its frames.
- Select for replacement a frame from a process with lower priority number.

Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame belongs to some other process; one process can take a frame from another. Local replacement requires that each process select from only its allocated frames.

Consider an allocation scheme where we allow high priority processes to select frames from low priority processes for replacement. A process can select a replacement from among its own frames or the frames of any lower priority process. This approach allows a high priority process to increase its frame allocation at the expense of the low priority process.

Allocation of frames

Each process needs a minimum number of frames so that its execution may be guaranteed on a given machine. Let's consider the MOV X,Y instruction. The instruction is 6 bytes long (16-bit offsets) and might span 2 pages. Also, two pages to handle <u>source</u> and two pages are required to handle destination (assuming 16-bit source and destination).
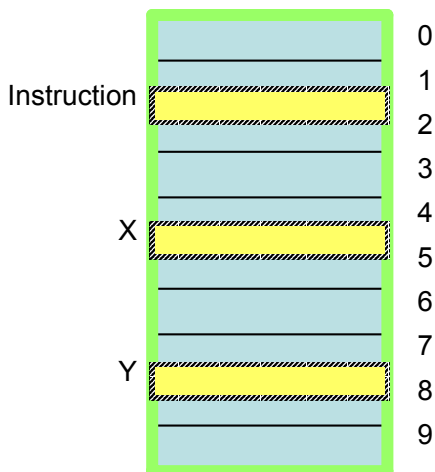
Minimum frames required to guarantee execution of the MOV X,Y instruction

There are three major allocation schemes:

- **Fixed allocation**
  In this scheme free frames are equally divided among processes
- **Proportional Allocation**
  Number of frames allocated to a process is proportional to its size in this scheme.

- **Priority allocation**
  Priority-based proportional allocation

Here is an example of frame allocation:

Number of free frames = 64
Number of processes = 3
**Process sizes**: P1 = 10 pages; P2 = 40 pages; P3 = 127 pages

- **Fixed allocation**
  64/3 = 21 frames per process and one put in the free frames list

- **Proportional Allocation**
  - s_i = Size of process P_i
  - $S = \sum s_i$
  - m = Number of free frames
  - a_i = Allocation for P_i = $(s_i / S) * m$
    - $a_1 = (10 / 177) * 64 = 3$ frames
    - $a_2 = (40 / 177) * 64 = 14$ frames
    - $a_3 = (127 / 177) * 64 = 45$ frames
  - Two free frames are put in the list of free frames

Thrashing

If a process does not have "enough" pages, the page-fault rate is very high. This leads to low CPU utilization. The operating system thinks that it needs to increase the degree of multiprogramming, because it monitors CPU utilization and find it to be decreasing due to page faults. Thus another process is added to the system and hence thrashing occurs and causes throughput to plunge.

A process is **thrashing** if it is spending more time paging (i.e., swapping pages in and out) than executing. Thrashing results in severe performance problems:
- Low CPU utilization
- High disk utilization
- Low utilization of other I/O devices
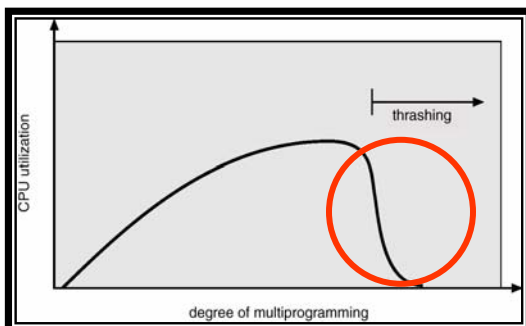
Thrashing

The figure shows that as the degree of multiprogramming increases CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased further, thrashing sets in and CPU utilization drops sharply. At this point we must decrease the degree of multiprogramming. We can limit

the effects of thrashing by using a local replacement scheme. With local replacement if one process starts thrashing it cannot steal frames from another process and cause the latter to thrash also. Pages are replaced with regard to the process of which they are a part. Hence local page replacement prevents thrashing to spread among several processes. However if processes are thrashing, they will be in the queue for the paging device most of the time. The average service time for a page fault will increase and effective access time will increase even for a process that is not thrashing.

Operating Systems
Lecture No. 41

Reading Material
- Chapter 10 of the textbook
- Lecture 41 on Virtual TV

Summary
- Thrashing
- The Working Set Model
- Page Fault Frequency Model
- Other Considerations
  - Prepaging
  - Page size
  - Program structure
- Examples of Virtual Memory Systems

Thrashing

If a process does not have enough frames, it will quickly page fault. At this point, if a free frame is not available, one of its pages must be replaced so that the desired page can be loaded into the newly vacated frame. However since all its pages are in active use, the replaced page will be needed right away. Consequently it quickly faults again and again. The process continues to fault, replacing pages for which it then faults and brings back in right away. This high paging activity is called **thrashing**. In this case, *only one process is thrashing*. A process is thrashing if it is spending more time paging than executing.

Thrashing results on severe performance problems. The operating system monitors CPU utilization and, if CPU utilization is too low, the operating system increases the degree of multiprogramming by introducing one or more new processes to the system. This decreases the number of frames allocated to each process currently in the system, causing more page faults and further decreasing the CPU utilization. This causes the operating system to introduce more processes into the system. As a result CPU utilization drops even further and the CPU scheduler tries to increase the degree of multiprogramming even more. Thrashing has occurred and system throughput plunges. The page fault rate increases tremendously. As a result the effective memory access time increases. Along with low CPU utilization, there is high disk utilization. There is low utilization of other I/O devices. No work is getting done, because the processes are spending all their time paging and the system spend most of its time servicing page fault. Now *the whole system is thrashing*—the CPU utilization plunges to almost zero, the paging disk utilization becomes very high, and utilization of other I/O devices becomes very low.

If a global page replacement algorithm is used, it replaces pages with no regard to the process to which they belong. Now suppose that a process enters a phase in its execution and needs more frames. It starts faulting and taking frames away from other processes. These processes need those pages however and so they also fault taking frames away

from other processes. These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.

The following graph shows the relationship between the degree of multiprogramming and CPU utilization.
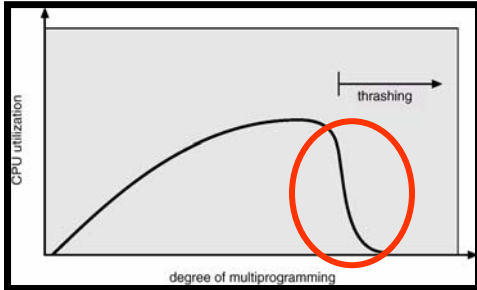
Relationship between the degree of multiprogramming and CPU utilization

Thus in order to stop thrashing, the degree of multiprogramming needs to be reduced. The effects of thrashing can be reduced by using a local page replacement. With local replacement if one process starts thrashing it cannot steal frames from another process and cause the latter to thrash also. Pages are replaced with regard to the process if which they are a part. However, if processes are thrashing they will be in the queue for the paging device most of the time. The average service time for a page fault will increase due to the longer average queue for the paging device. Thus the effective access time will increase even for a process that is not thrashing, since a thrashing process is consuming more resources

Locality of Reference

The locality model states that as a process executes it moves from locality to locality. A locality is a set of pages that are actively used together. A program is generally composed of several different localities, which may overlap. The following diagram shows execution trace of a process, showing localities of references during the execution of the process.
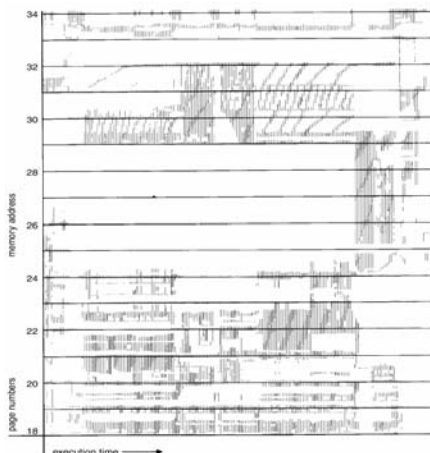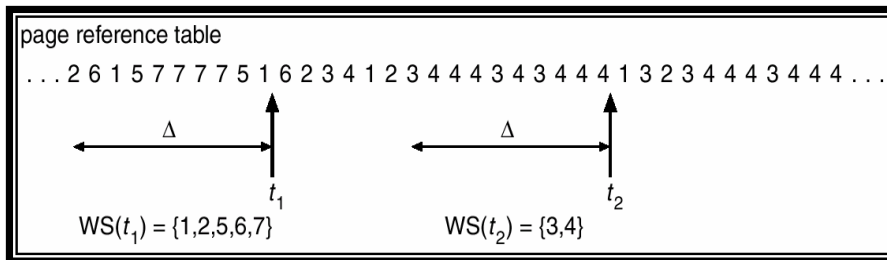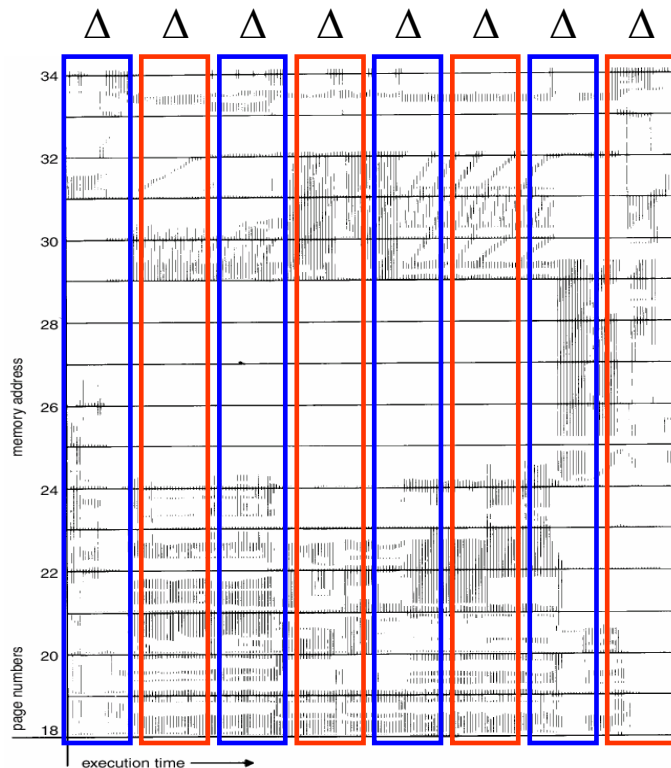

Process execution and localities of reference

Working Set Model

The working set model is based on the assumption of locality. This model uses a parameter Δ to define the working set window. The idea is to examine the most recent Δ page references. The set of pages in the most recent Δ page references is called the working set. If a page is in active use it will be in the working set. If it no longer being used it will drop from the working set Δ time units after its last reference. Thus the working set is an approximation of the program's locality.

In the following example, we use a value of Δ to be 10 and identify two localities of reference, one having five pages and the other having two pages.

We now identify various localities in the process execution trance given in the previous section. Here are the first two and last localities are: L1 = {18-26, 31-34}, L2 = {18-23, 29-31, 34}, and Last = {18-20, 24-34}. Note that in the last locality, pages 18-20 are referenced right in the beginning only and are effectively out of the locality.

Process execution trace and localities of reference

The accuracy of the working set model depends on the selection of Δ. If Δ is too small, it will not encompass the entire locality; if Δ is too large, it may overlap several localities. In the extreme if Δ is infinite, the working set is the set of pages touched during the process execution. The most important property of the working set is its size. If we compute the working set size, WSS_i for each process in the system we can consider

$$D = \Sigma \, WSS_i$$

where, D is the total demand for frames. Each process is actively using the pages in its working set. Thus, process *i* needs WSS_i frames. If the total demand is greater than the total number of frames (D > m), thrashing will occur, because some processes will not have enough frames.

Use of the working set model is then simple, the operating system monitors the working set of each process and allocates to that working set enough frames to provide it with its working set size. If there are enough extra frames another process can be initiated. If the sum of the working set sizes increases, exceeding the total number of available frames, the operating system selects a process to suspend. The process' pages are written out and its frames are reallocated to other processes. The suspended process can be restarted later.

The difficulty with the working set model is to keep track of the working set. The working set window is a moving size window. At each memory reference a new reference appears at one end and the oldest reference drops off the other end. We can approximate the working set model with a fixed interval timer interrupt and a reference bit.

For example, assume Δ = 10,000 references and the timer interrupts every 5000 references. When we get a timer interrupt we copy and clear the reference bit values for each page. Thus if a page fault occurs we can examine the current reference bit and 2 in memory bits to determine whether a page was used within the last 10,000 to 15,000 references. If it was used at least one of these bits will be on, otherwise they will be off. Thus after Δ references, if one of the bits in memory = 1 then the page is in the working set. Note that this arrangement is not completely accurate because we cannot tell where within an interval of 5,000 a reference occurred. We can reduce the uncertainty by increasing the number of our history bits and the frequency of interrupts. However the cost to service these more frequent interrupts will be correspondingly higher.

Page Fault Frequency

Page fault frequency is another method to control thrashing. Since thrashing has a high page fault rate, we want to control the page fault frequency. When it is too high we know that the process needs more frames. Similarly if the page-fault rate is too low, then the process may have too many frames. The operating system keeps track of the upper and lower bounds on the page-fault rates of processes. If the page-fault rate falls below the lower limit, the process loses frames. If page-fault rate goes above the upper limit, process gains frames. Thus we directly measure and control the page fault rate to prevent thrashing. The following diagram shows the working of this scheme.
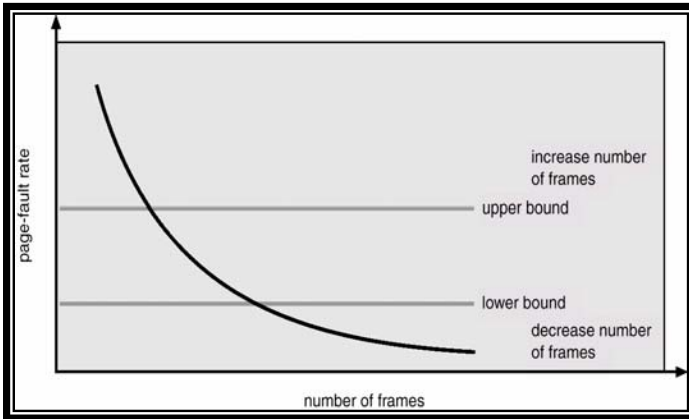
Controlling thrashing with page fault frequency

Other considerations

Many other things can be done to help control thrashing. We discuss some of the important ones in this section.

Pre-paging

An obvious property of a pure demand paging system is the large number of page faults that occur when a process is started. This situation is the result of trying to get the initial locality into memory. Pre-paging is an attempt to prevent this high level of initial paging. The strategy is to bring into memory at one time all the pages that will be needed.

Pre-paging may be an advantage in some cases. The question is simply whether the cost of using pre-paging is less than the cost of the servicing the corresponding page faults.

Page Size

How do we select a page size? One concern is the size of the page table. For a given virtual memory space, decreasing the page size increases the number of pages and hence the size of the page table. Because each active process must have its own copy of the page table, a large page size is desirable.

On the other hand, memory is better utilized with smaller pages. If a process is allocated memory starting at location 00000, and continuing till it has as much as it needs, it probably will not end exactly on a page boundary. Thus, a part of the final page must be allocated. This causes internal fragmentation and to minimize this, we need a small page size.

Another problem is the time required to read or write a page. I/O time is composed of seek, latency and transfer times. Transfer time is proportional to the amount transferred, and this argues for a small page size. However, latency and seek times usually dwarf transfer times, thus a desire to minimize I/O times argues for a larger page size. I/O overhead is also reduced with small page size because locality improves. This is because a smaller page size allows each page to match program locality more accurately.

Some factors (internal fragmentation, locality) argue for a small page size, whereas others (table size, I/O time) argue for a large page size. There is no best answer. However the historical trend is towards larger pages.

Program Structure
Demand paging is designed to be transparent to the user program. However, in some
cases system performance can be improved if the programmer has an awareness of the
underlying demand paging and execution environment of the language used in the
program. We illustrate this with an example, in which we initialize a two dimensional
array (i.e., a matrix).

Consider the following program structure in the C programming language. Also note
that arrays are stored in <u>row-major order</u> in C (i.e., matrix is stored in the main memory
row by row), and page size is such that each row is stored on one page.

```
Program 1
        int A[1024][1024];

        for (j = 0; j < 1024; j++)
            for (i = 0; i < 1024; i++)
                A[i,j] = 0;
```

Since this code snippet initializes the matrix column by column, it causes 1024 page
faults while initializing one column. This means that execution of the code causes **1024 x
1024** page faults.

Now consider the following program structure.

```
Program 1
        int A[1024][1024];

        for (i = 0; i< 1024; i++)
            for (j= 0; j < 1024; j++)
                A[i,j] = 0;
```

In this case, matrix is accessed row by row, causing 1 page fault per row. This means that
execution of the code causes **1024** page faults.

Example Systems
1. A demand paging system with the following utilizations:

   CPU   = 20%
   Paging disk   = 97.7%
   Other I/O devices = 5%

   Which of the following will improve CPU utilization?
   - Install a faster CPU
   - Increase degree of multiprogramming
   - Decrease degree of multiprogramming
   - Install more main memory

Clearly, the system is thrashing, so the first two are not going to help and the last two will help. Think about the reasons of this answer.

2. Which of the following programming techniques and structures are "good" for a demand paged environment? Which are bad? Explain your answer.
    - Stack
    - Hash table
    - Sequential search
    - Binary search
    - Indirection
    - Vector operations

You should try to answer this question on your own. Focus on how the given data structures and techniques access data. Sequential access means "good" for demand paging (because it causes less page faults) and non-sequential access means "bad" for demand paging environment.

Operating Systems
Lecture No. 42

Reading Material
- Chapter 11 of the textbook
- Lecture 42 on Virtual TV

Summary
- File Concept
- File Types
- File Operations
- Access Methods
- Directories
- Directory Operations
- Directory Structure

The File Concept
Computers can store information on several different storage media, such as magnetic disks, magnetic tapes and optical disks. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit (the file). Files are mapped by the OS onto physical devices. These storage devices are usually non-volatile, so the contents are persistent through power failures, etc. A file is a named collection of related information that is recorded on secondary storage. Data cannot be written to secondary storage unless they are within a file. Commonly, files represent programs (source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric or binary. In essence it is a contiguous logical address space.

File Structure
A file has certain defined structure characteristics according to its type. A few common types of file structures are:

None – file is a sequence of words, bytes

Simple record structure
> Lines
> Fixed length
> Variable length

Complex Structures
> Formatted document
> Relocatable load file

UNIX considers each file to be a sequence of bytes; no interpretation of these bytes is made by the OS. This scheme provides maximum flexibility but little support. Each application program must include its own code to interpret an input file into the

appropriate structure. However all operating systems must support at least one structure-that of an executable file-so that the system is able to load and run programs.

File Attributes
Every file has certain attributes, which vary from one OS to another, but typically consist of these:

Name: The symbolic file name is the only information kept in human-readable form

Type: This information is needed for those systems that support different types.

Location: This location is a pointer to a device and to the location of the file on that device.

Size: The current size of the file (in bytes, words or blocks) and possibly the maximum allowed size are included in this attribute.

Protection: Access control information determines who can do reading , writing, etc.

Owner

Time and date created: useful for security, protection and usage monitoring.

Time and date last updated: useful for security, protection and usage monitoring.

**Read/write pointer value**

Where are Attributes Stored?
File attributes are stored in the directory structure, as part of the **directory entry** for a file, e.g., in DOS, Windows, or in a separate data structure; in UNIX/Linux this structure is known as the **inode** for the file.

Directory Entry
A file is represented in a directory by its directory entry. Contents of a directory entry vary from system to system. For example, in DOS/Windows a directory entry consists of file name and its attributes. In UNIX/Linux, a directory entry consists of file name and inode number. Name can be up to 255 characters in BSD UNIX compliant systems. Inode number is used to access file's inode. The following diagrams show directory entries for DOS/Windows and UNIX/Linux systems.

File Name	Date	...	FAT

DOS/Windows

File Name	Inode Number
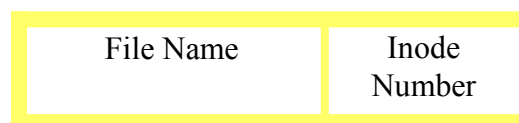
UNIX/Linux

File Operations
Various operations can be performed on files. Here are some of the commonly supported operations. In parentheses are written UNIX/Linux system calls for the corresponding operations.

- Create (`creat`) —two steps are necessary to create a file. First, space must be found for the file in the file system. Second, an entry for the new file must be made in the directory.
- Open (`open`) — The open operation takes a file name and searches the directory, copying the directory entry into the open-file table. The open system call can also accept access-mode information-read-only, read-write, etc. It typically returns a pointer to the entry in open-file table.
- Write (`write`) —To write to a file, we make a system call, specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the location of the file. The system must keep a *write* pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
- Read (`read`) — To read from a file we use a system call that specifies the name of the file, and where (in memory) the next block of the file should be put. The system needs top keep a read pointer to the location in the file where the next read is to take place. Once the read has taken place, the read pointer needs to be updated. A given process is usually only reading or writing to a file. The current pointer location is kept as a process **current-file-position pointer**. Both read and write use the same pointer
- Reposition within file (`lseek`) — A directory is searched for the appropriate entry and the current-file-position is set to a given value. This is often known as a file seek.
- Delete (`unlink`) — Search the directory for the named file, and then release the file space and erase the directory entry. File can be deleted using the unlink system call.
- Truncate (`creat`) — A user may want to erase the contents of the file but keep its attributes. This function allows all attributes to be unchanged except for file length., which is set to zero and file space is released. This can be achieved using creat with a special flag
- Close (`close`) — When a file is closed, the OS removes its entry in the open-file table.

File Types: Extensions
A common technique for implementing files is to include the type of the file as part of the file name. The name is split into two parts, a name and an extension, usually separated by a period character. In this way, the user and the OS can tell from the name alone, what the type of a file is.

The operating system uses the extension to indicate the type of the file and the type of operations that can be done on that file. In DOS/Windows only a file with .exe, .com, .bat extension can be executed.

The UNIX system uses a crude magic number stored at the beginning of some files to indicate roughly the type of the file-executable program, batch file/shell script, etc. Not all files have magic numbers, so system features cannot be based solely on this type of information. UNIX does allow file name extension hints, but these extensions are not enforced or depended on by the OS; they are mostly to aid users in determining the type of contents of the file. Extension can be used or ignored by a given application.

The following tables shows some of the commonly supported file extensions on different operating systems.

file type	usual extension	function
executable	exe, com, bin or none	read to run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rrf, doc	various word-processor formats
library	lib, a, so, dll, mpeg, mov, rm	libraries of routines for programmers
print or view	arc, zip, tar	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm	binary file containing audio or A/V information

Common file types

File Types in UNIX
UNIX does not support supports seven types of file:
- **Ordinary file**: used to store data on secondary storage device, e.g., a source program(in C), an executable program. Every file is a sequence of bytes.
- **Directory**: contains the names of other files and/or directories.
- **Block-special file**: correspond to block oriented devices such as a disk. They are used to access such hardware devices.
- **Character-special file**: correspond to character oriented devices, such as keyboard
- **Link file** (created with the `ln -s` command): is created by the system when a symbolic link is created to an existing file, allowing you to rename the existing file and share it without duplicating its contents without
- **FIFO** (created with the mkfifo or mknod commands or system calls): enable processes to communicate with each other. A FIFO(name pipe) is an area in the kernel that allows two processes to communicate with each other provided they are running on the same system , but the processes do not have to be related to each other.
- **Socket** (in BSD-compliant systems—socket): can be used by the process on the same computer or on different computers to communicate with each other.

File Access
Files store information that can be accessed in several ways:

Sequential Access

Information in the file is processed in order, one record after the other. A read operation reads the next potion of the file and automatically advances a file pointer which tracks the I/O location. Similarly, a write operation appends to the end of the file and advances to the end of the newly written material. Such a file can be rest to the beginning and on some systems; a program may be able to skip forward or backward, n records.
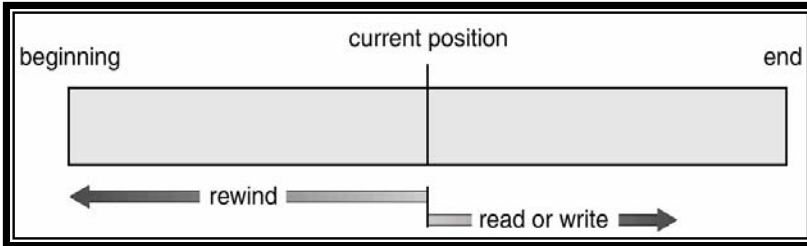

Sequential Access File

Direct Access

A file is made up of fixed length logical record that allow program to read and write records in no particular order. For the direct-access method, the file operations must be modified to include the block number as a parameter (read n (n = relative block number), write n for instance). An alternate approach is to retain read next and write next and to add an operation, *position file to n,* where n is the block number. The block number provided by the user to the OS is normally a *relative block number*, an index relative to the beginning of the file.

sequential access	implementation for direct access
reset	$cp = 0;$
read next	*read cp;* $cp = cp+1;$
write next	*write cp;* $cp = cp+1;$

Sequential Access on a Direct Access File

Directory Structure

It is a collection of directory entries. To manage all the data, first disks are split into one or more partitions. Each partition contains information about files within it. This information is kept within device directory or volume table of contents.

Directory

Files and Directories
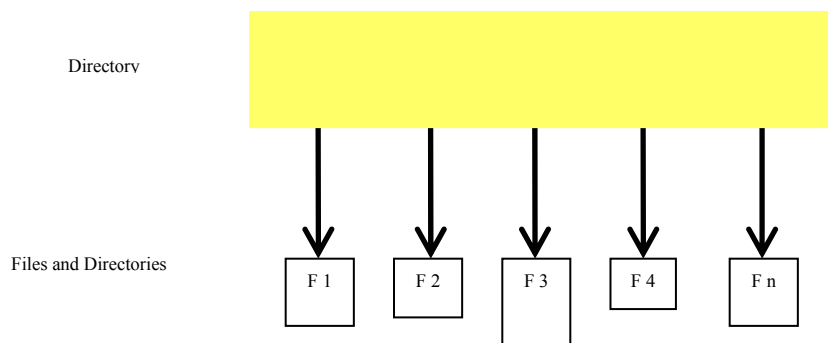
Directory Operations
The following directory operations are commonly supported in contemporary operating systems. Next to each operation are UNIX system calls or commands for the corresponding operation.

- Create — `mkdir`
- Open — `opendir`
- Read — `readdir`
- Rewind — `rewinddir`
- Close — `closedir`
- Delete — `rmdir`
- Change Directory — `cd`
- List — `ls`
- Search

Directory Structure
When considering a particular directory structure we need to consider the following issues:

1. **Efficient Searching**
2. **Naming** – should be convenient to users
   - Two users can have same name for different files
   - The same file can have several different names
3. **Grouping** – logical grouping of files by properties, (e.g., all Java programs, all games, ..)

Single-Level Directory
All files are contained in the same directory, which is easy to support and understand. However when the number of files increases or the system has more than one user, it has limitations. Since all the files are in the same directory, they must have unique names.
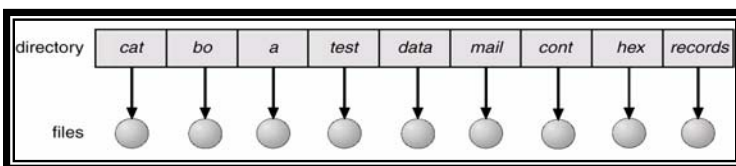
Single-level directory structure

Two-Level Directory
There is a separate directory for each user.
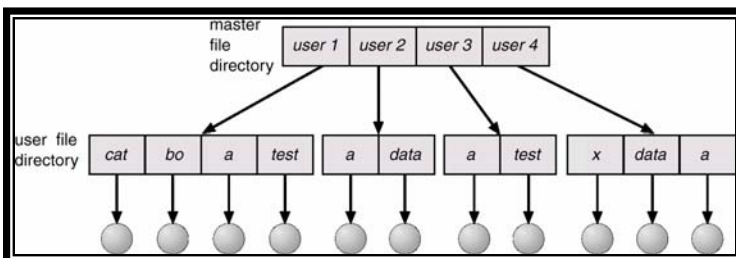
Two-level directory strucutre

When a user refers to a particular file, only his own user file directory (UFD) is searched. Thus different users can have the same file name as long as the file names within each UFD are unique. This directory structure allows efficient searching. However, this structure effectively isolates one user from another, hence provides no grouping capability.

Tree Directory
Here is the tree directory structure. Each user has his/her own directory (known as user's home directory) under which he/she can create a complete directory tree of his/her own.
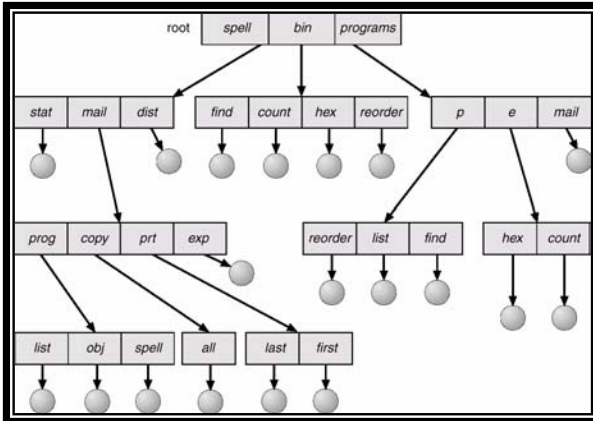
Tree directory structure

The tree has a root directory. Every file in the system has a unique pathname. A path name is the path from the root, through al the subdirectories to a specified file. A directory/subdirectory contains a set of files or subdirectories. In normal use, each user has a current directory. The current directory should contain most of the files that are of current interest to the user. When a reference to a file is made, the current directory is searched. If a file is needed that is not in the current directory, then the user must either specify a path name or change the directory to the directory holding the file(using the cd system call).This structure hence supports efficient searching. Allowing the user to define his own subdirectories permits him to impose a structure on his files.A lso users can access files of other users.

UNIX / Linux Notations and Concepts
- Root directory (/)
- Home directory
    - ~, $HOME, $home
    - cd ~
    - cd
- Current/working directory (.)
    - pwd
- Parent of Current Directory (..)
- Absolute Pathname
    - Starts with the root directory
    - For example, /etc, /bin, /usr/bin, /etc/passwd, /home/students/ibraheem
- Relative Pathname

- o Starts with the current directory or a user's home directory
- o For example, ~/courses/cs604, ./a.out

Operating Systems
Lecture No. 42

Reading Material
- Chapter 11 of the textbook
- Lecture 43 on Virtual TV

Summary
- Directory Structures
- Links in UNIX/Linux
- File System Mounting
- File Sharing
- File Protection

Acyclic-Graph Directories

A tree structure prohibits sharing of files. An acyclic graph allows directories to have shared subdirectories and files. The same file may be in two different directories.
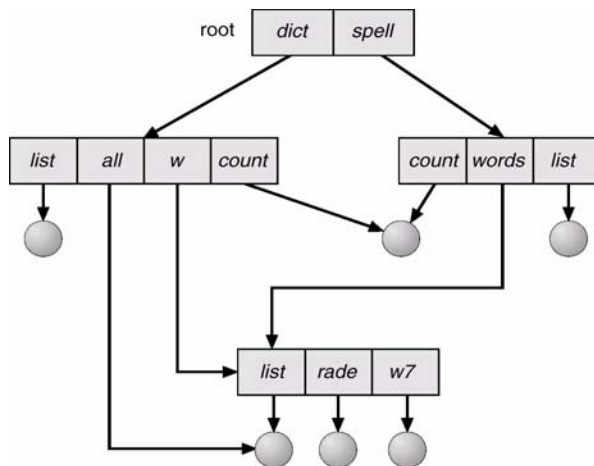
Acyclic-graph directory structure

A shared file is not the same as two copies of the file. Only one actual copy exists, so any changes made by one user are immediately visible to the other. A common way of implementing shared files and directories is to create a new directory entry called a link, which is effectively a pointer to another file or subdirectory. A link can be implemented as an absolute or relative path name. A file may now have multiple absolute path names. This problem is similar to the aliasing problem in programming languages. Consequently distinct file name may refer to the same files. If we are traversing the entire file system-to find a file, to accumulate statistics, etc, this problem becomes significant since we do not want to traverse the shared structures more than once. Another problem involves deletion.

If the file is removed when anyone deletes it, we may end up with dangling pointers to the now-nonexistent file.

Solutions: Another approach is to preserve the file until all references to it are deleted. When a link or a copy of the directory entry is establishes, a new entry is added to the file-reference list. When a link is deleted, we remove its entry on the list. The file is deleted when its file-reference list is empty. Since the reference list can be very large we can keep a count of the number of references. A new link or directory increments the **reference count**, deleting a link or entry decrements the count. When the count is 0, the file can be deleted. UNIX uses this solution for hard links. **Backpointers** can also be maintained so we can delete all pointers.

General Graph Directory
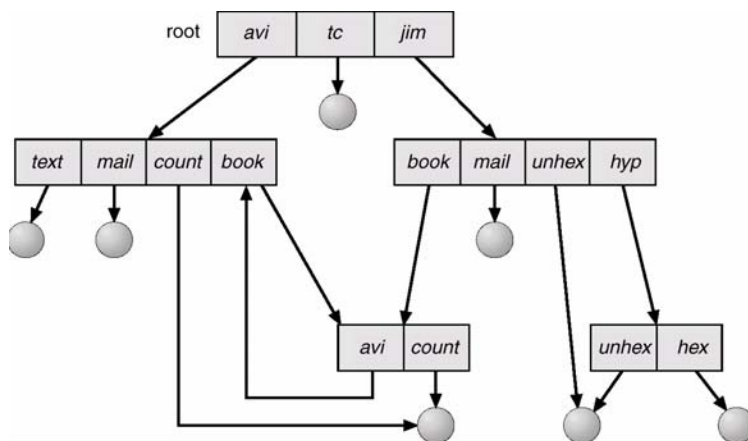
General graph directory

One serious problem with using an acyclic-graph structure is ensuring that there are no cycles. A solution is to allow only links to files not subdirectories. Also every time a new link is added use a **cycle detection algorithm** to determine whether it is OK. If cycles are allowed, we want to avoid searching any component twice. A similar problem exists when we are trying to determine when a file can be deleted. A value of 0 in the reference count means no more references to the file/directory can be deleted. However, cycles can exist, e.g, due to self-referencing. In this case we need to use a garbage collection scheme, which involves traversing the entire file system, marking everything that can be accessed. Then a second pass collects everything that is not marked onto a list of free space. However this is extremely time consuming and is seldom used. However it is necessary because of possible cycles in a graph.

Links in UNIX
UNIX supports two types of links:
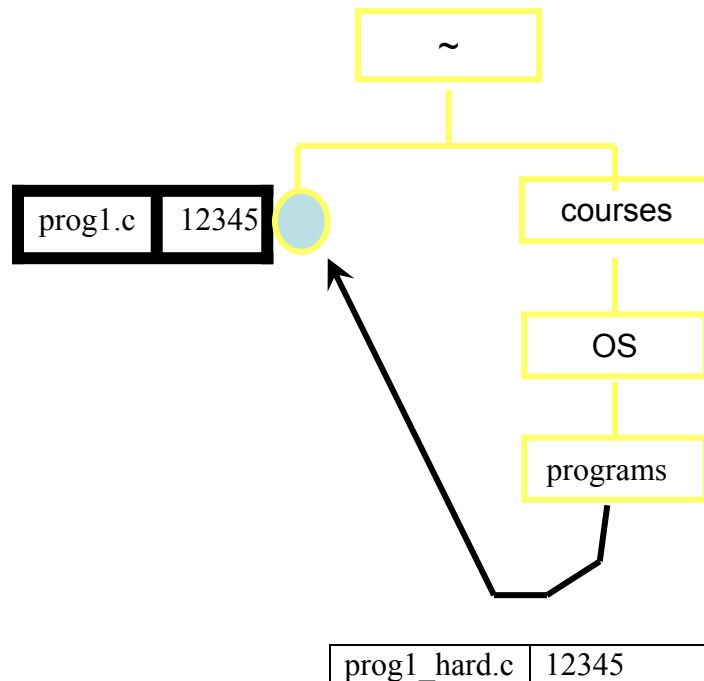- **Hard links**
- **Soft (symbolic) links**

The ln command is used to create both links, ln –s is used to create a soft link
- `ln [options] existing-file new-file`
- ln [options] existing-file-list directory

Examples: The first command creates a hard link ~/courses/OS/programs/prog1_hard.c to an existing file ~/prog1.c. The second command creates a soft link ~/prog2_soft.c to an existing file ~/courses/OS/programs/prog2.c. The diagrams below show the directory structures after these links have been created. Note that directory entries for hard links to the same file have the same inode number.

```
ln ~/prog1.c ~/courses/OS/programs/prog1_hard.c
ln -s ~/courses/OS/programs/prog2.c ~/prog2_soft.c
```

Hard Links

When a hard link is created, a directory entry for the existing file is created—there is still only one file. Both entries have the same inode number. The link count is incremented by one in the inode for the file. No hard links can be created for directories. Also hard links cannot be established between files that are on different file systems. In UNIX, a file is removed from the file system only if its hard link count is 0.

Soft Links
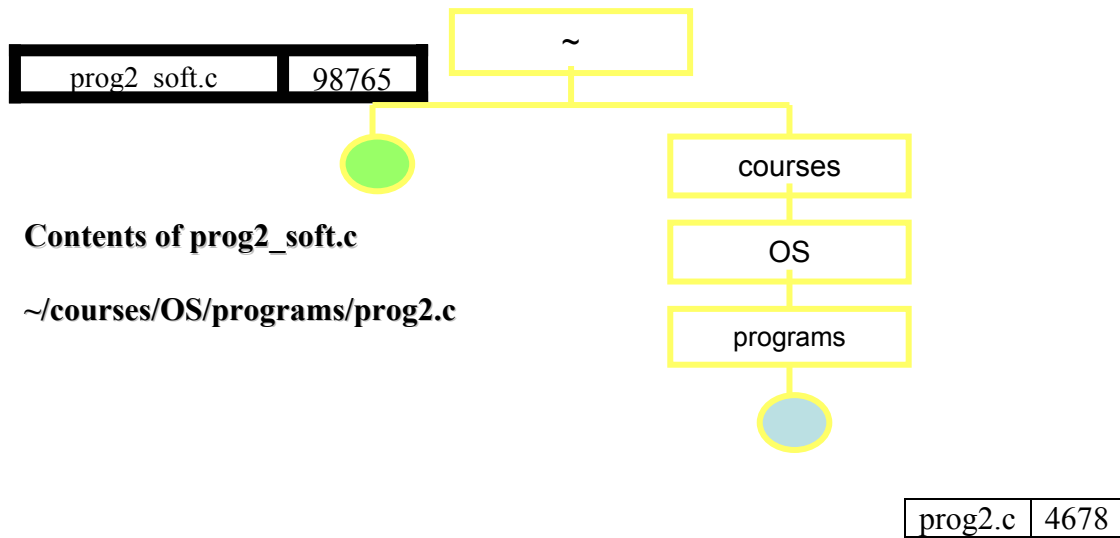
A file of type 'link' is created, which contains the pathname for the existing file as specified in the <mark>ln</mark> command. The existing file and the new (link) files have different inode numbers. When you make a reference to the link file, the UNIX system sees that the type of file is link and reads the link file to find the pathname for the actual file to which you are referring. When the existing file is removed, you have a 'dangling pointer' to it in the link file. Soft links take care of all the problems inherent in hard links. They are flexible. You may have soft links to directories and across file systems. However, UNIX has to support an additional file type, the link type, and a new file is created for every link, slowing down file operations.

File System Mounting

A file system is best visualized as a tree, rooted at /. /dev, /etc, /usr, and other directories in the root directory are branches, which may have their own branches, such as /etc/passwd, /usr/local, and /usr/bin. Filling up the root file system is not a good idea, so splitting /var from / is a good idea.Another common reason to contain certain directory trees on other file systems is if they are to be housed on separate physical disks, or are separate virtual disks, or CDROM drives.

    Mounting makes <mark>file systems,</mark> files, directories, devices, and special files available for use at a particular location. **Mount point** is the actual location from which the file system is mounted and accessed. You can mount a file or directory if you have access to the file or directory being mounted and write permission for the mount point

There are types of mounts:

- <mark>Remote mount</mark>
- <mark>Local mount</mark>

Remote mounts are done on a remote system on which data is transmitted over a <mark>telecommunication line.</mark> **Local mounts** are mounts done on your <mark>local system.</mark>

Mounting in UNIX

All files accessible in a Unix system are arranged in <mark>one big tree</mark>, the file hierarchy, rooted at /. These files can be spread out over several devices. The `mount` command serves to attach the file system found on some device to the big file tree. Conversely, the umount command will detach it again. Here is the syntax of the `mount` command
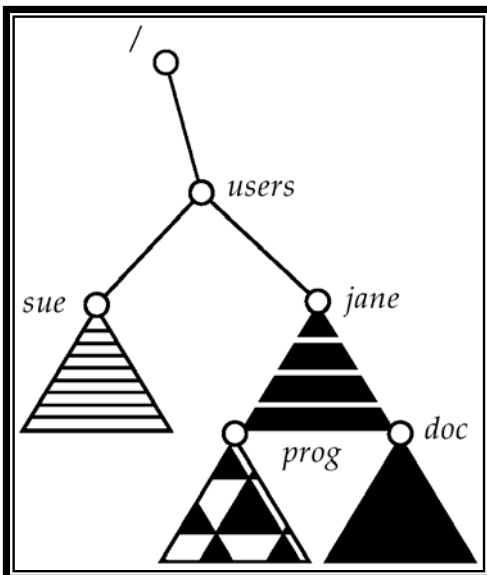
```
mount -t type device dir
```

This command tells the kernel to attach the file system found on *device* (which is of type *type*) at the directory *dir*. The previous contents (if any) and owner and mode of *dir* become invisible. As long as this file system remains mounted, the pathname *dir* refers to the root of the file system on *device*.
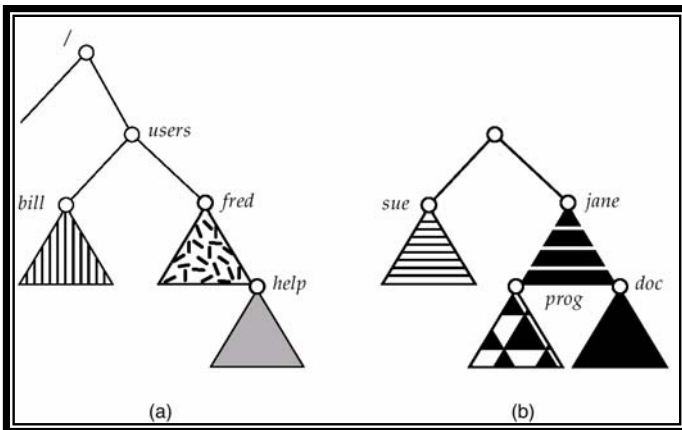
Existing Tree   Unmounted filesystem

New Tree after mounting Filesystem

File System Space Usage

On SuSE Linux
```
$ df
Filesystem              1K-blocks      Used Available Use% Mounted on
/dev/hda3                74837584  11127072  59908892  16% /
tmpfs                      257948        12    257936   1% /dev/shm
/dev/hda1                   19976      6960     11985  37% /boot
inbox:/var/spool/mail
                         66602516   3319996  59899232   6% /var/spool/mail
upfile1a:/usr1.a        230044816  70533680 147825456  33% /usr1.a
upfile2a:/usr2.a        230044816 118228296 100130840  55% /usr2.a
upibma:/usr3.a           16713676    804252  15039103   6% /usr3.a
upfile4a:/usr4.a        230044816  14594384 203764752   7% /usr4.a
$
```

On Solaris 2
```
$ df -v
Mount Dir    Filesystem       blocks         used      free       %used
/            /dev/dsk/c0t12d  7557677     2484225   4997876        34%
/proc          /proc               0           0         0         0%
/etc/mntta     mnttab              0           0         0         0%
/dev/fd        fd                  0           0         0         0%
/var/run       swap           510103          22    510081         1%
/tmp           swap           683241      173160    510081        26%
/oldexport   /dev/dsk/c0t8d0  4668856     4229110    393058        92%
/export/ho   /dev/dsk/c0t12d 23684712    21714309   1733556        93%
$
```

File Sharing

Sharing of files on multi-user systems is desirable. People working on the same project need to share information. For instance: software engineers working on the same project need to share files or directories related to the project

Sharing may be done through

- **Duplicating files:** Make copies of the file and give them to all team members. This scheme works well if members of the team are to work on these shared files sequentially. If they work on the files simultaneously, the copies become inconsistent and no single copy reflects the works done by all members. However it is simple to implement.
- **Common login** for members of a team: The system admin creates a new user group and gives the member access to the new account. All files and directories created by any team member under this account and are owned by the team. This works well if number of teams is small and teams are stable. However a separate account is needed for the current project and the system administrator has to create a new account for every team
- Setting appropriate **access permissions.** Team members put all shared files under one member's account and the access permissions are set so all the members can access it. This scheme works well if *only* this team's members form the user group. File access permissions can be changed using the chmod system call:

```
chmod [options] octal-mode file list
chmod [options] symbolic -mode file-list
```

A few examples:

–To let people in your UNIX group add, delete, and rename files in a directory of yours - and read or edit other people's files if the file permissions let them - use **chmod 775** *dirname*.

–To make a private file that only you can edit, use `chmod 600 filename`. To protect it from accidental editing, use `chmod 400 filename`.

- **Common groups** for members of a team. : System admin creates a new user group consisting of the members of team only. All team members get individual logins and set access permissions for their files so that they are accessible to other group members
- **Links.** A link is a way to establish a connection between the file to be shared and the directory entries of the users who want to have aces to this file. The two types of links supported by UNIX:
    - –Hard link
    - –Soft/symbolic link

Operating Systems
Lecture No. 44

Reading Material
- Chapters 11 and 12 of the textbook
- Lecture 44 on Virtual TV

Summary
- File Protection
- In-Memory Data Structures
- Space Allocation Techniques
- Contiguous, Linked, Index

Protection
The need to protect files is a direct result of the ability to access files. Systems that do not permit access to the files of other users do not need protection. Thus we could provide complete protection by prohibiting access. Alternatively we could provide free access with no protection. Both approaches are too extreme for general use. What is needed is controlled access. File owner/creator should be able to control
- What can be done
- By whom

Several types of operations may be controlled:
- Read: read from the file
- Write: write or rewrite to the file
- Execute: Load the file into memory and execute it
- Append: Write new information at the end of the file
- Delete: Delete the file and free its space for possible reuse
- List: List the name and attributes of the file

UNIX Protection
UNIX recognizes three modes of access: **read**, **write**, and **execute** (r, w, x). The execute permission on a directory specifies permission to **search** the directory.

The three classes of users are:
- **Owner:** user is the owner of the file
- **Group:** someone who belongs to the same group as the owner
- **Others:** everyone else who has an account on the system

A user's access to a file can be specified by an octal digit. The first bit of the octal digit specifies the read permission, the second bit specifies the write permission, and the third bit specifies the execute permission. A bit value 1 indicates permission for access and 0 indicates no permission. Here is an example:
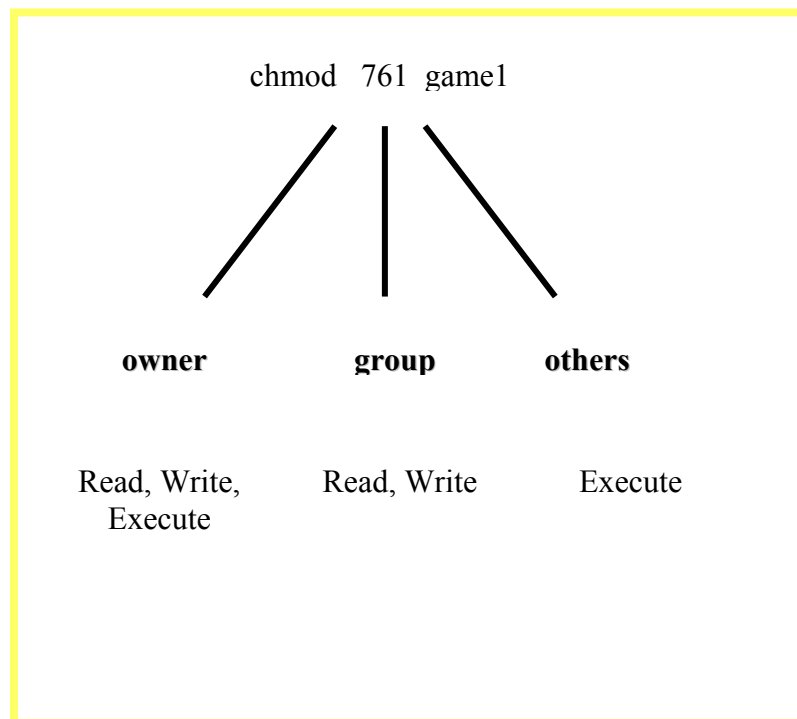
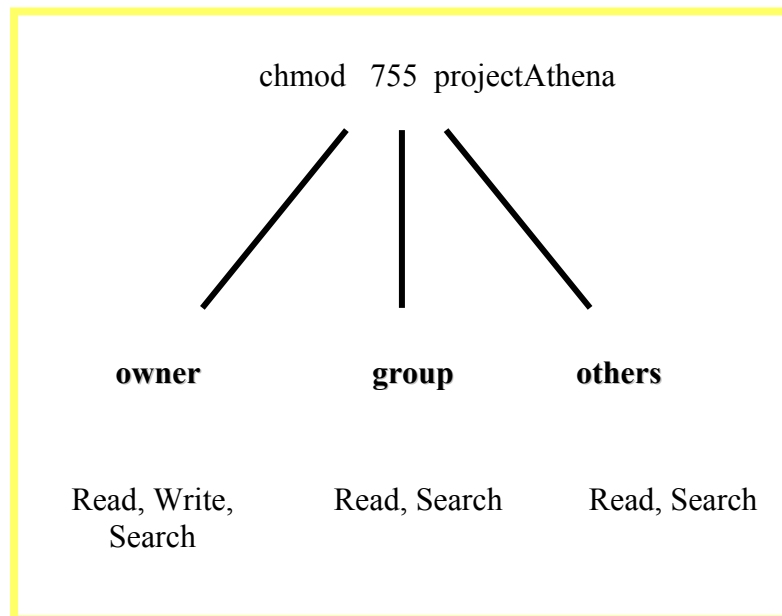|		**r w x**
a) Owner **access:** 7		1 1 1
b) Group **access:** 6		1 1 0
c) Public **access:** 1		0 0 1

Each user in a UNIX system belongs to a group of users as assigned by the system administrator when a user is allocated an account on the system. A user can belong to multiple groups, but a typical UNIX user belongs to a single group.

For a particular file or subdirectory, we need to set appropriate **access permissions** for proper protection.

Default Permissions

The default permissions on a UNIX/Linux system are 777 for executable files and directories and 666 for text files. You can use the umask command to set permission bits on newly created files and directories to 1, except for those bits that are set to 1 in the 'mask'. You can use the chmod command to set permissions on existing files and directories. We give some examples of the chmod and umask commands below.

chmod   761   game1

owner   **group**   **others**

Read, Write,   Read, Write   Execute
Execute

```
                    chmod   755  projectAthena


                owner            group           others


            Read, Write,      Read, Search     Read, Search
              Search
```

Sample commands

chmod 700 ~ ………….. Set permissions on home directory to 700
chmod 744 ~/file……… Set permissions on ~/file to 744
chmod 755 ~/directory… Set permissions on ~/directory 755
ls –l ~ ………………….. Display permissions and some other attributes for all files and
                              directories in your home directory
ls –ld ~ ……………….… Display permissions and some other attributes for your home directory
ls –l prog1.c …………… Display permissions and some other attributes for prog1.c in your
                              current directory
ls –ld ~/courses ……….... Display permissions and some other attributes for your home directory


The **umask** command sets default permissions on newly created files and directories as
                              (default permissions – mask value)
Here are some sample commands
umask ……… Display current mask value (in octal)
umask 022 …. Set mask value to octal 022 (turn off write permission for 'group' and 'others'
touch temp1 .. Create an empty file called temp1
ls –l temp1 …. Display default permissions and some other attributes for the temp1 file

File Control Block

<mark>A file control block is a memory data structure that contains most of the attributes of a file.</mark> In UNIX, this data structure is called <mark>inode</mark> (for index node). Here are possible values in this data structure.
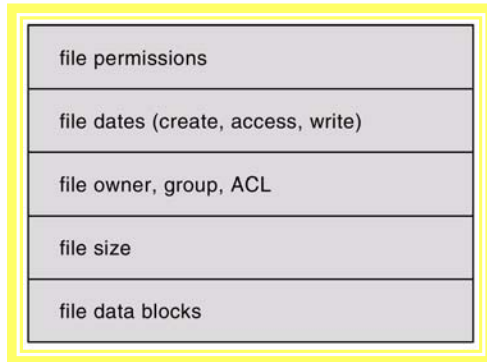

UNIX inode

In-Memory Data Structures

The following upper-level data structures needed for file system support.

- An in-memory partition table containing information about each mounted partition
- An in-memory directory structure that holds the directory information of recently accessed directories
- The system-wide open file table contains pointer to the FCB (UNIX inode) of each open file as well as read/write pointer
- The FCB for each open file
- The per process file descriptor table contains a pointer to the appropriate entry in the system wide open file table as well as other information

Here are the connections between various in-memory data structures. UNIX specific mappings follow this diagram.

Connections between various in-memory data structures

Per Process File Descriptor Table

File Descriptor

File Table

Inode Table

File's contents

0
1
2
3
4

OPEN_MAX — 1
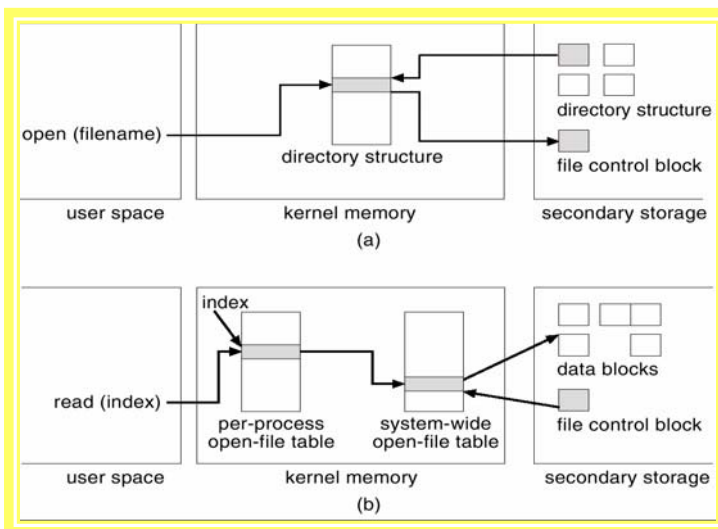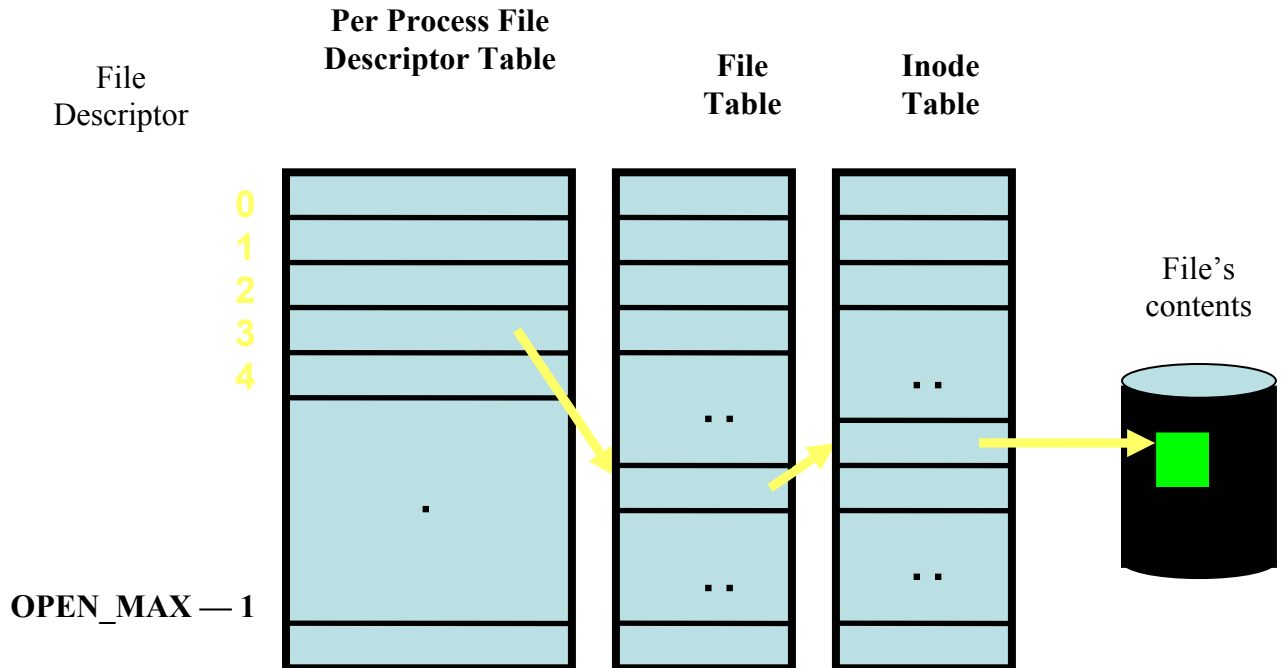
From File Descriptor to File Contents—The UNIX/Linux In-Memory Data Structures

The open call passes a file name to the file system. When a file is opened, the directory structure is searched for the given file name and file's inode. An entry is made in the per process open-file table (aka the file descriptor table), with a pointer to the entry in the system wide open file table. The system wide open file table contains the pointer to the current location in the file and a pointer to file's inode. The open call returns an index for the appropriate entry in the per-process file system table. All file operations are performed via this index, which is called the file descriptor in UNIX/Linux jargon.

Space Allocation Methods

We now turn to some file system implementation issues, in particular space allocation techniques and free space management methods. Here are the three commonly used methods for file space allocation.

- Contiguous allocation
- Linked allocation
- Indexed allocation
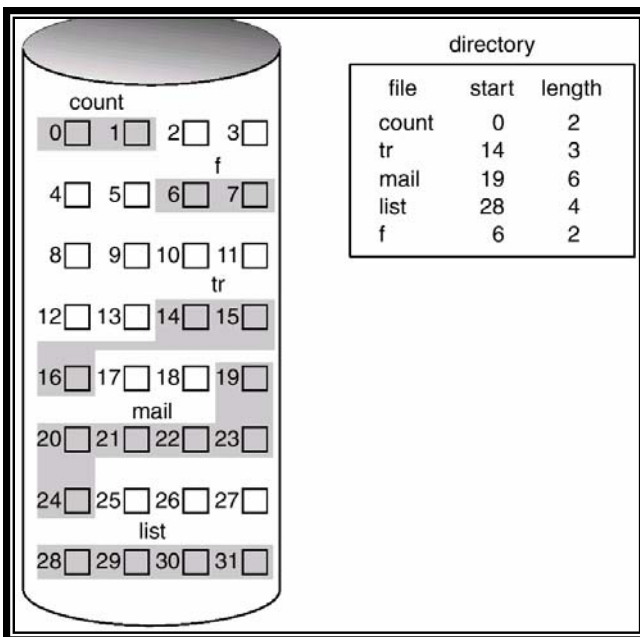
Contiguous Allocation

The contiguous allocation method requires each file to occupy a set of contiguous blocks on the disk. The directory entry for each file contains starting block number and file size (in blocks). Disk addresses define a linear ordering on the disk. With this ordering, assuming only one job is accessing the disk, accessing b+1 block after block b normally requires no head movement. When head movement is needed it is only one track. Both sequential and direct access can be supported by contiguous allocation. For direct access to block I of a file that starts at block b we can immediately access block b+i.

Best-fit, first-fit, or worst-fit algorithms are the strategies used to select a hole from the set of available holes. Neither first fit, nor best fit is clearly best in terms of both time and storage utilization, but first fit is generally faster.

These algorithms suffer from the problem of external fragmentation. As files are allocated or deleted, the free disk is broken into little pieces. This situation results in **external fragmentation** of disk (similar to external fragmentation of main memory due to segmentation). Disk defragmenter utility needs to be used for removing external fragmentation.

Determining how much space is needed for a file is another problem. User needs to declare file size, and estimating file size may be difficult. Also file growth is expensive in contiguous allocation. Worst-fit space allocation algorithm can be used to allow growth in a file's size.

The following diagram shows an example of the contiguous allocation scheme.

Contiguous allocation

Linked Allocation
Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request. There is no wastage of space. However, a major disadvantage with linked allocation is that it can be used only for sequential access files. To find the ith block of a file, we must start at the beginning of that file and follow the pointers until we get back to the ith block. Consequently it is inefficient to support a direct access capability for linked allocation files.
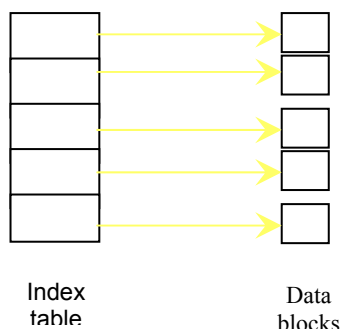
Here is an example of linked allocation.
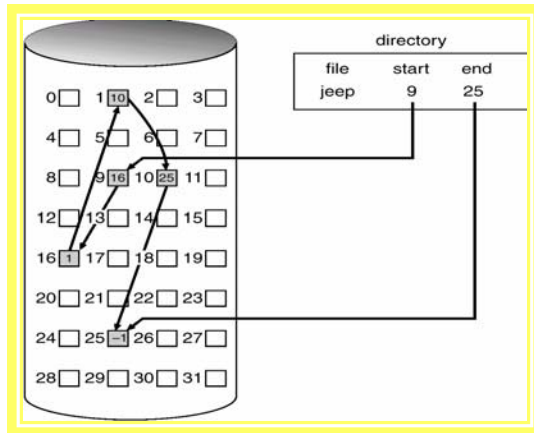
Linked allocation

Index Allocation

Indexed allocation brings all the pointers to the block together into a disk block, known as the **index block**. Here is the logical view of the relationship between an index block and a file's data blocks.

Index
table

Data
blocks

Logical view of index allocation

Each file has its own index block, which is an array of disk block addresses. The *i*th entry in the index block points to the *i*th block of the file. The directory contains the address of the index block. To read the *i*th block, we use the pointer in the *i*th index-block entry to find and read the desired block Here is an example of index allocation.
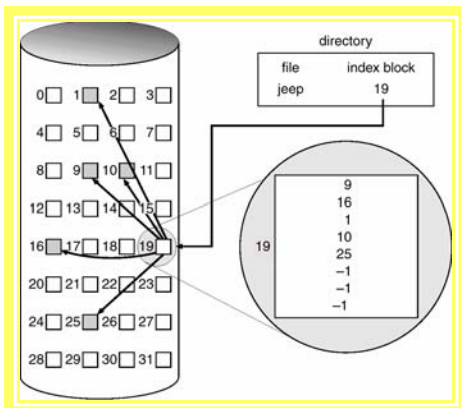
Index allocation

Operating Systems
Lecture No. 45

Reading Material
- Chapters 12 and 14 of the textbook
- Lecture 45 on Virtual TV

Summary
- Space Allocation Techniques (continued)
- Free Space Management
- Disk Structure and Scheduling

Index Allocation (continued from previous lecture)

Indexed allocation supports direct access without suffering from external fragmentation because any free block on the disk may satisfy a request for more space. Depending on the disk block size and file system size, a file may need more than one index block. In this case there are two ways of organizing index blocks:

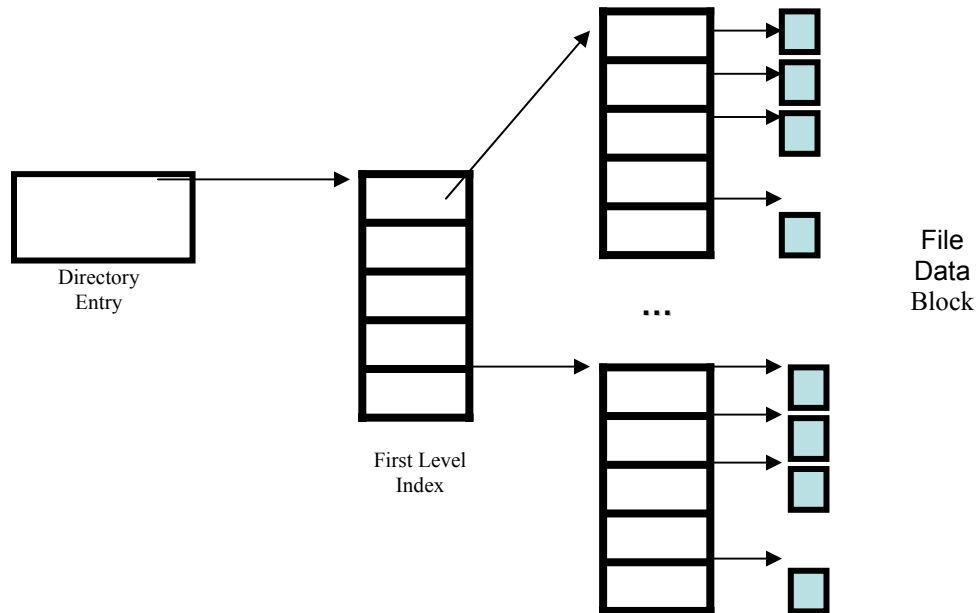### Linked scheme (linked list of index blocks)

An index block is normally one disk block. Thus, it can be read and written directly by itself. To allow for large files, we may link together several index blocks. For example, an index block might contain a small header giving the name of the file and a set of first 100 disk-blocks addresses. The next address (the last word in the index block) is nil (for a small file) or a pointer to another index block (for a large file), as shown below.

Directory Entry
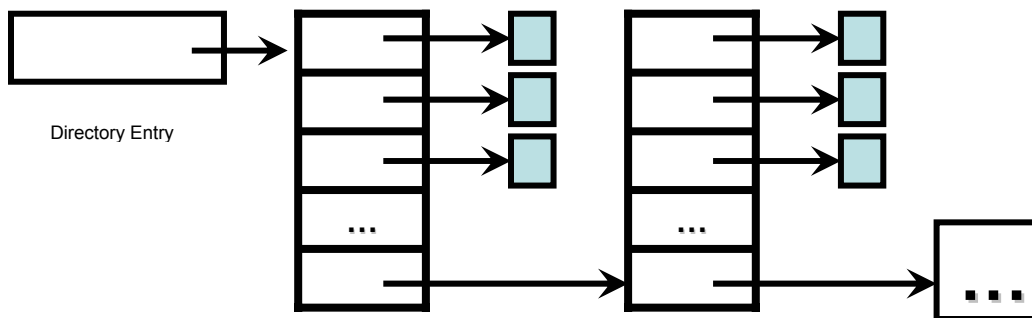
Linked scheme for interconnecting index blocks

Multi-level index scheme
The second method of handling multiple index blocks is to maintain multi-level indexing. In the following diagram, we show two-level index table.

Two level Index Table

UNIX Space Allocation
The UNIX file manager uses a combination of indexed allocation and linked lists for the index table. It maintains 10-15 direct pointers to file blocks, and three indirect pointers (one-level indirect, two-level indirect, and three-level indirect), all maintained in file's inode, as shown below.
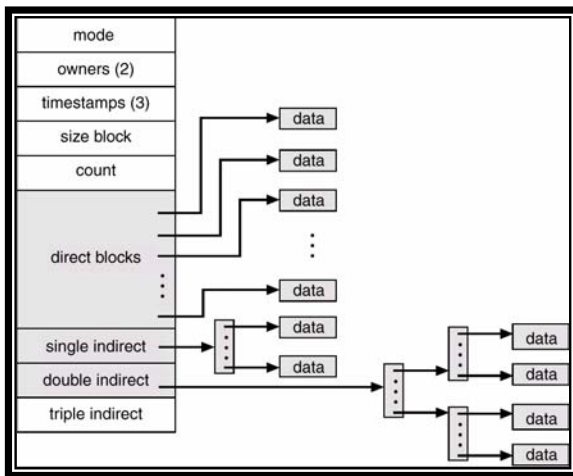
UNIX inode

Let's consider a UNIX system with following attributes:

- The block size is 4 KB block
- 4-byte disk pointers (which means, 1024 points per disk block)
- 10 direct pointers to file blocks

Maximum file size and pointer overhead?

10*4=40 KB of data may be accessed directly (in the above case). The maximum file size depends on the size of the blocks and the size of the disk addresses used in the system. The next pointers point to indirect blocks. The single indirect block is an index block containing not the data but rather the addresses of blocks that do contain data. Then there is a double indirect block pointer, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. Finally, the triple indirect block pointer points to first-level index block, which points to second-level index blocks, which point to third-level index blocks, which point to data blocks. With the given parameters, the maximum file size will be $[10 + 1024 + 1024^2 + 1024^3]$ blocks—multiply this by the block size to get size in bytes. Similarly, you can calculate the pointer overhead for the largest file.

File Allocation Table (FAT)

The file system on an MS-DOS floppy disk is based on **file allocation table** (FAT) file system in which the disk is divided into a reserved area (containing the boot program) and the actual file allocation tables, a root directory and file space. Space allocated for files is represented by values in the allocation table, which effectively provide a linked list of all the blocks in the file. Each entry is indexed by a block number and value in a table location contains block number for the next file block. First block number for a file is contained in file's directory entry. Special values designate end of file, unallocated and bad blocks. The following diagram summarizes the overall picture of FAT.
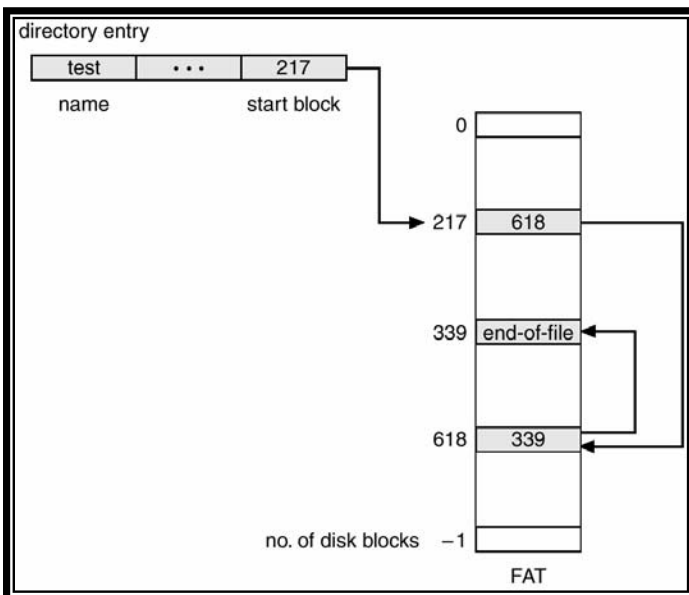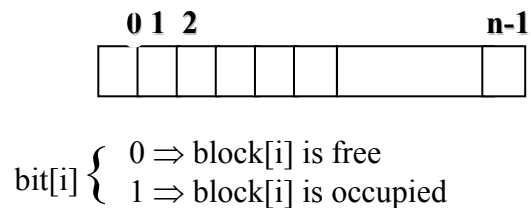

File Allocation Table (FAT)

Free-Space Management

Since disk space is limited, we need to reuse the space from deleted files for new files if possible. To keep track of free disk space, the system maintains a **free-space list.** The free space list records all *free* disk blocks-those not allocated to some file or directory. To create a file we search the free-space list for the required amount of space and allocate the space to the new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free space list.

Bit vector

Frequently, the free space list is implemented as a bit map or bit vector. Each block is represented by 1 bit. If the block is free, the bit is 1;if it is allocated, the bit is 0. This approach is relatively simple and efficient in finding the first free block or n consecutive free blocks on the disk.

$$\begin{array}{cccccc} \mathbf{0\ 1\ 2} & & & & & \mathbf{n\text{-}1} \end{array}$$

$$\mathrm{bit}[i] \begin{cases} 0 \Rightarrow \mathrm{block}[i] \text{ is free} \\ 1 \Rightarrow \mathrm{block}[i] \text{ is occupied} \end{cases}$$

The calculation of block number is:
(number of bits per word) * (number of 0-value words) + offset of first 1 bit

Example for **overhead of bit map**

$$\begin{array}{ll} \text{Block size} & = 4\,\text{KB} = 2^{12}\ \text{bytes} \\ \text{Disk size} & = 40\,\text{GB} = 40 * 2^{30}\ \text{bytes} \\ \text{Overhead} & = 40 * 2^{30}/2^{12} = 40 * 2^{18}\ \text{bits} \\ & = 40 * 32\,\text{KB} = 1280\,\text{KB} \end{array}$$

Linked list (free list)

Another approach to free space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory. The first block contains a pointer to the next free disk block and so on. However this scheme is not efficient. To traverse the list, we must read each block, which requires substantial I/O time. It cannot get contiguous space easily. The following diagram shows an example of free space management by using the linked list approach.
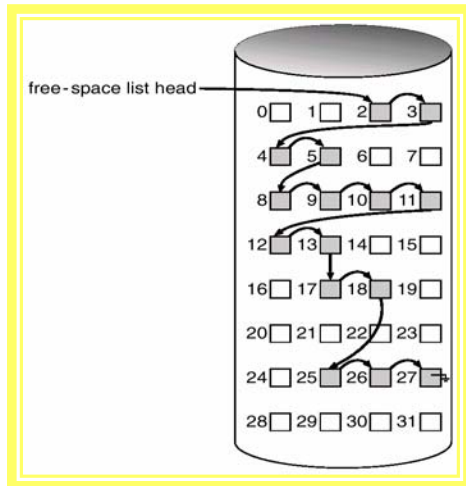
 Linked free space list on disk

   Similar to the example given for the bit map above, you can calculate the overhead for maintaining free space with linked list. We leave it as an exercise for you.

Grouping
A modification of free-list approach is to store the addresses of n free blocks in the first free block. The first n-1 blocks of these blocks are actually free. The last block contains addresses of the next n free blocks, and so on. The importance of this implementation is that the addresses of a large number of free blocks can be found quickly.

Counting
We keep the address of the first free block and the number n of free contiguous blocks that follow the first block in each entry of a block. This scheme is good for contiguous allocation. Although each entry requires more space, the overall list will be shorter.

I/O Operations
A number of I/O operations (inserting, deleting, and reading a file block) needed for the various allocation schemes indicate the goodness of these schemes. The following example illustrates this.

Assumptions
- Directory, Bit-map, and index blocks are in the main memory
- Worst-case and best-case scenarios
- File size of 100 blocks

Determine the number of I/O operations needed to
1. Insert a block after the 50th block
2. Read 50th block
3. Insert 101st block
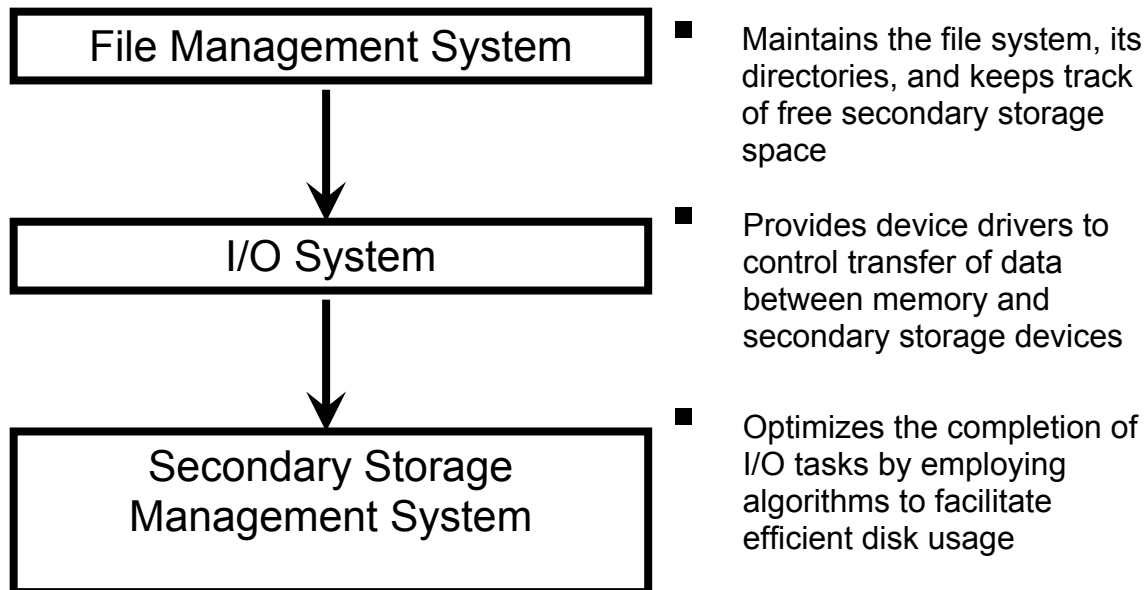4. Delete 50th block

We discussed this in the lecture. Please review the lecture. Here is how we approach the first part for the worst case scenario. In the worst-case, you don't have free block before or after the file. This means that you need to identify 101 contiguous free blocks on the disk, move the first 50 blocks to the new location (read into memory and write them to the new disk location, requiring 100 I/O operations), write the new block (one I/O operation), and move the last 50 blocks to the new location (another 100 I/O operations). Since the directory entry and bit-map blocks will be modified, we need to write them to disk (two I/O operations). This results in a total of 100+1+100+2 = 203 I/O operations.

In the best-case, we do have at least one free block available before or after the file, resulting in a total of 100+1+2 = 103 I/O operations. 100 operations are needed for shifting (i.e., moving) the first or last 50 blocks to left or right.

You can answer the remaining questions for contiguous allocation following the same approach and reasoning. Similarly, you can answer these questions for linked and index approach. When you are done, you will realize that index allocation approach is the best because it requires the smallest number of I/O operations for various file operations.

Secondary Storage Management

The following diagram shows the hierarchy of three kernel modules used for mapping user view of directory structure, free space management, file I/O, and secondary storage management. We have discussed some details of the top-most layer. We will not discuss details of the I/O system. Here is the discussion of one of the primary functions of the lowest layer in the diagram, i.e., disk scheduling.

| File Management System | ■ Maintains the file system, its directories, and keeps track of free secondary storage space |

| I/O System | ■ Provides device drivers to control transfer of data between memory and secondary storage devices |

| Secondary Storage Management System | ■ Optimizes the completion of I/O tasks by employing algorithms to facilitate efficient disk usage |

Three layers of file OS kernel used for managing user view of files, file operations, and file storage to disk

Disk Structure

Disks provide the bulk of secondary storage for modern computer systems. Magnetic tape was used as an early secondary storage medium but the access is much slower than for disks. Thus tapes are currently used mainly for backup, for storage of infrequently used information etc.

Modern disk drives are addressed as large one dimensional array of logical blocks, where the logical block is the smallest unit of transfer. The size of a logical block is usually 512 bytes, although some disks can be low-level formatted to choose a different logical block size, such as 1024 bytes.

The one dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Block 0 is the first sector of the first track on the outermost sector. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to the innermost.

By using this mapping, we can – at least in theory – convert a logical block number into an old style disk address that consists of a cylinder number, a track number within the cylinder and a sector number within that rack. In practice it is difficult to perform this translation for two reasons. First, most disks have some defective sectors but the mapping hides this by substituting spare sectors from elsewhere on the disk. Second, the number of sectors per track is not a constant on some drives. On media that use a constant linear velocity (CLV) the density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length so the more sectors it can hold. As we move from the outer zones to the inner zones, the number of sectors per track decreases. Tracks in the outermost tracks typically hold 40% more sectors than do tracks in the innermost zone. The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data, moving under the head. Alternatively the disk rotation speed can stay constant and the density of bits decreases from inner tracks to outer tracks to keep the data rate constant. This method is used in hard disks and is known as constant angular velocity (CAV).

Disk Scheduling

One of the responsibilities of the operating system is to use the computer system hardware efficiently. For the disk drives, meeting this responsibility entails having a fast access time and disk bandwidth. The access time has two major components. The seek time is the time for the disk arm to move the heads to the cylinder containing the desired sector. The rotational latency is the additional time waiting for the disk to rotate the desired sector to the disk head. The disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer. We can improve both the access time and the bandwidth by scheduling the servicing of disk I/O requests in a good order. Some of the popular disk-scheduling algorithms are:

- First-come-first-serve (FCFS)
- Shortest seek time first (SSTF)
- Scan
- Look
- Circular scan (C-Scan)
- Circular look (C-Look)

We now discuss the first four of these algorithms with an example each. We assume a disk with 200 cylinders.

First Come First Served Scheduling
The simplest form of disk scheduling is FCFS. This algorithm is intrinsically fair, but it generally does not provide the fastest service. Consider for example a disk queue with requests for I/O to blocks on cylinders

$$98,183,37,122,14,124,65,67$$

in that order. If the disk head is initially at cylinder 53 and the direction of movement is from left to right (i.e., from cylinder 0 to cylinder 199), it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65 and finally to 67, for a total head movement to of 640 cylinders.
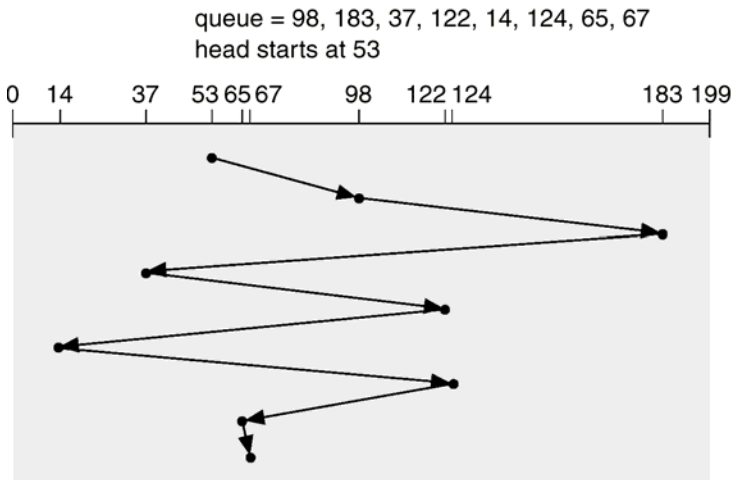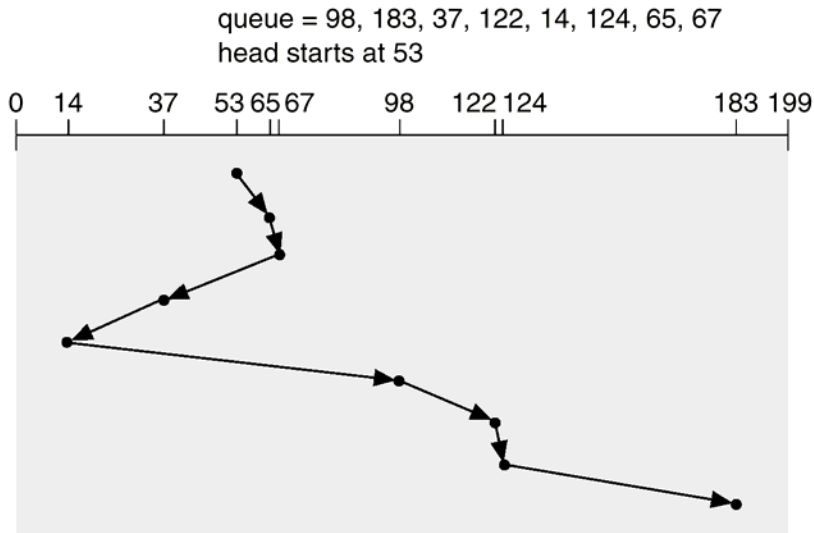

First-come-first-serve disk scheduling example

The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule. If the requests for cylinders 37 and 14 could be serviced together before or after the requests at 122 and 124, the total head movement could be decreased substantially and performance could be thereby improved.

SSTF Scheduling
It seems reasonable to service all the requests close to the current head position, before moving the head far away to service other requests. This assumption is the basis for the shortest seek time first (SSTF) algorithm. The SSTF algorithm selects the request with the minimum seek time from the current head position. Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head position.

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53

Shortest-seek-time-first (SSTF) disk scheduling algorithm

For our example request queue, the closest request to the initial head position 53 as at cylinder 67. From there, the request at cylinder 37 is closer than 98, so 37 is served next. Continuing we service the request at cylinder 14, then 98, 122, 124 and finally 183. This scheduling method results in a total head movement of only 236 cylinders—a little more than one third of the distance needed for FCFS scheduling of this request queue. This algorithm gives a substantial improvement in performance. However, it is not optimal; for the given example, the total head movement will be 208 cylinders if requests at cylinders 37 and 14 are served first.

Scan

In the Scan algorithm the disk arm starts at one end of the disk, and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed and servicing continues. The head continuously scans back and forth across the disk. We again use our example.

Before applying Scan to schedule requests, we need to know the direction of head movement in addition to the head's current position. If the disk arm is moving towards 0, the head will service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk servicing the requests at 65, 67, 98, 122, 124 and 183. The total head movement (or seek distance) is 236 cylinders. If a request arrives in queue just in front of the head, it will be serviced almost immediately; a request arriving behind the head will have to wait until the arm moves to the end of the disk, reverses direction and comes back.

The Scan algorithm is sometimes called the **elevator algorithm**, since the disk arm behaves like an elevator in a building servicing all the requests (people at floors), going up and then reversing to service the requests going down. The figure in the following diagram shows movement of the disk head for the request queue used for the previous examples.

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53

0   14   37   53 65 67   98   122 124   183 199
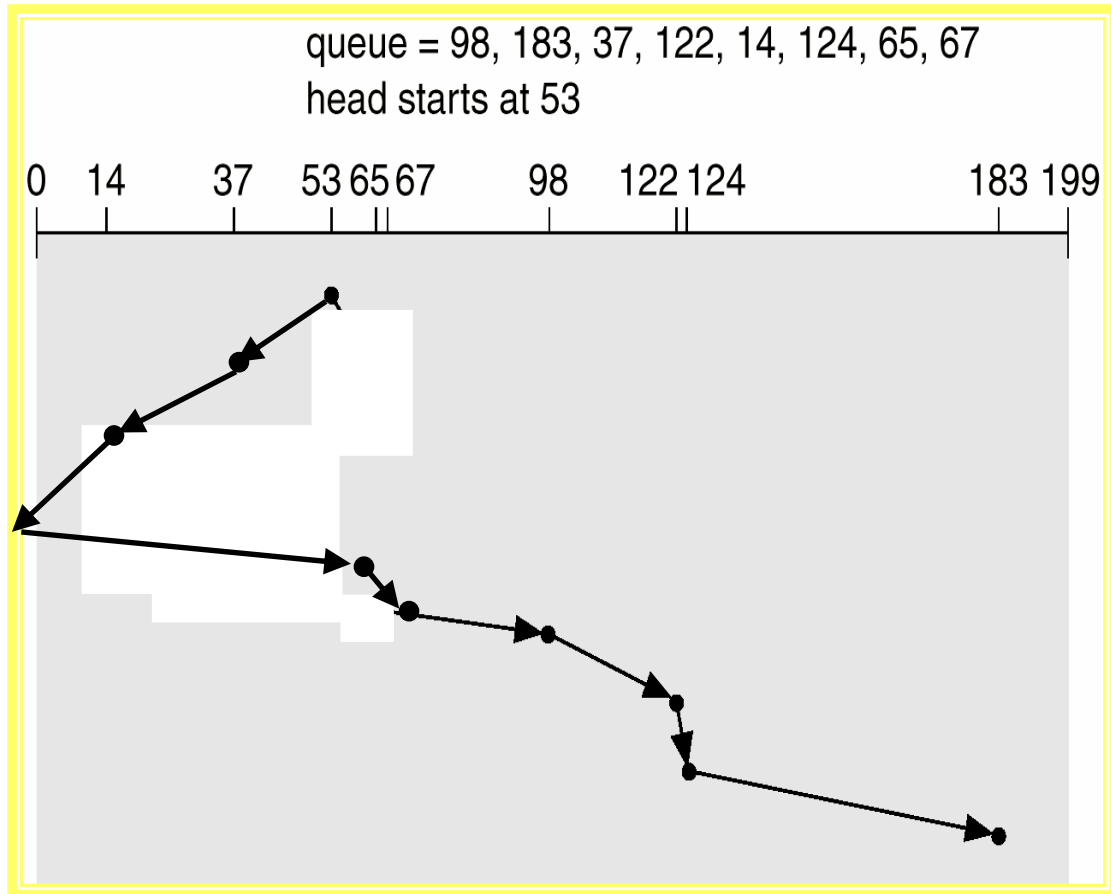
Scan disk scheduling algorithm with disk head moving from right to left

Look algorithm
This algorithm is a version of SCAN. In this algorithm the arm only goes as far as the last request in each direction, then reverses direction immediately, serving requests while going in the other direction. That is, it looks for a request before continuing to move in a given direction. For the given reques queue, the total head movement (seek distance) for the Look algorithm is 208.
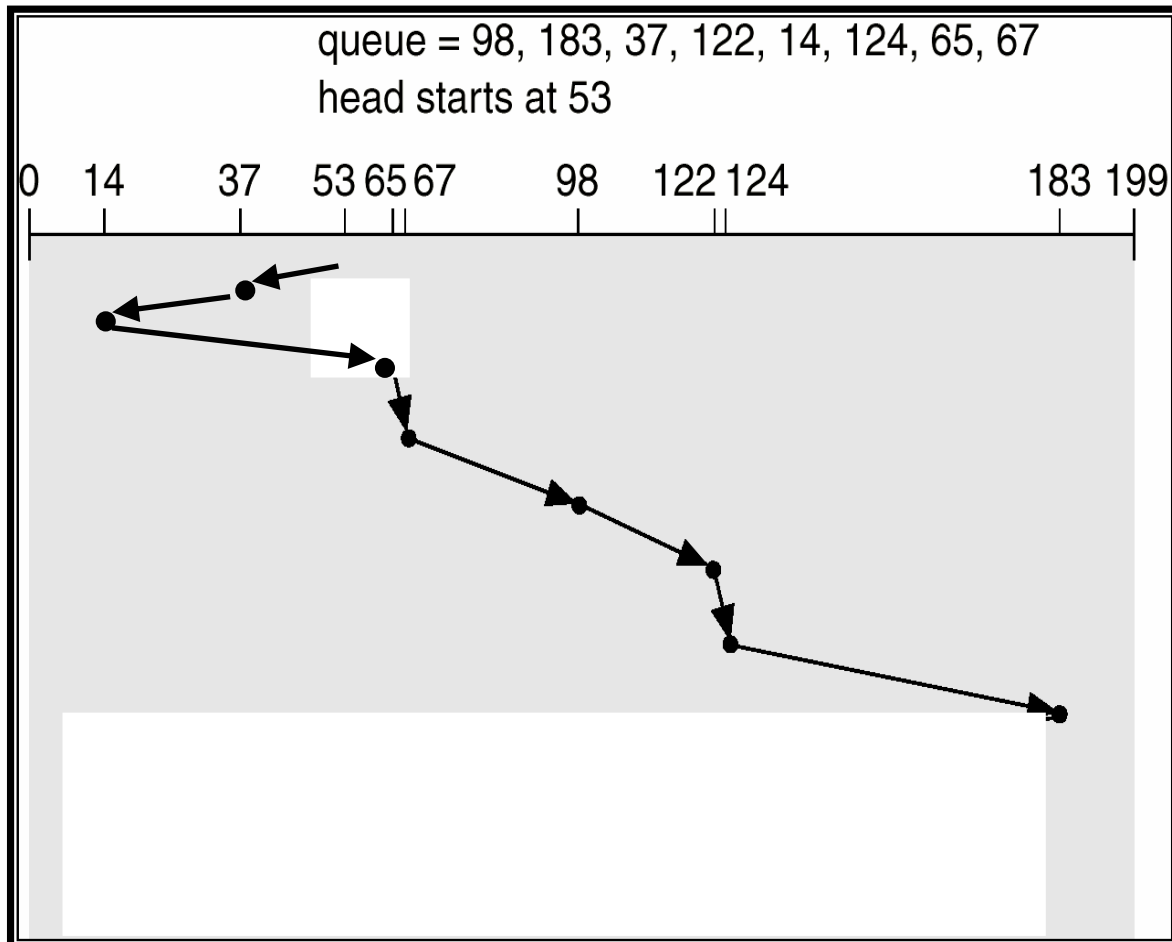
Look disk scheduling algorithm with the disk head moving from righ to left

C-Scan and C-Look algorithms
In the C-Scan and C-Look algorithms, when the disk head reverses its direction, it moves all the way to the other end, without serving any requests, and then reverses again and starts serving requests. In other words, these algorithms serve requests in only one direction.