OPERATING SYSTEMS

LESSON-1

Objectives:

Hello,students, it is your first class and I will introduce you to the basic concepts of an operating system -

- You will understand what is an operating system.
- Why should you learn Operating System.

To understand an Operating Systems ,you need to know what is an Operating System

An **Operating System** is system software which may be viewed as an organized collection of software consisting of procedures for operating a computer and providing an environment for execution of programs. It acts as an interface between users and the hardware of a computer system.

Now ,I will explain you the main purpose of an Operating System

- **Convenience**: transform the raw hardware into a machine that is more amiable to users.
- **Efficiency**: manage the resources of the overall computer system

Operating system can also be defined as:

- System software which may be viewed as an organized collection of software consisting of procedures for operating a computer and providing an environment for execution of programs.
- A large collection of software which manages resources of the computer system, such as memory, processor, file system and input/output devices. It keeps track of the status of each resource and decides who will have a control over computer resources, for how long and when.
- It acts as an interface between users and hardware of a computer system.

Colloquially, the term is most often used to mean all the software which "comes with" a computer system before any applications are installed.

Examples of operating systems

- UNIX
- GNU/Linux
- Mac OS
- MS-DOS

Let us discuss the fundamental goal of a Computer System

The fundamental goal of computer system is to solve user problems. Accordingly to achieve this goal has been designed. Since hardware alone cannot be used to solve the user problems software's are developed. These programs required certain common operations. The common operations for controlling and allocating resources are then brought together into one piece of software i.e. operating system. An operating system may process its tasks sequentially or concurrently. It means that the resources of the computer system may be dedicated to a single program until its completion or they may be allocated among several programs in different stages of execution.

Why should you need an Operating System?

The feature of operating system is to execute multiple programs in interleaved fashion or different time cycle is called multiple programming systems. Some of the important reasons why do you need an Operating System are as follows:

- User interacts with the computer through operating system in order to accomplish his/her task since it is his primary interface with a computer.
- It helps the user in understand the inner functions of a computer very closely.
- Many concepts and techniques found in operating system have general applicability in other applications.

An operating system is an essential component of a computer system. The primary objectives of an operating system are to make computer system convenient to use and utilizes computer hardware in an efficient manner.

An operating system is a large collection of software which manages resources of the computer system, such as memory, processor, rite system and input/output devices. It keeps track of the status of each resource and decides who will have a control over computer resources, for how long and when. The positioning of operating system in overall computer system is shown in figure 1.

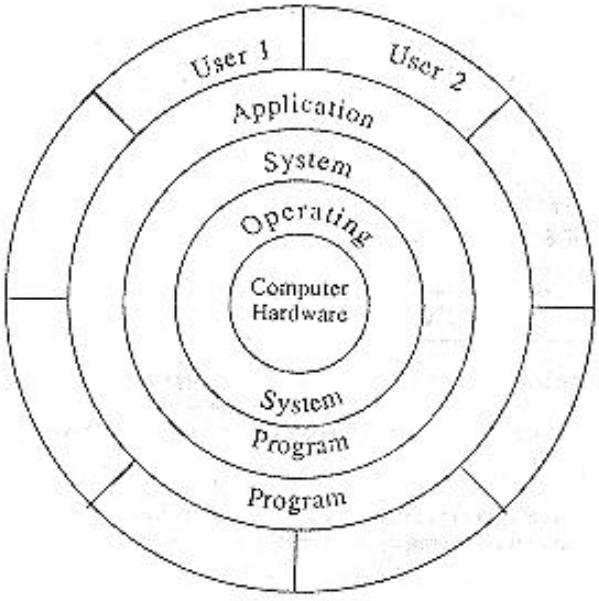


Figure 1: Component of computer system

From the diagram, it is clear that operating system directly controls computer hardware resources. Other programs rely on facilities provided by the operating system to gain access to computer system resources. There are two ways one can interact with operating system:

- By means of Operating System Call in a program
- Directly by means of Operating System Commands

System Call:

System calls provide the interface to a running program and the operating system. User program receives operating system services through the set of system calls. Earlier these calls were available in assembly language instructions but now a days these features are supported through high-level languages like C, Pascal etc., which replaces assembly language for system programming. The use of system calls in C or Pascal programs very much resemble pre-defined function or subroutine calls.

As an example of how system calls are used, let us consider a simple program to copy data from one file to another. In an interactive system, the following system calls will be generated by the operating system:

1. Prompt messages for inputting two file names and reading it from terminal.
2. Open source and destination file.
3. Prompt error messages in case the source file cannot be open because it is protected against access or destination file cannot be created because there is already a file with this name.
4. Read the source file.
5. Write into the destination file.
6. Display status information regarding various Read/Write error conditions. For example, the program may find that the end of the file has been reached or that there was a hardware failure. The write operation may encounter various errors, depending upon the output device (no more disk space, physical end of tape, printer out of paper add so on).
7. Close both files after the entire file is copied.

As you can observe, a user program takes heavy use of the operating system. All interaction between the program and its environment must occur as the result of requests from the program to the operating system.

Let us discuss various Operating System Commands:

Apart from system calls, users may interact with operating system directly by mews of operating system commands.

For example, if you want to list files or sub-directories in MS-DOS, you invoke dir command. In either case, the operating system acts as an interface between users and the hardware of a computer system. The fundamental goal of computer systems is to solve user problems. Towards this goal computer hardware is designed. Since the bare hardware alone is not very easy to use, programs (software) are developed. These programs require certain common operations, such as controlling peripheral devices. The command function of controlling and allocating resources are then brought together into one piece of software; the operating system.

To see what operating systems are and what operating systems do, let us consider how they have evolved over the years. By tracing that evolution, you can identify the common elements of operating systems and examine how and why they have developed as they have.

OS and Hardware Development

OS development has gone hand-in-hand with hardware development:

- Interrupts drive data transfer (using multiple CPUs, one designed exclusively for I/O processing)
- Direct-memory-access (DMA) data transfer
- Hardware memory protection (to validate addresses)
- Hardware instruction protection (only "special" users can execute some machine instructions)
- Support for other interrupts: clock.
- Old days: "**busy wait**", e.g. printing:

Pseudo-code:

1. check hardware switch (physical memory location which printer can change) to see if printer is ready for next character.
2. if switch not set, go to step 1.
3. send printer next char.
4. if more to print, go to step 1.
5. stop.

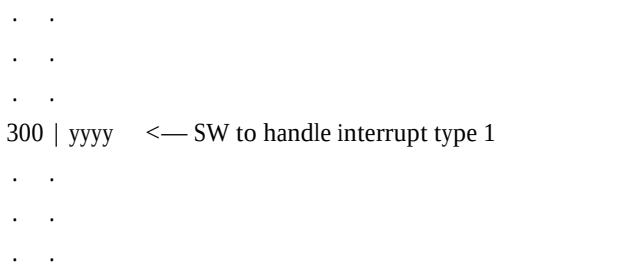
(steps 1 and 2 constitute the busy wait.) This same idea works for buffers as well as single characters: processor fills buffer (of maybe 512 characters), then uses busy wait for external device to signal that it has emptied the buffer. Though buffers are going to be more helpful if not using busy wait (why?)

This waste of valuable CPU time (due to speed mismatches between CPU and external devices) is reduced by introduction of hardware interrupts.

Interrupt Handling

main memory

```
addr |  inst
-----+-------
 100 | br 200    Assume HW supports 16 interrupts
 101 | br 300
 102 | br 350
 104 | br 550
  .    .
  .    .
  .    .
 115 | br 950
  .    .
  .    .
  .    .
 200 | xxxx     <— SW to handle interrupt type 0
 201 | xxxx
```



HW support: for each interrupt type (e.g. clock interrupt, IO completion, change to supervisor state, invalid instruction) the HW changes to program counter to a "prewired" memory address (in the above example, 100 for interrupt type 0, 101 for interrupt type 1, etc.) This results in the specific code to handle that interrupt being executed.

Examples:

1. Suppose clock interrupts are "wired" to memory location 100. Then in a typical RR system, the interrupt causes a transfer to code which checks the cause of the timer interrupt. If the cause is that a process has gotten its "time slice" then scheduling code will save the state of the current process, maybe do accounting, pick the next process to execute, restore its state, and start it.
2. Suppose memory location 102 is associated with the "invalid instruction" interrupt. What then?

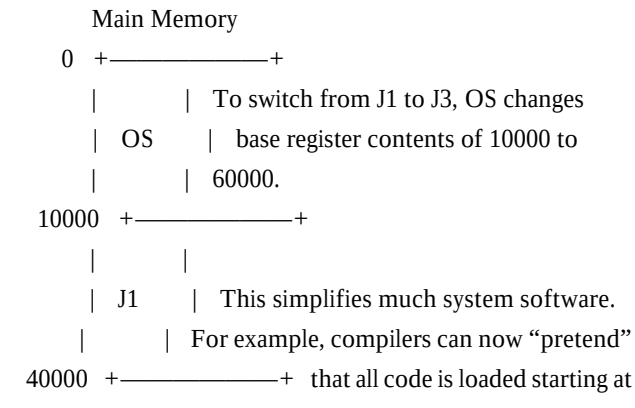
Well, for one thing, this allows things the same machine code to be used in machines that don't have quite the same instruction sets (say, across a product line from cheap slow processors to expensive fast processors). (Homework: explain how this might be done.)

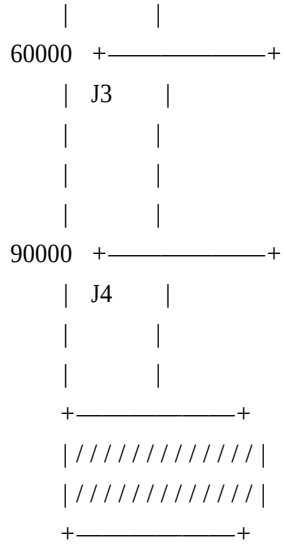
Once the hardware supports this, then I/O can be handled much more efficiently. It is also used for security. If some instructions are privileged, then if a "normal" process attempts to execute them, the OS (through interrupt code) can validate what is happening.

The point has been to do things FAST. So that things are only checked when they need to be. This requires a combination of HW and SW support.

Memory Protection:

1. Base register: contents of base register are automatically added (by HW) to each address. Not done by SW since this would be too slow.





2. Limit register: make sure user does not reference beyond allocation memory size. Load limit register with size of memory allocated to process. HW **traps** (that is causes an interrupt) if the process attempts to use an address larger than limit register contents.

Note: this is NOT checked in software (much too slow). It must be done in hardware as a side effect of addressing memory SO THAT NOTHING SLOWS MEMORY REFERENCES DOWN. THE SAME GOES FOR THE USE OF THE BASE REGISTER. If memory references are validated, the mechanism must be very fast.

How does the OS get around this? Two basic approaches:

- some machines allow interrupts to be turned off, or
- interrupt handler can check to see who is trying to do memory reference.

Review Exercise:

1. Define an Operating System?

_____
_____
_____
_____
_____
_____
_____
_____

2. Explain the need for an Operating System?

_____
_____
_____
_____
_____
_____
_____
_____

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems / Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

Notes

LESSON-2

Today I will explain you the evolution of an operating system and various types of operating systems

Evolution of an Operating system:

As the need for better processing raised due to the increase in demand for better processing speed and efficiency the operating systems have been enhanced with extra features.

Let us see what is Serial Processing?

Instructions and data are feeded into the computer by means of console switches or perhaps through a hexadecimal keyboard. Programs used to be started by loading the program counter register with the address of the first instruction of a program and its result used to be examined by the contents of various registers and memory locations of the machine. Therefore programming in this style caused a low utilization of both users and machine.

With the advent of input output devices such as punched cards, paper tapes and language translators (compiler) assembler) bought significant computer system utilization. Programs were coded into programming languages and then they are changed into object code (binary code) by translator and then automatically loaded into memory by a program called loader. Then the control is transferred to loaded program, the exhibition of a program begins and its result gets displayed or printed. Once in memory, the program may be re run with a different set of input data.

Let us have the look on the various Problems faced: The process of development and preparation of a program in such environment is slow and cumbersome due to serial processing and numerous manual processing.

In typical programming environment the following steps are performed.

- The source code is created in the editor by writing a user program.
- The source code is converted into binary code by the translator and
- The loader is called to load executable programs into main memory for execution. If syntax error is detected, the whole program must be restarted from the beginning.
- The next evolution was the replacement of *card - decks* with standard input output with some useful library programs, which were further linked with user program through system software called linker. While there' was a definite improvement overt machine language approach, the serial mode of operation is obviously not very efficient. This results in low utilization of resources.

Next Batch Processing has been evolved.

Batch Processing:

During the time that tapes were being mounted or programmer was operating the console, the CPU sets idle. The next step in the logical evaluation of operating system was to automate the sequencing of operations involved in program execution and in the mechanical aspects of program development Programs With similar requirements were batched together and 11m through the computer as a group.

Example:

Operator received one FORTRAN program, one COBOL program and another FORTRAN program. If he runs in that order ,he would have to load FORTRAN compiler tapes, then COBOL program Compiler and fma11y FORTRAN compiler again. If he runs the two FORTRAN programs as a batch to save time.

With batch processing utilization of system resources has improved quite a bit

Let us see,what happens when a job is stopped?

When the job is stopped, the operator would have to notice that fact by observing the console, determines why the program is stopped and then loads the card reader or paper tape reader for the next job and restarts the computer.

Problems in batch Processing

- The CPU sits idle when there is a job transition.
- Speed discrepancy between fast CPU and comparatively slow input/output devices such as card reader, printers.

The fist problem i.e. idle time of CPU can be overcomes by a small program called a resident monitor will be created, which resides always in the memory.

Resident Monitor:

It acts according to the directives given by a programmer through *control* earth-which contain commands belonging *to job control languages* such as information like marking of job's beginning and ending, commands for loading and executing programs etc...

Example:

$COB	- Execute the FORTAN compiler
$JOB	- First card of a job.
$END	- Last card of a job.
$Load	- Load program into memory.
$RUN	- Execute the user program.

Card Deck for Simple COBAL batch program

The second problem has been overcomes over the years through the technological improvement resulted in faster I/O devices. But CPU speed increased even faster. Therefore, the need was to increase the throughput and resource utilization by overlapping I/O and processing operations. Dedicated I/O processors, peripheral controllers brought a major development.

The development of Direct Memory Access(DMA) chip Was a major achievement, which directly transfer the entire block of data from its own memory buffer to main memory without intervention of CPU .DMA can transfer data between high speed I/O devices and main memory ,while the CPU is executing. CPU requires to be interrupted per block only by DMA. Apart from DMA, there are other two approaches to improve system performance by overlapping input/output and processing. These are:

- Buffering
- Spooling.

1. Buffering:

It is a method of overlapping input/output and processing of a single job. The idea is quite simple. After data has been read and the CPU starts operating on it, the input device is instructed to begin the next input immediately. The CPU and the input device are both busy. The CPU can begin the processing of the newly read data, while the input device starts to read the following data. Similarly, this can be done for output In this case; the CPU creates data that is put into buffer until an output device can accept it.

Now ,I will explain you the situation –",what happens if the CPU is fast"?

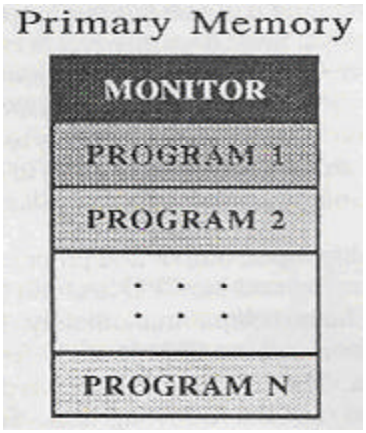
- In the case of input, the CPU finds an empty buffer and has to wait for the input device.
- In Case of output, the CPU can proceed at full speed until, eventually all system buffers are full .Then the CPU waits for the output device. This situation occurs with input/output bound jobs where the amount of input/output relation to computations very high. Since the CPU is faster than the input/output device, the speed of execution is controlled by the input/output device, not by the speed of the CPU.

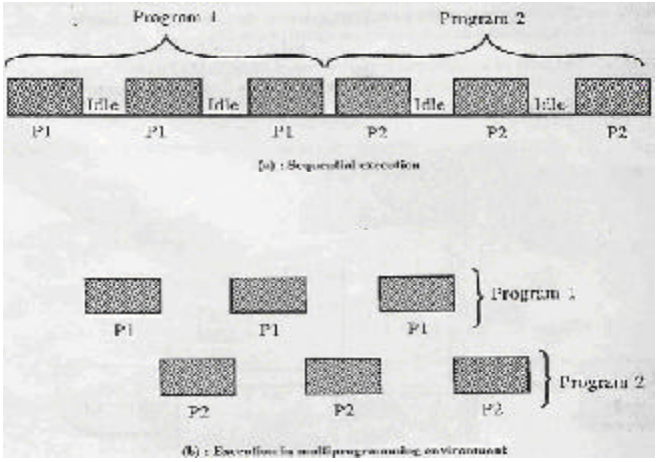
2. Spooling: It stands for simultaneous peripheral operation on line. It is essentially use the disk as a large buffer for reading and for storing output files as shown in the figure.

Spooling allows CPU to overlap the input of one job with the computation and output of other jobs. Even in a simple system, the spooler may be reading the input of one job while printing the output of a different job. Compared to buffering approach spooling is better.

Multi-Programming

Buffering and spooling improve system performance by overlapping the input, output and computation of a single job, but both of them have limitations. A single user cannot always keep CPU or I/O devices busy at all times. Multiprogramming offers a more efficient approach to increase system performance. It refers to a computer system's ability to support more than one process (program) at the same time. Multiprocessing operating systems enable several programs to nm concurrently. This is a kind of parallel processing. More number of programs competing for system resources which lead to better utilization of system resources. The idea is implemented as follows. The main memory of a system contains more than on program as shown in the figure.





Multiprogramming

The operating system picks one of the programs and start executing. During execution process program 1 may need some *I/*

O operation to complete. In a sequential Execution environment, the CPU would sit idle. In a Multiprogramming system, operating system will simply switch over the next program. When that program needs to wait for some *I/O* operation, it switches over to program 3 and so OIL if there is no other new program left in the main memory, the CPU will pass its control back to the previous programs.

Compared to operating system which supports only sequential execution, multiprogramming system requires some form of CPU and memory management strategies.

With each new generation of operating systems, you are introduced to new ways of thinking about how our computers work. To simplify things for the user, you must deploy a consistent interface in which they can do their work. It is equally important to extend this consistency to programmers, so they too can benefit. As an operating system ages, it gradually becomes burdened with a plethora of interfaces which break the simplicity of its original architecture. UNIX originally followed the "everything is a file" mantra, only to lose sight of that design with numerous task-specific APIs for transferring files (FTP, HTTP, RCP, etc.), graphics (X11, svgalib), printers (lp, lpr), etc. Plan 9, introduced in 1989, and demonstrated how even a GUI can be represented as a set of files, revitalizing the "everything is a file" idea.

Let us discuss the Types of an Operating System

1. Batch Operating System:

As discussed earlier during batch processing environment it requires grouping of similar jobs which consist of programs, data and system commands.

The suitability of this type of processing is in programs with large computation time with no need of user interaction / involvement. Some examples of such programs include payroll, forecasting, statistical analysis and large scientific number crunching programs., Users are not required to wait while the job is being processed. They can submit their programs to operators and return later to collect them.

But it has two major disadvantages:

- Non-interactive environment
- Off-line debugging

Non-interactive environment: There are some difficulties with a batch system from the point of view of programmer or user. Batch operating systems allow little or no interaction between users and executing programs. The turn around time taken between job submission and job completion in batch operating system is very high. Users have no control over intermediate results of a program. This type, of arrangement does not create flexibility in software development.

The second disadvantage with this approach is that programs must be debugged which means a programmer cannot correct bugs the moment it occurs.

Process scheduling (i.e. allocation strategy for a process to a processor), memory management file management and I/O management in batch processing are quite simple.

Jobs are typically processed in the order of submission, that is, in the first come, first served basis.

Memory is usually divided into two areas. One of them is permanently fixed for containing operating system routines and the other part contains only user programs to be executed; when one Program is over, the new program is loaded into the same area.

Since there is only one Program in the execution at a time, there is no competition for 110 devices, therefore, allocation and de-allocation for 1/0 devices is very trivial.

Access to files is also serial and there is hardly a need of Protection and file access control mechanism.

2. Multiprogramming Operating System :

Multiprogramming operating systems compared to batch operating systems are fairly sophisticated. As illustrated in figure 5, multiprogramming has a significant potential for improving system throughput and resource utilization with very minor differences. Different forms of multiprogramming operating system are multitasking, multiprocessor and multi-user operating systems. In this section, we will briefly discuss the main features and functions of these systems.

Multitasking Operating Systems:

A running state of a program is called a process or a task. A multitasking operating system (also called multiprocessing operating system) supports two or more active processes simultaneously. Multiprogramming operating system is operating system which, in addition to supporting multiple concurrent process (several processes in execution states simultaneously) allows the instruction and data from two or more separate processes to reside in primary memory simultaneously.

Note that multiprogramming implies multiprocessing or multitasking operation, but multiprocessing operation (or multitasking) does not imply multiprogramming. Therefore, multitasking operation is one of the mechanism that multiprogramming operating system employs in managing the totality of computer related resources like CPU, memory and I/O devices.

The simplest form of multitasking is called serial multitasking or context switching. This is nothing more than stopping one temporarily to work on another. If you have used sidekick, then you have used serial multitasking. While a program is running, you decide that you want to use the calculator, so you pop it and use it. When you stop using the calculator, the Program continues running.

Multiuser operating system allow simultaneous access to a computer system through or more terminals. Although frequently associated with multiprogramming, multiuser operating system does, not imply multiprogramming or multitasking. A dedicated transaction processing system such as railway reservation system that hundreds of terminals under control of a single program is an example of multiuser operating system. On the other hand, general purpose time sharing systems (discussed later in this section) incorporate features of both multiuser and multiprogramming operating system. Multiprocess operation without multiuser support can be found in the operating system of some advanced personnel computers and in real systems (discussed later).

Time Sharing System:

It is a form of multiprogrammed Operating system which operates in an interactive mode with a quick response time. The user types a request to the computer through a keyboard. The computer processes it and a response (if any) is displayed on the user's terminal. A time sharing system allows the many users to simultaneously share the computer resources. Since each action or command in a time-shared system take a very small fraction of time, only a little CPU time is needed for each user. As the CPU switches rapidly from one user to another user, each user is given impression that he has his own computer, while it is actually one computer shared among many users.

Most time sharing system use time-slice (round robin) scheduling of CPU. In this approach, Programs are executed with rotating priority that increases during waiting and drops after the service is granted. In Order to prevent a program from monopolising the processor, a program executing longer than the system defined time-slice in interrupted by the operating system and placed at the end of the queue of waiting program.

Memory management in time sharing system Provides for the protection and separation of user programs. Input/output management feature of time-sharing system must be able to handle multiple users (terminals). However, the processing of terminals interrupts are not time critical due to the relative slow speed of terminals and users. As required by most multiuser environment allocation and deallocation of devices must be performed in a manner that preserves system integrity and provides for good performance.

The words multiprogramming, multiprocessing and multitasking are often confused. There are, of course, some distinctions between these similar, but distinct terms.

The term multiprogramming refers to the situation in which a single CPU divides its time between more than one job. Time sharing is a special case of multiprogramming, where a single CPU serves a number of users at interactive terminals.

In multiprocessing, multiple CPUs perform more than one job at one time. Multiprogramming and multiprocessing are not mutually exclusive. Some mainframes and super mini computers have multiple CPUs each of which can juggle several jobs.

The term multitasking is described any system that runs or appears to run more than one application program one time. An effective multitasking environment must provide many services both to the user and to the application program it runs. The most important of these are resource management which divides the computers time, memory and peripheral devices among competing tasks and inter-process communication, which lets tasking coordinate their activities by exchanging information.

Real-time Systems:

It is another form of operating system which is used in environments where a large number of events mostly external to computer systems, must be accepted and processed in a short time or within certain deadlines. Examples of such applications are flight control, real time simulations etc. Real time systems are also frequently used in military application.

A primary objective of real-time system is to provide quick response times. User convenience and resource utilization are of secondary concern to real-time system. In the real-time system each process is assigned a certain level of priority according to the relative importance of the even it processes. The processor is normally allocated to the highest priority process among those which are ready to execute. Higher priority process usually pre-emptive execution of lower priority processes. This form of scheduling called, priority based pre- emptive scheduling, is used by a majority of real-time systems.

Memory Management:

In real-time operating system there is a little swapping of program between primary and secondary memory. Most of die time, processes remain in primary memory in order to provide quick response, therefore, memory management in real-time system is less demanding compared to other types of multiprogramming system. On the other hand, processes in real- time system tend to cooperate closely thus providing feature for both protection and sharing of memory.

I/O Management:

Time-critical device management is one of the main characteristics of real-time systems. It also provides sophisticated form of interrupt management and I/O buffering.

File Management:

The primary objective of file management in real-time systems is usually the speed of access rather than efficient utilisation of secondary storage. In fact, some embedded real-time systems do not have secondary memory. However, where provided file management of real-time system must satisfy the same requirement as those found in time sharing and other multiprogramming systems.

3. Network Operating System

A network operating system is a collection of software and associated protocols that allow a set of autonomous computers which are interconnected by a computer network to be used together in a convenient and cost-effective manner. In a network operating system, the users are aware of existence of multiple computers and can log in to remote machines and copy files from one machine to another machine.

Some of typical characteristics of network operating systems which make it different from distributed operating system (discussed in the next section) are the followings:

- Each computer has its own private operating system instead of running part of a global system wide operating system.
- Each user normally works on his/her own system; using a different system requires some kind of remote login, instead of having the operating system dynamically allocate processes to CPUs.
- Users are typically aware of where each of their files are kept and must move file from one system to another with explicit file transfer commands instead of having file placement managed by the operating system.

The system has little or no fault tolerance; if 5% of the personnel computers crash, only 5% of the users is out of business.

Network operating system offers many capabilities including:

- Allowing users to access the various resources of the network hosts

- Controlling access so that only users in the proper authorisation are allowed to access particular resources.
- Making the use of remote resources appear to be identical to the use of local resources
- Providing up-to-the minute network documentation on-line.

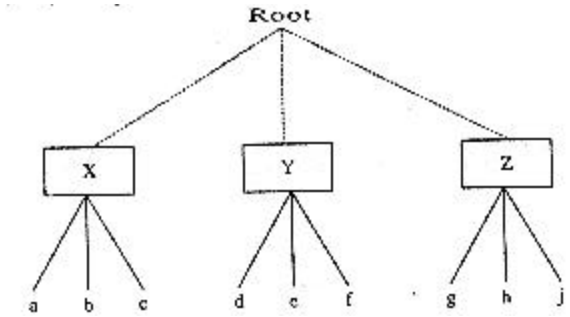
As we said earlier, the key issue that distinguishes a network operating system from a distributed one is how aware the users are of the fact that multiple machines are being used. This visibility occurs in three primary areas; file system, protection and program execution.

File System:
The important issue in file system is related to how a file is placed (accessed) on one system from another in a network. 'Mere are two important approaches to this problem.

- Running a special file transfer program
- Specifying a path name

Running a special file transfer program:
When connecting two or more systems together, the first issue that must be faced is how to access the rile system available on some other system. To deal with this issue user runs a special file transfer program that copies the needed remote file to the local machine, where they can then be accessed normally. Sometimes remote printing and mail is also handled this way. One of the best known examples of network that primarily support file transfer and mail via special programs is the UNIX's UUCP (user to user control program) program and its network USENET.

Path name specification:
The second approach in this direction is that programs from one machine can open files on another machine by providing a path name telling where the file is located.



A (virtual) subdirectory above the root directory provides access to remote tiles protection:

Execution Location

Program execution is the third area in which machine boundaries are visible in network operating systems. When a user or a running program wants to create a new process, where is the process created? At least four schemes have been used thus far. The first of these is that the user simply says "CREATE PROCESS" in one way or another, and specifies nothing about where. Depending on the implementation, this can be the best or worse Way to do it.

The second approach to process location is to allow users to run jobs on any machine by first logging in there. In this model, processes on different machines cannot communicate or exchange data, but a simple manual load balancing is possible.

The third approach is a special command that the user types at a terminal to cause a program to be executed on a specific machine. A typical command might be

remote vax4 who

to run the who program on machine vax4. In M arrangement, the environment of the new process is the remote machine. In other words, if that process tries to read or write files from its current working directory, it will discover that its working directory is on the remote machine, and that files that were in the parent process's directory are no longer present. Similarly, files written in the working directory will appear on the remote machine, not the local one.

The fourth approach is to provide the "CREATE PROCESS" system call with a parameter specifying where to run the new process, possibly with a new system call for specifying the default site. As with the previous method, the environment will generally be the remote machine. In many cases, signals and other forms of inter-process communication between processes do not work properly among processes on different machines.

Now let us see how file system protection and program execution are supported in distributed operating systems.

4. Distributed Operating System

A distributed operating system is one that looks to its users like an ordinary centralized operating system but runs on multiple independent CPUs. The key concept here is transparency. In other words, the use of multiple processors should be invisible to the user. Another way of expressing the same idea is to say that user views the system as virtual uniprocessor but not as a collection of distinct machines. In a true distributed system, users are not aware of where their programs are being run or where their files are residing; they should all be handled automatically and efficiently by the operating system.

Distributed operating systems have many aspects in common with centralized ones but they also differ in certain ways. Distributed operating system, for example, often allow programs to run on several processors at the same time, thus requiring more complex processor scheduling (scheduling refers to a set of policies and mechanisms built into the operating systems that controls the order in which the work to be done is completed) algorithms in order to achieve maximum utilisation of CPU's time.

Fault-tolerance is another area in which distributed operating systems are different. Distributed systems are considered to be more reliable than uniprocessor based system. They perform even if certain part of the hardware is malfunctioning. This additional feature, supported by distributed operating system has enormous implications for the operating system.

I will tell you the Advantages of Distributed Operating Systems

There are three important advantages in the design of distributed operating system:

1. **Major breakthrough in microprocessor technology**: Micro-processors have become very much powerful and cheap, compared with mainframes and minicomputers, so it has

become attractive to think about designing large systems consisting of small processors. These distributed systems clearly have a price/performance advantages over more traditional systems.

2. **Incremental Growth**: The second advantage is that if there is a need of 10 per cent more computing power, one should just add 10 per cent more processors. System architecture is crucial to the type of system growth, however, since it is hard to give each user of a personal computer another 10 per cent.
3. **Reliability:** Reliability and availability can also be a big advantage; a few parts of the system can be down without disturbing people using the other parts; On the minus side, unless one is very careful, it is easy for the communication protocol overhead to become a major source of inefficiency.

Now let us see how file system, protection and program execution are supported in distributed operating system.

File System:

Distributed operating system supports a single global file system visible from all machines. When this method is used, there is one directory for executable programs (in UNIX, it is bin directory), one password file and so on. When a program wants to read the password file it does something like

Open (*/etc/password", READ-ONLY)

without reference to where the file is. It is upto the operating system to locate the file and arrange for transport of data as they are needed.

The convenience of having a single global name space is obvious. In addition, this approach means that operating system is free to move files around among machines to keep all the disks generally full and busy and that the system can maintain replicated copies of files if it chooses. When the user or program must specify the machine name, the system cannot decide on its own to move a file to a new machine because that would change the (user visible) name used to access the file. Thus in a network operating system, control over file placement must be done manually by the users, whereas in a distributed operating system it can be done automatically by the system itself.

Protection:

In a true distributed system there is a unique UID for every user, and that UID should be valid on all machines without any mapping. In this way no protection problems arise on remote access to files; a remote access can be treated like a local access with the same UID. There is a difference between network operating system and distributed operating system in implementing protection issue. In networking operating system, there are various machines, each with its own user to UID mapping but in distributed operating system there is a single system wide mapping that is valid everywhere.

Program Execution:

In the most distributed case the system chooses a CPU by looking at the processing load of the machine, location of file to be used etc. In the least distributed case, the system always run the process on one specific machine (usually the machine on which the user is logged in).

An important difference between network and distributed operating system is how they are implemented. A common way to realize a networking operating system is to put a layer of software on top of the native operating system of the individual machines. For example one could write a special library package that could intercept all the system calls and decide whether each one was local or remote. Although most system calls can be handled this way without modifying kernel (kernel is that part of operating system that manages all resources of computer).

Historical Development of Operating Systems

1. Open shop

- Each user was allocated a block of time to load and run his/her program, which was input from punch cards.
- Debugging consisted of inspecting the internal machine states and patching them directly.
- Device drivers (device-specific routines), functions, compilers, and assemblers had to be explicitly loaded.

2. operator-driven shop

- The computer operator loaded the jobs and collected output.
- Users debugged programs by inspecting a core dump, which was a hexadecimal listing of the exact contents of memory.
- The operator could batch jobs or rearrange them according to priority, run time, etc.

3. Offline input/output or simple batch system

- A separate computer was used for I/O.
- Several programs were first loaded onto tape, and then the full tape was read into the main computer.
- Program output and dumps were written to tape, and then printed from the tape by the auxiliary computer.
- A small resident monitor program reset the main computer after each job, interpreted some simple command language, performed some simple accounting, and did device-independent input and output.

4. Spooling systems = multiprogrammed batch systems

- treated separately in text (Sections 1.3.2.2 and 1.4), but were developed approximately simultaneously
- Example: IBM OS/360
- spool: simultaneous peripheral operations on line
- Disks were used for intermediate storage: faster than tapes and allowed jobs to be processed in any order.
- A nucleus (or kernel) contained routines to manage processes (jobs) and device interrupts.
- Used interrupts to perform I/O (device tells computer when it is finished a task)
- Device drivers included in the nucleus
- A process (running program) requested assistance from the kernel by making a service call =system call
- A scheduler sorted incoming jobs according to priority and processor time needed
- Still used a human operator to mount data tapes needed by jobs, make some policy decisions about which jobs to run, and

to restart the resident monitor if it failed or was overwritten by a program

- Could do *multiprogramming = multitasking*: have more than one process somewhere between starting and finishing

5. **interactive multiprogramming** (**timeshared** system)

- Examples: CTSS, MULTICS, UNIX
- Users interact with the computer directly through a command language at a terminal
- A *command interpreter* defines interface
- A *session* lasts from *logon* to *logoff*
- Text editors allow users to create programs, text files, and data files online instead of with cards or tape
- User has the illusion that he/she is the only user of the computer, but there may actually be many simultaneous users
- recent PC operating systems, such as OS/2 and Windows 95, are single-user interactive multiprogrammed systems

6. **interactive uniprogramming**

- One user, one process at a time: personal computers
- Examples: CP/M (Control Program for Microcomputers), DOS (derived from Seattle Computing Product's SCP-DOS clone of CP/M — 1981)
- Processes can "terminate and stay resident" in memory, later to be reactivated by interrupts from the keyboard (primitive multiprogramming) - Large amounts of processing time can be devoted to providing a graphical user interface, since only one process is active at a time

7. **distributed computing**

- Communication between processes on different processors, e.g., e-mail, ftp, finger
- Separate computers share devices (printers)
- A user may execute processes on a different machine from the one he/she is on
- Allows load sharing: automatic movement of processes to other sites
- Increased fault tolerance for data and processes
- Tightly coupled system: processors share a main memory ... also called parallel system
- Loosely coupled system: processors have their own memory and communicate by exchanging messages ... what is usually meant by a distributed system

Release Dates for Recent Operating Systems

- UNIX - 1973
- DOS 1.0 - 1981
- MacOS - about 1984
- MacOS, System 5 - 1986
- OS/2 1.0 - 1987
- Windows3.0 - 1990
- Windows3.1 - 1991
- Windows95 - 1995
- WindowsNT - 1993

WindowsNT, v.4 - 1996

Windows98 - 1998

Windows 2000 - 2000

Review Exercise:

1. You explain the difference between Contrast Serial Processing, Batch Processing and Multiprogramming?

2. What is Buffering and Spooling?

3. List the main differences between Network operating systems and Distributed operating systems.

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems / Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

Notes

Objective:

Dear students after learning about the different types of operating systems , lets us now discuss the different functions and services of an operating system.

Today I will discuss briefly the services and Functions of an Operating System. They are listed as follows:

- Processor management
- Memory management
- Device management
- Storage management
- Application interface
- User interface

With the different types of operating systems in mind, it's time to look at the basic functions provided by an operating system.

When the power to a computer is turned on, the first program that runs is usually a set of instructions kept in the computer's **read-only memory** (ROM) that examines the system hardware to make sure everything is functioning properly. This **power-on self test** (POST) checks the CPU, memory, and basic input-output systems (BIOS) for errors and stores the result in a special memory location. Once the POST has successfully completed, the software loaded in ROM (sometimes called **firmware**) will begin to activate the computer's disk drives. In most modern computers, when the computer activates the hard disk drive, it finds the first piece of the operating system: the bootstrap loader.

The **bootstrap loader** is a small program that has a single function: It loads the operating system into memory and allows it to begin operation. In the most basic form, the bootstrap loader sets up the small driver programs that interface with and control the various hardware subsystems of the computer. It sets up the divisions of memory that hold the operating system, user information and applications. It establishes the data structures that will hold the myriad signals, flags and semaphores that are used to communicate within and between the subsystems and applications of the computer. Then it turns control of the computer over to the operating system.

The operating system's tasks, in the most general sense, fall into six categories:

- Processor management
- Memory management
- Device management
- Storage management
- Application interface
- User interface

While there are some who argue that an operating system should do more than these six tasks, and some operating-system vendors do build many more utility programs and auxiliary functions into their operating systems, these six tasks define the core of nearly all operating systems. Let's look at the tools the operating system uses to perform each of these functions.

Processor Management

The heart of managing the processor comes down to two related issues:

- Ensuring that each process and application receives enough of the processor's time to function properly.
- Using as many processor cycles for real work as is possible

The basic unit of software that the operating system deals with in scheduling the work done by the processor is either a **process** or a **thread**, depending on the operating system.

It's tempting to think of a process as an application, but that gives an incomplete picture of how processes relate to the operating system and hardware. The application you see (word processor or spreadsheet or game) is, indeed, a process, but that application may cause several other processes to begin, for tasks like communications with other devices or other computers. There are also numerous processes that run without giving you direct evidence that they ever exist. A process, then, is software that performs some action and can be controlled by a user, by other applications or by the operating system.

It is processes, rather than applications, that the operating system controls and schedules for execution by the CPU. In a single-tasking system, the schedule is straightforward. The operating system allows the application to begin running, suspending the execution only long enough to deal with interrupts and user input. **Interrupts** are special signals sent by hardware or software to the CPU. It's as if some part of the computer suddenly raised its hand to ask for the CPU's attention in a lively meeting. Sometimes the operating system will schedule the priority of processes so that interrupts are **masked** that is, the operating system will ignore the interrupts from some sources so that a particular job can be finished as quickly as possible. There are some interrupts (such as those from error conditions or problems with memory) that are so important that they can't be ignored. These **non-maskable** interrupts (NMIs) must be dealt with immediately, regardless of the other tasks at hand.

While interrupts add some complication to the execution of processes in a single-tasking system, the job of the operating system becomes much more complicated in a multi-tasking system. Now, the operating system must arrange the execution of applications so that you believe that there are several things happening at once. This is complicated because the CPU can only do one thing at a time. In order to give the appearance of lots of things happening at the same time, the operating system has to switch between different processes thousands of times a second. Here's how it happens.

- A process occupies a certain amount of RAM. It also makes use of registers, stacks and queues within the CPU and operating system memory space.
- When two processes are multi-tasking, the operating system allots a certain number of CPU execution cycles to one program.
- After that number of cycles, the operating system makes copies of all the registers, stacks and queues used by the processes, and notes the point at which the process paused in its execution.
- It then loads all the registers; stacks and queues used by the second process and allow it a certain number of CPU cycles.
- When those are complete, it makes copies of all the registers, stacks and queues used by the second program, and loads the first program.

All of the information needed to keep track of a process when switching is kept in a data package called a **process control block**. The process control block typically contains:

- An ID number that identifies the process
- Pointers to the locations in the program and its data where processing last occurred
- Register contents
- States of various flags and switches
- Pointers to the upper and lower bounds of the memory required for the process
- A list of files opened by the process
- The priority of the process

The status of all I/O devices needed by the process

When the status of the process changes, from pending to active, for example, or from suspended to running, the information in the process control block must be used like the data in any other program to direct execution of the task-switching portion of the operating system.

This process swapping happens without direct user interference, and each process gets enough CPU cycles to accomplish its task in a reasonable amount of time. Trouble can come, though, if the user tries to have too many processes functioning at the same time. The operating system itself requires some CPU cycles to perform the saving and swapping of all the registers, queues and stacks of the application processes.

If enough processes are started, and if the operating system hasn't been carefully designed, the system can begin to use the vast majority of its available CPU cycles to swap between processes rather than run processes. When this happens, it's called **thrashing**, and it usually requires some sort of direct user intervention to stop processes and bring order back to the system.

One way that operating-system designers reduce the chance of thrashing is by reducing the need for new processes to perform various tasks. Some operating systems allow for a "process-lite," called a **thread**, that can deal with all the CPU-intensive work of a normal process, but generally does not deal with the various types of I/O and does not establish structures requiring the extensive process control block of a regular process. A process may start many threads or other processes, but a thread cannot start a process.

So far, all the scheduling we've discussed has concerned a single CPU. In a system with two or more CPUs, the operating system must divide the workload among the CPUs, trying to balance the demands of the required processes with the available cycles on the different CPUs.

Asymmetric operating systems use one CPU for their own needs and divide application processes among the remaining CPUs. **Symmetric** operating systems divide themselves among the various CPUs, balancing demand versus CPU availability even when the operating system itself is all that's running.

Even if the operating system is the only software with execution needs, the CPU is not the only resource to be scheduled. **Memory management** is the next crucial step in making sure that all processes run smoothly.

Memory and Storage Management

When an operating system manages the computer's memory, there are two broad tasks to be accomplished:

- Each process must have enough memory in which to execute, and it can neither run into the memory space of another process nor be run into by another process.
- The different types of memory in the system must be used properly so that each process can run most effectively.

The first task requires the operating system to set up **memory boundaries** for types of software and for individual applications.

As an example, let's look at an imaginary system with 1 megabyte (1,000 kilobytes) of RAM. During the boot process, the operating system of our imaginary computer is designed to go to the top of available memory and then "back up" far enough to meet the needs of the operating system itself. Let's say that the operating system needs 300 kilobytes to run. Now, the operating system goes to the bottom of the pool of RAM and starts building up with the various driver software required to control the hardware subsystems of the computer. In our imaginary computer, the drivers take up 200 kilobytes. So after getting the operating system completely loaded, there are 500 kilobytes remaining for application processes.

When applications begin to be loaded into memory, they are loaded in block sizes determined by the operating system. If the block size is 2 kilobytes, then every process that is loaded will be given a chunk of memory that is a multiple of 2 kilobytes in size. Applications will be loaded in these fixed block sizes, with the blocks starting and ending on boundaries established by words of 4 or 8 bytes.

These blocks and boundaries help to ensure that applications won't be loaded on top of one another's space by a poorly calculated bit or two. With that ensured, the larger question is what to do when the 500-kilobyte application space is filled.

In most computers, it's possible to add memory beyond the original capacity. For example, you might expand RAM from 1 to 2 megabytes. This works fine, but tends to be relatively expensive. It also ignores a fundamental fact of computing - most of the information that an application stores in memory is not being used at any given moment. A processor can only access memory one location at a time, so the vast majority of RAM is unused at any moment. Since disk space is cheap compared to RAM, then moving information in RAM to hard disk can greatly expand

RAM space at no cost. This technique is called **virtual memory management**.

Disk storage is only one of the memory types that must be managed by the operating system, and is the slowest. Ranked in order of speed, the types of memory in a computer system are:

- **High-speed cache** - This is fast, a relatively small amount of memory that are available to the CPU through the fastest connections. Cache controllers predict which pieces of data the CPU will need next and pull it from main memory into high-speed cache to speed up system performance.
- **Main memory** - This is the RAM that you see measured in megabytes when you buy a computer.
- **Secondary memory**- This is most often some sort of rotating magnetic storage that keeps applications and data available to be used, and serves as virtual RAM under the control of the operating system.

The operating system must balance the needs of the various processes with the availability of the different types of memory, moving data in blocks (called **pages**) between available memory as the schedule of processes dictates.

Device Management

The path between the operating system and virtually all hardware not on the computer's motherboard goes through a special program called a **driver**. Much of a driver's function is to be the translator between the electrical signals of the hardware subsystems and the high-level programming languages of the operating system and application programs. Drivers take data that the operating system has defined as a file and translate them into streams of bits placed in specific locations on storage devices, or a series of laser pulses in a printer.

Because there are such wide differences in the hardware controlled through drivers, there are differences in the way that the driver programs function, but most are run when the device is required, and function much the same as any other process. The operating system will frequently assign high-priority blocks to drivers so that the hardware resource can be released and readied for further use as quickly as possible.

One reason that drivers are separate from the operating system is so that new functions can be added to the driver and thus to the hardware subsystems - without requiring the operating system itself to be modified, recompiled and redistributed.

Through the development of new hardware device drivers, development often performed or paid for by the manufacturer of the subsystems rather than the publisher of the operating system, input/output capabilities of the overall system can be greatly enhanced.

Managing input and output is largely a matter of managing **queues** and **buffers**, special storage facilities that take a stream of bits from a device, perhaps a keyboard or a serial port, hold those bits, and release them to the CPU at a rate slow enough for the CPU to cope with.

This function is especially important when a number of processes are running and taking up processor time. The operating system will instruct a buffer to continue taking input from the device, but to stop sending data to the CPU while the process using the input is suspended. Then, when the process needing input is made active once again, the operating system will command the buffer to send data.

This process allows a keyboard or a modem to deal with external users or computers at a high speed even though there are times when the CPU can't use input from those sources.

Managing all the resources of the computer system is a large part of the operating system's function and, in the case of real-time operating systems, may be virtually all the functionality required. For other operating systems, though, providing a relatively simple, consistent way for applications and humans to use the power of the hardware is a crucial part of their reason for existing.

Interface to the World

Application Interface

Application program interfaces (APIs) let application programmers use functions of the computer and operating system without having to directly keep track of all the details in the CPU's operation. Let's look at the example of creating a hard disk file for holding data to see why this can be important.

A programmer writing an application to record data from a scientific instrument might want to allow the scientist to specify the name of the file created. The operating system might provide an API function named **MakeFile** for creating files. When writing the program, the programmer would insert a line that looks like this:

```
MakeFile [1, %Name, 2]
```

In this example, the instruction tells the operating system to create a file that will allow random access to its data (1), will have a name typed in by the user (%Name), and will be a size that varies depending on how much data is stored in the file (2). Now, let's look at what the operating system does to turn the instruction into action.

1. The operating system sends a query to the disk drive to get the location of the first available free storage location.
2. With that information, the operating system creates an entry in the file system showing the beginning and ending locations of the file, the name of the file, the file type, whether the file has been archived, which users have permission to look at or modify the file, and the date and time of the file's creation.
3. The operating system writes information at the beginning of the file that identifies the file, sets up the type of access possible and includes other information that ties the file to the application.

In all of this information, the queries to the disk drive and addresses of the beginning and ending point of the file are in formats heavily dependent on the manufacturer and model of the disk drive.

Because the programmer has written her program to use the API for disk storage, she doesn't have to keep up with the instruction codes, data types, and response codes for every possible hard disk and tape drive.

The operating system, connected to drivers for the various hardware subsystems, deals with the changing details of the hardware - the programmer must simply write code for the API and trust the operating system to do the rest.

APIs have become one of the most hotly contested areas of the computer industry in recent years. Companies realise that programmers using their API will ultimately translate into the ability to control and profit from a particular part of the industry. This is one of the reasons that so many companies have been willing to provide applications like readers or viewers to the public at no charge. They know consumers will request that programs take advantage of the free readers, and application companies will be ready to pay royalties to allow their software to provide the functions requested by the consumers.

User Interface

A User Interface (UI**)** brings structure to the interaction between a user and the computer. In the last decade, almost all development in user interfaces has been in the area of the **graphical user interface** (GUI), with two models, Apple's Macintosh and Microsoft's Windows, receiving most of the attention and gaining most of the market share. There are other user interfaces, some graphical and some not, for other operating systems.

Unix, for example, has user interfaces called **shells** that present a user interface more flexible and powerful than the standard operating system text-based interface. Programs such as the Korn Shell and the C Shell are text-based interfaces that add important utilities, but their main purpose is to make it easier for the user to manipulate the functions of the operating system.

There are also graphical user interfaces, such as X-Windows and Gnome, which make Unix and Linux more like Windows and Macintosh computers from the user's point of view.

It's important to remember that in all of these examples, the user interface is a program or set of programs that sits as a layer above the operating system itself. The same thing is true, with somewhat different mechanisms, of both Windows and Macintosh operating systems. The core operating-system functions, the management of the computer system, lie in the **kernel** of the operating system. The **display manager** is separate, though it may be tied tightly to the kernel beneath.

The ties between the operating-system kernel and the user interface, utilities and other software define many of the differences in operating systems today, and will further define them in the future.

Let us take the overview of Additional Operating System Functions

Additional functions exist not for helping the user, but rather for ensuring efficient system operations.

Resource allocation-allocating resources to multiple users or multiple jobs running the same time.

Accounting-keep track of and record which users use how much and what kinds of computer resources for account billing or for accumulating usage statistics.

Protection - ensuring that all access to system resources is controlled.

Command-Interpreter System

Many commands are given to the operating system by control statements which deal with:

process creation and management

I/O handling

secondary-storage management

main-memory management

file-system access

protection

networking

The program that reads and interprets control statements is called variously:

control-card interpreter

command-line interpreter

shell (in UNIX) Its function is to get and execute the next command statement.

Operating System Services

Program execution – system capability to load a program into memory and to run it.

I/O operations – since user programs cannot execute I/O operations directly, the operating system must provide some means to perform I/O.

File-system manipulation – program capability to read, write, create, and delete files.

Communications – exchange of information between processes executing either on the same computer or on different systems tied together by a network. Implemented via *shared memory* or *message passing*.

Error detection – ensure correct computing by detecting errors in the CPU and memory hardware, in I/O devices, or in user programs.

List of Additional Operating System Functions

Additional functions exist not for helping the user, but rather for ensuring efficient system operations.

Resource allocation-allocating resources to multiple users or multiple jobs running at the same time.

- **Accounting-**keep track of and record which users use how much and what kinds of computer resources for account billing or for accumulating usage statistics.
- **Protection-**nsuring that all access to system resources is controlled.

In the next coming lessons we will learn these functions in detail

Review Exercise:-

1) Explain briefly the general functions of an Operating System?

_____

_____

_____

_____

_____

_____

_____

_____

2) Briefly explain the services of an Operating System?

_____

_____

Notes

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems / Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

LESSON 4:

Objective:

Today I will describe you the Operating Systems Structure

Since you all know operating system is a very large and complex software, it must be engineered carefully, if it is to be functioned properly and to be modified easily. It should be developed as a collection of several smaller modules with carefully defined inputs, outputs and functions rather than a single piece of software.

Let us now examine different types of operating systems.

1. Simple Structure:

This type of structures is not well defined .Such type of operating system are as :

- Small
- Simple.
- Limited systems.
- Grew beyond their scope.

Example 1:

MS-DOS was written to provide the most functionality in the least space.

- not divided into modules
- Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated.

MS-DOS Layer Structure

2. Layered Approach: The operating system architecture based on layered approach is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface. Higher level layers uses the functions (operations) and services of only lower-level layers.

An Operating System Layer

Example of Windows 2000.

Advantages:

- As the system is divided into layers/modules verification and debugging of the system is easy/simple.
- Easy to find and rectify the errors ,as we can find in which layer the error has occurred.
- A layer need now the operations implemented by the lower layer, thus hides the existence of certain data structures, operations and hardware.

Let us discuss the Difficulties for designing layered approach:

- The layered approach involves careful definition of the layers, since the higher level layer uses the services of the lower level layer.

For Example, the device driver for the secondary memory must be a lower level than memory management routines since memory management requires the ability to use the backing store.

- This approach tends to be less efficient than others. For example, When a user program executes an I/O operation, it executes a system call that is traped to the I/O Layer, which in turn calls the memory management layer, which in turn calls the CPU-scheduling layer, which is then passed to the hardware. At each layer , the parameters may be modified, data may need to be passed, and so on. Each layer adds overhead to the system call, thus system call takes longer time when compared to a non-layered system.

OS/2 Operating System Layer

OS/2 is the descendant of MS-DOS adds additional features such as multitasking and dual-mode operation and other addition features. In OS/2 fewer layers with more functionality are designed, providing most of the advantages of modularized code while avoiding the difficult problems of layer definition and interaction. The advantage in this type of operating system is ,direct access to low-level facilities is not allowed, providing the operating system with more control over the hardware and more knowledge of which resources each user program is using.

3. Kernel approach: The kernel is the heart of the operating system. It is the part of operating system which directly makes interface with hardware system. When the system is booted, the kernel is read into the memory. It stays in memory while the system is running .Its main functions are:

- To provide a mechanism for creation and deletion of processes.
- To provide process scheduling, memory management and I/O management.
- To provide mechanism for synchronization of process so that processes synchronize their actions.
- To provide mechanism for interprocess communication.
- The UNIX operating system is based on kernel approach .It consists of two parts:

Kernel

· **System Program**:Programs and commands call on the kernel's services. The kernel, in turn, consults its data tables as it schedules user's programs, allocates resources to the program, and manages the low-level exchange of data with the computer's hardware.

For example, when a program requests file services, the program gives the kernel a system call. The kernel oversees the accessing of the disk drive where the file resides. The kernel gets the data and transfers it to the buffer. The data is then picked up by the parts of the kernel can be configured to accommodate variations in hardware .The kernel contains a changeable set of device drivers to accommodate numerous devices.

Making a system call.
An illustration of UNIX kernel built on the system's hardware core.

With in the kernel, individual segments of programs or routines, carry out the kernel's work. The routines allocate memory resources, schedule CPU time, and manage access to system resources.

The kernel also monitors the system for error conditions and hardware problems. At higher level, the routines provide programs with entry points to kernel services. All UNIX programs use the kernel's system call.

4. **Virtual Machine:** A virtual machine takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware. A virtual machine provides an interface identical to the underlying bare hardware. The operating system creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory.

- The resources of the physical computer are shared to create the virtual machines.
- CPU scheduling can create the appearance that users have their own processor.
- Spooling and a file system can provide virtual card readers and virtual line printers.
- A normal user time-sharing terminal serves as the virtual machine operator's console.
- Protection is excellent, but no sharing possible.
- Virtual privileged instructions are trapped.
- Useful for running different OS simultaneously on the same machine.

Virtual System Models
From the user's point of view, virtual machine can be made to appear to very similar to existing real machine or they can be entirely different. An important aspect of this technique is that each user can run operating system of his own choice.

To understand this concept, let us try to understand the difference between conventional multiprogramming system and virtual machine multiprogramming. In conventional multiprogramming processes are allocated a portion of the real machines resources. The same machine resources are distributed among several processes.

In virtual multiprogramming system, a single real machine gives an illusion of several virtual machines, each having its own virtual processor, storage and I/O devices possibly with much larger capacity. Process scheduling can be used to share the CPU and make it appear that user's have their own processors. Virtual memory organization technique can create illusion of very large memory for program execution.

Lets us now cover the advantages and disadvantages of Virtual Machines

Advantages :

- The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines.
- A virtual-machine system is a perfect vehicle for operating-systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.
- The higher degree of separation between independent virtual machine aids in ensuring aids privacy and security.

Disadvantages
permits no direct sharing of resources.

- The virtual machine concept is difficult to implement due to the effort required to provide an *exact* duplicate to the underlying machine.

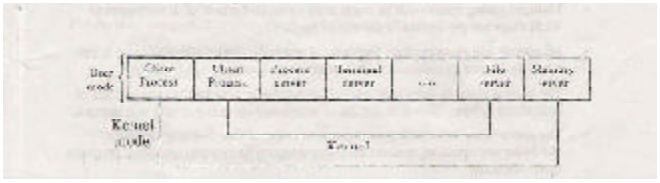
5. **Client-Server Model:** A trend in the modern operating system is to take moving code up into the higher layers even further, and remove as much as possible from the operating system, leaving a minimal kernel. The usual approach is to implement most of the operating system functions in user processes. To request a service, such as reading a block of a file, a user process (now known as client process) sends the request to a server process, which then does the work and sends back the answer.

   In this model, the kernel does all the communication between the clients and servers by splitting the operating system into parts, each of which only handles:

- File service
- Process service
- Terminal service
- Memory service.

This way, each part becomes small and manageable. Furthermore, because all the servers run as user-mode processes, and not in kernel mode, they do not have direct access to the hardware. As a consequence, if a bug in the file server is triggered, the file service may crash, but this will not usually bring the whole system down.

The Client-Server Model



Advantages :

It is adaptable in distributed systems. If a client communicates with a server by sending it messages, the client need not know whether the message is handled locally in its own machine, or whether it was sent across a network to a server on a remote machine. As far as the client is concerned, the same thing happens in both cases, request was sent and a reply came back.

Message from Client to Server

Machine 1 Machine 2 Machine 3 Machine 4

The picture painted above of a kernel that handles only the transport of messages from clients to servers and back is not completely realistic. Some operating system functions (such as loading commands into the physical I/O device registers) are difficult, if not impossible, to do from user-space programs. Here are two ways of dealing with this problem.

- Is to have some critical server processes (eg:- I/O device drivers) actually run in kernel mode, with complete access to all the hardware, but still communicate with other processes using the normal message mechanism.

Is to build a minimal amount of mechanism into the kernel, but leave the policy decisions up to servers in user space. For example, the kernel might recognize that a message sent to a certain special address means to take the contents of that message and load in into the I/O device registers for some disk, to start a disk read. In this example, the kernel would not even inspect the bytes in the message to see if they were valid or meaningful; it would just blindly copy them into the disk's device registers. (Obviously some scheme for limiting such message to authorized processes only must be used). The split between mechanism and policy is an important concept; it occurs again and again in operating

Systems in various contexts.

Exercise:

1. Contrast Multiprogramming with Client-Server Model?

_____

_____

_____

_____

2. Explain the difficulties in designing a layered approach?

_____

_____

_____

_____

3. Write the advantages and disadvantages of Virtual Machine?

_____

_____

_____

_____

4. Explain the following with an example?

- Layered Approach.

_____

_____

_____

- Kernel Approach.

_____

_____

_____

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems / Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

System Services

Windows

MGR

& GDI

Windows 2000 Kernel

Hardware Abstraction Layer (HAL)

IO

Manager

Graphics

Device

Drivers

VM

Manager

Security

Reference

Monitor

Process

Manager

User Program

(Requests Services)

UNIX System Kernel

(Provides Services)

Call

Request Service

- Type
- Details
- Data

Return

Service complete

- Status
- Data

Job N

Job 2

Virtual Machine

Operating System

Virtual

Machine 1

Virtual

Machine N

Virtual

Machine 2

Notes

Objectives:

To make you understand the basic concepts and terminology in Operating Systems.

Let us see what is an Interface?

An operating system is conceptually broken into three sets of components: a shell, a kernel and low-level system utilities. As the name implies the shell is an outer wrapper to the kernel which in turn talks directly to the hardware.

Hardware <-> Kernel <-> Shell <-> Applications

They are two types of interfaces:

- Command line interpreter
- Graphical user interfaces

A **command line interpreter** is a program which reads a line of text the user has typed and interprets this text in the context of a given system.

A **graphical user interface** (or **GUI**, often pronounced "goo-ee") is a method of interacting with a computer that uses graphical images and widgets in addition to text.

We are going to discuss the command line inteface in deal now

A **command line interface** or **CLI** is a method of interacting with a Computer by giving it lines of textual commands (ie, a sequence of characters) either from keyboard input or from a script. The computer system (ie, the program then accepting such input) then carries out the command given. The result may be textual output, or the initialization and running of some program, or a modification to the graphical output on a monitor or printer, or some change in a file system (eg, delete a file, modify a file, or create a file). The term is usually used in contrast to a graphical user interface (GUI) in which commands are typically issued by moving a pointer (eg, a mouse cursor) and/or pressing a key (ie, 'clicking', often on a key mounted on a mouse).

Examples of programs implementing command line interfaces include the (various) Unix shells, VMS' DCL (Digital Command Language), and related designs like CP/M and MS-DOS's command.com, both based heavily on DEC's RSX and RSTS operating system interfaces (which were also command line interfaces). These programs are often called command line interpreters.

There are other programs which use CLIs as well. The CAD program AutoCAD is a prominent example. In some computing environments like the Oberon or Smalltalk user interface, most of the text which appears on the screen may be used for giving commands.

Let us discuss the advantages of a command line interface over Graphical user Interface:

Even though new users seem to learn GUIs more quickly to perform common operations, well designed CLIs have several advantages:

- All options and operations are invokable in a consistent form, one 'level' away from the basic command. With most GUIs, the available operations and options often appear on different menus with differing usage patterns. They may be separated on several different menu levels as well. In either case, different applications (or utilities) may have different patterns; if so there is little advantage in either approach. Both are likely to annoy users.
- All options and operations are controlled in more or less the same way. The less in this case is a common accusation against CLIs. It should be no more difficult to understand and perform a rare operation than a common one. Few GUIs offer comparable access to the entire range of available options.

The first graphical user interface was designed by Xerox Corporation's Palo Alto Research Center in the 1970s, but it was not until the 1980s and the emergence of the Apple Macintosh that graphical user interfaces became popular. One reason for their slow acceptance was the fact that they require considerable CPU power and a high-quality monitor, which until recently were prohibitively expensive.

In addition to their visual components, graphical user interfaces also make it easier to move data from one application to another. A true GUI includes standard formats for representing text and graphics. Because the formats are well-defined, different programs that run under a common GUI can share data. This makes it possible, for example, to copy a graph created by a spreadsheet program into a document created by a word processor.

Many DOS programs include some features of GUIs, such as menus, but are not *graphics based*. Such interfaces are sometimes called *graphical character-based user interfaces* to distinguish them from true GUIs.

The use of pictures rather than just words to represent the input and output of a program. A program with a GUI runs under some windowing system (e.g. The X Window System, Microsoft Windows, Acorn RISC OS, NEXTSTEP). rating at them. This contrasts with a command line interface where communication is by exchange of strings of text.

Well-designed graphical user interfaces can free the user from learning complex command languages. On the other hand, many users find that they work more effectively with a command-driven interface, especially if they already know the command language.

Graphical user interfaces, such as Microsoft Windows and the one used by the Apple Macintosh, feature the following basic components:

ü Pointer : A symbol that appears on the display screen and that you move to select objects and commands. Usually, the pointer

appears as a small angled arrow. Text -processing applications, however, use an *I-beam pointer* that is shaped like a capital *I.*

- ü **Pointing Device:**A device, such as a mouse or trackball, that enables you to select objects on the display screen.
- ü **Icons:**Small pictures that represent commands, files, or windows. By moving the pointer to the icon and pressing a mouse button, you can execute a command or convert the icon into a window. You can also move the icons around the display screen as if they were real objects on your desk.
- ü **Desktop:** The area on the display screen where icons are grouped is often referred to as the desktop because the icons are intended to represent real objects on a real desktop.
- ü **Windows:** You can divide the screen into different areas. In each window, you can run a different program or display a different file. You can move windows around the display screen, and change their shape and size at will.
- ü **Menus:** Most graphical user interfaces let you execute commands by selecting a choice from a menu.

- CLIs often can double as scripting languages (see shell script) and can perform operations in a batch processing mode without user interaction. That means that once an operation is analyzed and understood, a 'script' implementing that understanding can be written and saved. The operation can thereafter be carried out with no further analysis and design effort. With GUIs, users must start over at the beginning every time, as GUI scripting (if available at all) is almost always more limited. Simple commands do not even need an actual script, as the completed command can usually be assigned a name (an 'alias') and executed simply by typing <*operating system*> A means of communication between a program and its user, based solely on textual input and output. Commands are input with the help of a keyboard or similar device and are interpreted and executed by the program. Results are output as text or graphics to the terminal.
- Command line interfaces usually provide greater flexibility than graphical user interfaces, at the cost of being harder for the novice to use. Consequently, some hackers look down on GUIs as designed.

System Call:

System call provides the interface to running program and the operating system.User program receives operating system services through the set of system calls.

- Generally available as assembly-language instructions.
- Languages defined to replace assembly language for systems programming allow system calls to be made directly (e.g., C, C++)

An example of system call, let us consider a simple program to copy data form one file to another. In an interactive system the following system calls are generated by the operating system.

- Prompt messages for inputting two file names and reading it from terminal.
- Open source and destination files.
- Prompt error messages in case the source file can not opened because of
- Its protection against access or destination file can not create because there is already a file with this name.
- Read the source file
- Write into the destination file.
- Display status information regarding various read/write error conditions
- Close both files after the entire file is copied

For example:

Program may find that the end of file has been reached or there was a hardware failure. The write operation may encounter various errors, depending upon the output device (no disk space, printer out of paper etc.)

From the above observation we can say that a user program takes heavy use of the operating system. All interactions between the program and its environment must occur as a result of request from that program to the operating system.

Let see the 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 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.

Now I will explain you the Working of System call.

- Obtain access to system space
- Do parameter validation
- System resource collection (locks on structures)
- Ask device/system for requested item
- Suspend waiting for device
- Interrupt makes this thread ready to run
- Wrap-up
- Return to user

There are 11 (or more) steps in making the system call **read (fd, buffer, nbytes)**

Two ways of passing data between programs.

Message Passing
Shared Memory

Examples of various system calls.

Review Exercise:

1. Compare the advantages and disadvantages of Command line Interface and Graphical user Interface?

_____

_____

_____

_____

2. What is system call? How does it work?

_____

_____

_____

_____

_____

_____

3. Explain briefly the different types of GUI Interfaces?

_____

_____

_____

_____

_____

Review Exercise

- What is an operating system?
- What are input and output devices?
- Where are operating systems found?
- What is a general purpose operating system?
- A simple operating system for a security system
- What are input output devices
- What is a single-user operating system?
- What is a multi-user operating system?
- Operating system utilities
- Operating system interfaces
- Advantages and problems of multi-user operating systems
- What is a multi-tasking operating system?

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems / Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

1.1 True/False: An operating system can be viewed as "resource allocator" to control various I/O devices and user programs.

Answer: True

1.2 True/False: "automatic job sequencing" means the System does not proceeds from one job to the next without human intervention.

Answer: False

1.3 Which of the following lists the different parts of the monitor.

a. Control card interpreter.

b. Control card interpreter, device drivers, and loader.

c Loader.

d. None of the above

Answer: B

1.4 In what ways are batch systems inconvenient for users?.

Answer:

a. Users can't interact with their jobs to fix problems.

b. Turnaround time is too long.

c. B only

d. A and B

e. A only

1.5 What were the advantages of off-line operations?

a. Main computer no longer constrained by speed of card reader.

b. Application programs used logical I/O devices instead of physical I/O devices; programs didn't have to be rewritten when new I/O devices replaced old ones.

c. Both of the above

d. None of the above

Answer: c

1.6 True/False; In a master/slave processor system, the master computer controls the actions of various slave computers.

Answer: true

1.7 True/false

MULTICS was a time-sharing system created on a large mainframe GE computer (since then taken over by Honeywell), by GE, by Bell Labs, and by faculty at MIT. It was very flexible, and oriented toward programmers. UNIX was inspired by MULTICS; but it was designed by Ritchie and Thompson in 1974 at Bell Labs for use on minicomputers. It was designed for program development, using a device-independent file system.

Answer: True

Self-Assessment Interactive Topic 2

2.1 True/False An interrupt is a hardware-generated change-of-flow within the system. An interrupt handler is summoned to deal with the cause of the interrupt; control is then returned to the interrupted context and instruction. An interrupt can be used to signal the completion of an I/O to obviate the need for device polling.

Answer: True

2.2 True/False A trap is a software-generated interrupt. A trap can be used to call operating system routines or to catch arithmetic errors.

Answer: True

2.3 True / False How is an interrupt executed? The I/O driver sends a signal through a special interrupt line to the CPU when it has finished with an I/O request.

Answer: True

2.4 True / False An interrupt vector is **a** list giving the starting addresses of each interrupt service routine.

Answer: True

2.5 True/ False Systems treat slow and fast devices differently, for slow devices, each character transferred causes an interrupt. For fast devices, each block of characters transferred causes an interrupt.

Answer: True

2.6 True/False The Introduction of base and limit registers that hold the smallest legal physical memory address, and the size of the range, respectively can prevent users from accessing other users' programs and data. As a user's job is started, the operating system loads these registers; if the program goes beyond these addresses, it is aborted. If another job starts up, these registers are reset for the new job.

Answer: True

2.7 True/False How can the operating system detect an infinite loop in a program? A timer (hardware) is added to system. Each user is allowed some predetermined time of execution (not all users are given same amount). If user exceeds these time limits, the program is aborted via an interrupt.

Answer: True

2.8 True/False: The operating system determines what mode it is in by using one bit (the monitor/user-mode bit) that gives the present state.

Answer: True

2.9 The following is a list of operations, followed by a description why each can be considered illegal.

a. Programming errors, such as illegal instruction, addressing fault.

c. Halting the computer.

d. Masking the interrupt so that none can occur. Turning on interrupts; or else job will interfere with I/O.

e. Changing mode from user to system; or else user can control system.

f. Using memory outside user area; invasion of privacy.

g. Modifying interrupt vectors in monitor; could crash system.

h. Accessing monitor memory; invasion of privacy.

Which of the following sets of operations the monitor considers illegal

a. none of the above

b. all of the above

c. only b

Answer: b

Cache Memory (bonus Discussion)

When are caches useful?

Answer: Caches are useful when two or more components need to exchange data, and the components perform transfers at differing speeds.

What problems do they solve?

Answer: Caches solve the transfer problem by providing a buffer of intermediate speed between the components. If the fast device finds the data it needs in the cache, it need not wait for the slower device.

What problems do they cause?

Answer: The data in the cache must be kept consistent with the data in the components. If a component has a data value change, and the datum is also in the cache, the cache must also be updated. This is especially a problem on multiprocessor systems where more than one process may be accessing a datum.

If a cache can be made as large as the device for which it is caching (for instance, a cache as large as a disk), why not make it that large and eliminate the device?

Answer: A component may be eliminated by an equal-sized cache, but only if: a) the cache and the component have equivalent state-saving capacity (that is, if the component retains its data when electricity is removed, the cache must retain data as well), and b) the cache is affordable, because faster storage tends to be more expensive.

Self assessment interactive Topic 3

3.1 What is the purpose of the command interpreter? Why is it usually separate from the kernel?

Answer: It reads commands from the user or from a file of commands and executes them, usually by turning them into one or more system calls. It is usually not part of the kernel since the command interpreter is subject to changes.

3.2 List five services provided by an operating system. Explain how each provides convenience to the users. Explain also in which cases it would be impossible for user-level programs to provide these services.

Answer:

- **Program executio**n. The operating system loads the contents (or sections) of a file into memory and begins its execution. A user-level program could not be trusted to properly allocate CPU time.
- **I/O operation**s. Disks, tapes, serial lines, and other devices must be communicated with at a very low level. The user need only specify the device and the operation to perform on it, while the system converts that request into device or controller specific commands. User-level programs cannot be trusted to only access devices they should have access to, and to only access them when they are otherwise unused.
- **File-system manipulatio**n. There are many details in file creation, deletion, allocation, and naming that users should not have to perform. Blocks of disk space are used by files and must be tracked. Deleting a file requires removing the name file information and freeing the allocated blocks. Protections must also be checked to assure proper file access. User programs could ensure neither adherence to protection methods nor could they be trusted to allocate only free blocks and deallocate blocks on file deletion.
- **Communication**s. Message passing between systems requires messages be turned into packets of information, sent to the network controller, transmitted across a communications medium, and reassembled by the destination system. Packet ordering and data correction must take place. Again, user programs might not coordinate access to the network device, or they may receive packets destined for other processes.

Error Detection.

Error detection occurs at both the hardware and software levels. At the hardware level, all data transfers must be inspected to ensure that data have not been corrupted in transit. All data on media must be checked to be sure they have not changed since they were written to the media. At the software level, media must be checked for data consistency; for instance, do the number of allocated and unallocated

blocks of storage match the total number on the device. There, errors are frequently process-independent (for instance, the corruption of data on a disk), so there must be a global program (the operating system) that handles all types of errors. Also, by having errors processed by the operating system, processes need not contain code to catch and correct all the errors possible on a system.

3.3 What is the main advantage of the layered approach to system design?

Answer: As in all cases of modular design, designing an operating system in a modular way has several advantages. The system is easier to debug and modify because changes affect only limited sections of the system rather than touching all sections of the operating system. Information is kept only where it is needed and is accessible only within a definedand restricted area, so any bugs affecting that data must be limited to a specific module or layer.

3.4 What is the main advantage for an operating-system designer of using a virtual-machine architecture? What is the main advantage for a user?

Answer: The system is easy to debug, and security problems are easy to solve. Virtual machines also provide a good platform for operating system research since many different operating systems may run on one physical system.

3.5 List system service functions provided for the convenience of the programmer. Tell what each does.

Answer: Program execution loads and executes programs, allows debugging I/O operations does all read and write operations File system management allows you to create, delete, open files, etc. Communications allows processes to communicate with each other Error detection CPU, hardware, instructions, device errors

3.6 List system service functions provided for efficient operation of the system.

Answer:

- Resource allocation
- Accounting
- Protection

3.7 List five or more functions to control processes and jobs.

Answer:

- Set error level
- Load/link/execute program
- Create new process
- Get/set process attributes
- Terminate process
- Wait for specific event or time
- Dump memory
- Trace instructions
- Create time profile

3.8 List eight or more functions for file manipulation.

Answer: Create, delete, open, close, read, write, and reposition files, get/set file attributes

3.9 List categories of systems programs.

Answer:

- File manipulation
- Get status information
- Modify files
- Programming language support
- Program loading/execution
- Communications
- Application programs.

3.10 What is a command interpreter? By what other names is it known?

Answer: Program that interprets the commands you type in at terminal, or enter through a batch file; gets and executes next user-specified command. Names: control card interpreter, command line interpreter, console command processor, shell.

Self assessment interactive topic 4

4.1 Several popular microcomputer operating systems provide little or no means of concurrent processing. Discuss the major complications that concurrent processing adds to an operating system.

Answer:

- A method of time sharing must be implemented to allow each of several processes to have access to the system. This method involves the preemption of processes that do not voluntarily give up the CPU (by using a system call, for instance) and the kernel being reentrant (so more than one process may be executing kernel code concurrently).
- Processes and system resources must have protections and must be protected from each other. Any given process must be limited in the amount of memory it can use and the operations it can perform on devices like disks.
- Care must be taken in the kernel to prevent deadlocks between processes, so processes aren't waiting for each other's allocated resources.

4.2 Describe the differences among short-term, medium-term, and long-term scheduling.

Answer:

- **Short-term** (CPU scheduler) - selects from jobs in memory, hose jobs which are ready to execute, and allocates the CPU to them.
- **Medium-term** - used especially with time-sharing systems as an intermediate scheduling level. A swapping scheme is implemented to remove partially run programs from memory and reinstate them later to continue where they left off.
- **Long-term** (job scheduler) - determines which jobs are brought into memory for processing.

The primary difference is in the frequency of their execution. The short-term must select a new process quite often. Long-term is used much less often since it handles placing jobs in the system, and may wait a while for a job to finish before it admits another one.

4.3 True/False: The long-term scheduler selects a group of I/O-bound jobs or a group of CPU-bound programs for subsequent activity.

Answer: False. It selects a mix of jobs for efficient machine utilization.

4.4 True / False Time sharing is many users interactively using a system "simultaneously;" each user gets a share of CPU-time, after other users have gotten their share. It uses medium-term scheduling, such as round-robin for the foreground. Background can use a different scheduling technique.

Answer: True

4.5 True/False Swapping is the process of copying a process out of memory onto a fast disk or drum, to allow space for other active processes; it will be copied back into memory when space is ample.

Answer: True

4.6 True/False context switching is the time needed to switch from one job to another

Answer: True

4.7 What two advantages do threads have over multiple processes? What major disadvantage do they have? Suggest one application that would benefit from the use of threads, and one

that would not.

Answer: Threads are very inexpensive to create and destroy, and they use very little re-sources while they exist. They do use CPU time for instance, but they don't have totally separate memory spaces. Unfortunately, threads must "trust" each other to not damage shared data. For instance, one thread could destroy data that all the other threads rely on, while the same could not happen between processes unless they used a system feature to allow them to share data. Any program that may do more than one task at once could benefit from multitasking. For instance, a program that reads input, processes it, and out-puts it could have three threads, one for each task. "Single-minded" processes would not benefit from multiple threads; for instance, a program that displays the time of day.

4.8 What resources are used when a thread is created? How do they differ from those used when a process is created?

Answer: A context must be created, including a register set storage location for storage during context switching, and a local stack to record the procedure call arguments, return values, and return addresses, and thread-local storage. A process creation results in memory being allocated for program instructions and data, as well as thread-like storage. Code may also be loaded into the allocated memory.

4.9 Describe the actions taken by a kernel to switch context

a. Among threads.

b. Among processes.

Answer:

a. The thread context must be saved (registers and accounting if appropriate), and an-other thread's context must be loaded.

b. The same as (a), plus the memory context must be stored and that of the next process must be loaded.

4.10 What are the differences between user-level threads and kernel-supported threads? Under what circumstances is one type "better" than the other?

Answer: User-level threads have no kernel support, so they are very inexpensive to create, destroy, and switch among. However, if one blocks, the whole process blocks. Kernel-supported threads are more expensive because system calls are needed to create and destroy them and the kernel must schedule them. They are more powerful because they are independently scheduled and block individually.

Notes

Self assessment interactive Topic 5

5.1 What is a CPU burst? An I/O burst?

Answer:

- CPU burst: a time interval when a process uses CPU only.
- I/O burst: a time interval when a process uses I/O devices only.

5.2 An I/O-bound program would typically have what kind of CPU burst?

Answer: Short.

5.3 What does "preemptive" mean?

Answer: Cause one process to temporarily halt, in order to run another.

5.4 What is the "dispatcher"?

Answer: Determines which processes are swapped out.

5.5 What is throughput?

Answer: Number of jobs done per time period.

5.6 List performance criteria we could select to optimize our system.

Answer: CPU use, throughput, turnaround time, waiting time, response time.

5.7 What is a Gantt chart? Explain how it is used.

Answer: A rectangle marked off horizontally in time units, marked off at end of each job or job-segment. It shows the distribution of time-bursts in time. It is used to determine total and average statistics on jobs processed, by formulating various scheduling algorithms on it.

5.8 What are the advantages of SJF? Disadvantages?

Answer: Provably optimum in waiting time. But no way to know length of next CPU burst.

5.9 What is indefinite blocking? How can it occur?

Answer: Also called starvation. A process with low priority that never gets a chance to execute. Can occur if CPU is continually busy with higher priority jobs.

5.10 What is "aging"?

Answer: Gradual increase of priority with age of job, to prevent "starvation."

5.11 What is SRTF (Shortest-Remaining-Time-First) scheduling?

Answer: A preemptive scheduling algorithm that gives high priority to a job with least amount of CPU burst left to complete.

5.12 What is round-robin scheduling?

Answer: Each job is given a time quantum slice to run; if not completely done by that time interval, job is suspended and another job is continued. After all other jobs have been given a quantum, first job gets its chance again.

5.13 True or False: Round-robin scheduling is preemptive.

Answer: True.

5.14 What is the time quantum used for?

Answer: Round-robin scheduling, to give each process the same processing time.

5.15 How should the time quantum be related to the context switch time?

Answer: Quantum should be very large compared to context switch time.

5.16 Describe the foreground-background approach.

Answer: Low priority processes run in background; high priority jobs run in foreground; background runs only when foreground is empty, or waiting for I/O.

5.17 How can multilevel queues be scheduled? Which might have priority over others?

Answer:

a. Each queue can have absolute priority over lower queues.

b. Time-slice queues can, giving each queue a certain percent of time.

5.18 What are multilevel feedback queues?

Answer: Processes move from one queue to another, depending on changes in its conditions (that is, the CPU burst may change).

5.19 What are the advantages and disadvantages of using implementation to compare various scheduling algorithms?

Answer:

- Advantages: completely accurate.
- Disadvantages: cost in coding, cost in modifying operating system, cost in modifying data structures, bad reactions from users due to changing and comparing various scheduling schemes.

5.20 List two ways several computers can work together on sharing load.

Answer:

- One computer controls others.
- Each computer acts independently..

5.21 Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

The processes are assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.

Inroder to solve the following 3 questions Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF,a nonpreemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 1) scheduling.

b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

answer :Turnaround time

	FCFS	RR	SJF	Priority
P1	10	19	19	16
P2	11	2	1	1
P3	13	7	4	18
P4	14	4	2	19
P5	19	14	9	6.

c. What is the waiting time of each process for each of the scheduling algorithms in part a?

answer Waiting time (turnaround time minus burst time)

	FCFS	RR	SJF	Priority
P1	0	9	9	6
P2	10	1	0	0
P3	11	5	2	16
P4	13	3	1	18
P5	14	9	4	1

d. Which of the schedules in part a results in the minimal average waiting time (over all processes)?

Answer: Shortest Job First

5.22 Suppose that a scheduling algorithm (at the level of short-term CPU scheduling) favors those processes that have used the least processor time in the recent past. Why will this algorithm favor I/O-bound programs and yet not permanently starve CPU-bound programs?

Answer: It will favor the I/O-bound programs because of the relatively short CPU burst request by them; however, the CPU-bound programs will not starve because the I/O-bound programs will relinquish the CPU relatively often to do their I/O.

6.1 What is the critical-section problem?

Answer: To design an algorithm that allows at most one process into the critical section at a time, without deadlock.

6.2 What is the meaning of the term *busy waitin*g? What other kinds of waiting are there? Can busy waiting be avoided altogether? Explain your answer.

Answer:

- A process is waiting for an event to occur and it does so by executing instructions.
- A process is waiting for an event to occur in some waiting queue (e.g., I/O, semaphore) and it does so without having the CPU assigned to it.
- Busy waiting cannot be avoided altogether.

6.3 Why does Solaris 2 implement multiple locking mechanisms? Under what circumstances does it use spinlocks, blocking semaphores, conditional variables, and readers–writers locks? Why does it use each mechanism?

Answer: Different locks are useful under different circumstances. Rather than make do with one type of lock which does not fit every lock situation (by adding code around the lock, for instance) it makes sense to include a set of lock types. Spinlocks are the basic mechanism used when a lock will be released in a short amount of time. If the lock is held by a thread which is not currently on a processor, then it becomes a blocking semaphore.

Condition variables are used to lock longer code sections, because they are more expensive to initiate and release, but more efficient while they are held. Readers-writers locks are used on code which is used frequently, but mostly in a read-only fashion. Efficiency is increased by allowing multiple readers at the same time, but locking out everyone but a writer when a change of data is needed.

6.4 Explain the differences, in terms of cost, among the three storage types: volatile, non-volatile, and stable.

Answer: Volatile storage is storage which fails when there is a power failure. Cache, main memory, and registers require a steady power source; when the system crashes and this source is interrupted, this type of memory is lost. Nonvolatile storage is storage which retains its content despite power failures. For example, disk and magnetic tape survive anything other than demagnetization or hardware/head crashes (and less likely things such as immersion in water, fire, etc.). Stable storage is storage which theoretically survives any type of failure. This type of storage can only be approximated with duplication.

6.5 Explain the purpose of the checkpoint mechanism. How often should checkpoints be per-formed? How does the frequency of checkpoints affect:

- System performance when no failure occurs?
- The time it takes to recover from a system crash?
- The time it takes to recover from a disk crash?

Answer: Checkpointing is done with log-based recovery schemes to reduce the amount of searching that needs to be done after a crash. If there is no checkpointing, then the entire log must be searched after a crash, and all transactions "redone" from the log. If checkpointing is used, then most of the log can be discarded. Since checkpoints are very expensive, how often they should be taken depends upon how reliable the system is. The more reliable the system, the less often a checkpoint should be taken.

6.6 Explain the concept of transaction atomicity.

Answer: A transaction is a sequence of instructions which, when executed as an atomic unit, takes the database from a consistent state to a consistent state.

Objectives

Today I will be covering the following topics given below :

- What is a Process?
- What is Process Management
- What is Context Switching
- What is a Process State?
- What is Process State Transition?

What is a process?
A container to run software in

What is a Process?

- You can talk about programs*executing* but what do you mean?
- At the very least, you are recognizing that some *program code* is resident in memory *and* the CPU is fetching the instructions in this code and executing them.
- Of course, a running program contains data to manipulate in addition to the instructions describing the manipulation. Therefore, there must also be some memory holding *data.*
- You are starting to talk of *processes* or *tasks* or even *jobs* when referring to the program code and data associated with any particular program.
- What would you need to save if you wanted to take a snapshot of a process so that you could put it aside for a short period, and then resume its execution later?

Process Management

This topic deals with handling the many programs that may be in main memory at once

Introduction to Process Management

A process is a program in execution. In general, a process will need certain resources-such as CPU time, memory, files, and I / O devices-to accomplish 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 of these processes can potentially execute concurrently.

The operating system is responsible for the following activities in connection with process management: the creation and deletion of both user and system processes; the scheduling of processes, and the provision of mechanisms for synchronization, communication, and deadlock handling for processes.

A process is more than the program code plus the current activity. A process generally also includes the process *stack* containing temporary data(such as subroutine parameters, return addresses, and temporary variables), and a *data section* containing global variables.

Process is allocated resources (such as main memory) and is available for scheduling.

A process is not the same as a program. Each process has a state which includes:

- a program counter: the location of the next instruction to be executed.
- value of all registers (or stack)
- values of all variables, including things such as file pointers (where to start the next read from an input file or where to put the next write to an output file).

For example, can have several emacs processes running simultaneously, each has a distinct state, but all processes may be executing the same machine code.

Note: A program by itself is not a process; a program is a *passive* entity, such as the contents of a file stored on disk, whereas a process is an *active* entity, with a program counter specifying the next instruction to execute.

The Process Model

Even though in actuality there are many processes running at once, the OS gives each process the *illusion* that it is running alone.

- **Virtual time**: The time used by just this processes. Virtual time progresses at a rate independent of other processes. Actually, this is false, the virtual time is typically incremented a little during systems calls used for process switching; so if there are more other processors more "overhead" virtual time occurs.
- **Virtual memory**. The memory as viewed by the process. Each process typically believes it has a contiguous chunk of memory starting at location zero. Of course this can't be true of all processes (or they would be using the same memory) and in modern systems it is actually true of no processes (the memory assigned is not contiguous and does not include location zero).

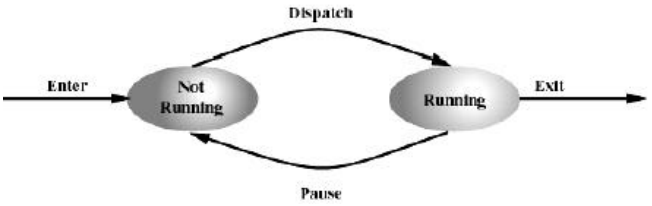
Think of the individual modules that are input to the linker. Each numbers its addresses from zero; the linker eventually translates these relative addresses into absolute addresses. That is the linker provides to the assembler a virtual memory in which addresses start at zero.

Virtual time and virtual memory are examples of abstractions provided by the operating system to the user processes so that the latter "sees" a more pleasant virtual machine than actually exists.

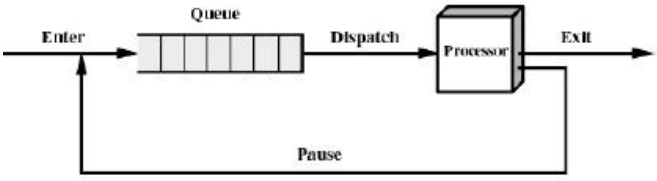
Two-State Process Model

Process may be one of two states

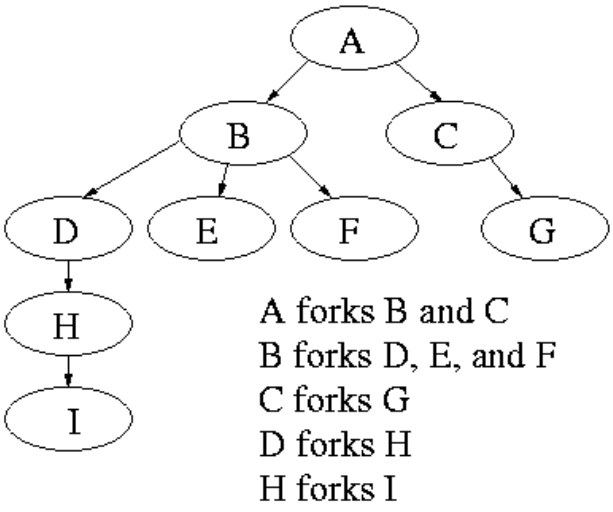
- Running
- Not– running



Process Not Running State



Process Hierarchies

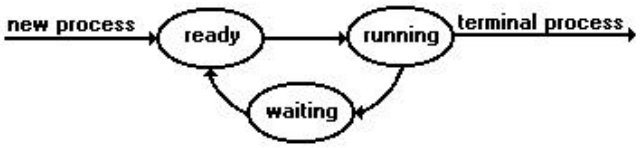


Modern general purpose operating systems permit a user to create and destroy processes.

- In unix this is done by the **fork** system call, which creates a **child** process, and the **exit** system call, which terminates the current process.
- After a fork both parent and child keep running (indeed they have the *same* program text) and each can fork off other processes.
- A process tree results. The root of the tree is a special process created by the OS during startup.
- A process can *choose* to wait for children to terminate. For example, if C issued a wait() system call it would block until G finished. Old or primitive operating system like MS-DOS are not multiprogrammed so when one process starts another, the first process is *automatically* blocked and waits until the second is finished.

Process State

As a process executes, it changes *state.* The state of a process is defined in part by that process's current activity. Each process may be in one of three states:



Process State Diagram

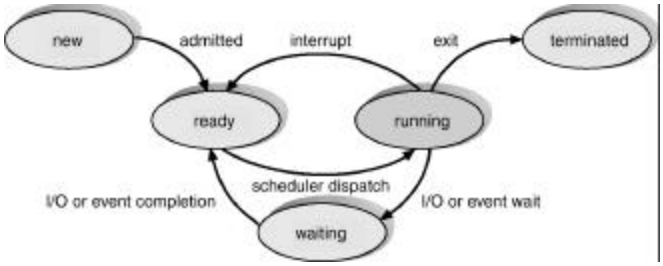
- **Running**. Instructions are being executed.
- **Waiting**. The process is waiting for some event to occur(such as an I/O completion).
- **Ready**. The process is waiting to be assigned to a processor

These names are rather arbitrary, and vary between operating systems. The states they represent are found on all systems, however. It is important to realize that in a single-processor system, only one process can be running at any instant. Many processes may be ready and waiting, however.

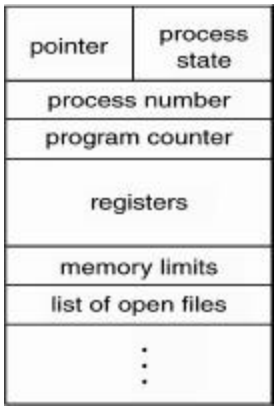
The PCB serves as the repository for any information that may vary from process to process.

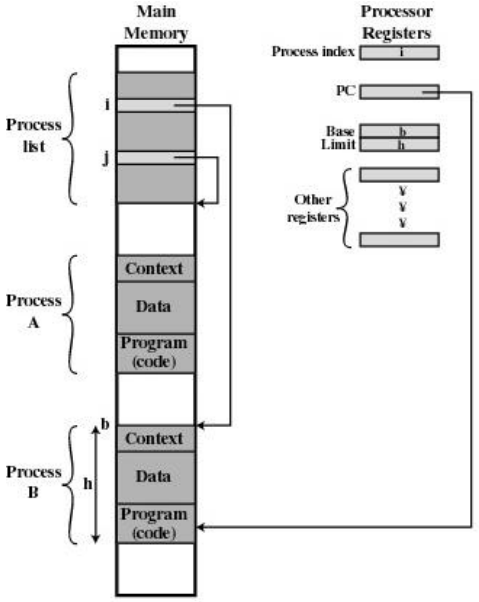
Process State

- As a process executes, it changes *state*
- **New**: The process is being created.
- **Running**: Instructions are being executed.
- **Waiting**: The process is waiting for some event to occur.
- **Ready**: The process is waiting to be assigned to a process.
- **Terminated**: The process has finished execution.



Process Control Block (PCB)





Information associated with each process.

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

Process Control Block

Each process is represented in the operating system by its own process control block (PCB). A PCB is a data block or record containing many pieces of the information associated with a specific process, including

- **Process State**. The state may be new, ready, running, waiting, or halted.
- **Program Counter**. The counter indicates the address of the next instruction to be executed for this process.
- **CPU State**. This includes the contents of general purpose registers, index registers, stack pointers, and any condition-code information. Along with the program counter, this state of information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
- **CPU Scheduling Information**. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information**. This information includes limit registers or page tables.
- **I/O Status Information**. The information includes outstanding I/O requests and a list of open files.

The PCB serves as the repository for any information that may vary from process to process.

Process Implementation

The operating system represents a process primarily in a data structure called a Process Control Block (PCB). You'll see Tack Control Block (TCB) and other variants. When a process is created, it is allocated a PCB that includes

- CPU Registers
- Pointer to Text (program code)
- Pointer to uninitialized data
- Stack Pointer
- Program Counter
- Pointer to Data
- Root directory

Default File Permissions

- Working directory
- Process State
- Exit Status
- File Descriptors
- Process Identifier (pid)
- User Identifier (uid)
- Pending Signals
- Signal Maps

Other OS-dependent information

These are some of the major elements that make up the process context. Although not all of them are directly manipulated on a context switch.

Context

The context, in this definition, refers to the contents of the CPU registers. Remember from earlier when you were talking about the instruction execution cycle and interrupt handling? The context is the stuff on the CPU which needs to be saved so that the CPU can restart execution at the current point at some later date (usually after an interrupt).

Process

A process is a program in execution. A program is just a collection of code. For example, you have the code for Netscape Communicator on your hard-drive. When you run that program by double clicking on its icon your operating system copies that code into RAM, sets up a whole lot of data structures (including

a context) and starts executing it. That's a process. A process is the collection of all the data structures the operating system users to track what the process is doing and where it is up to. It can be said that a process consists of two different sections _ a thread of execution. This is the context. The collection of CPU registers. Where the process is currently up to and the state of the CPU. Resource ownership A process owns resources. This includes resources such as memory, files, semaphores, and signals. The operating system must always be able to find where the process is and what resources it owns. That's usually the job of the Process Control Block (PCB).

Context Switching

- switch between runnable processes
- remove the running process from the processor and replace it with another runnable process

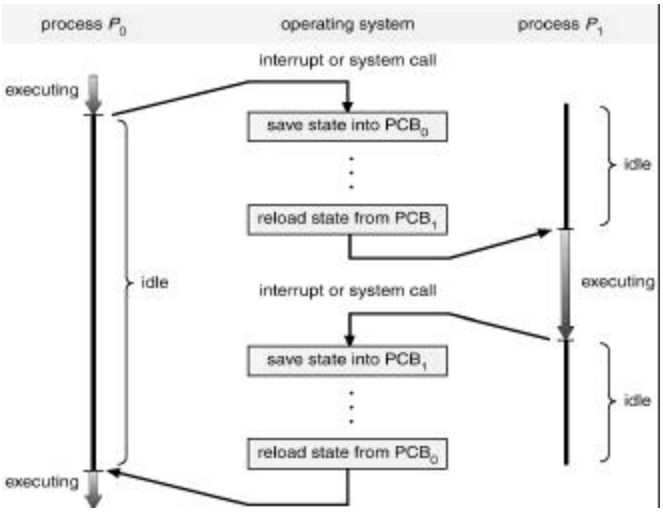
Steps in context switch:

1. enter the privileged state
2. copy the register values to a safe place, such as a stack (stored in memory)
3. load registers for the new context
4. re-enter user state

Contact Switch is the act of switching from one process to another is somewhat machine- dependent. A general outline is:

- the OS gets control (either because of a timer interrupt or because the process made a system call.
- Operating system processing info is updated (pointer to the current PCB, etc.)
- Processor state is saved (registers, memory map and floating point state, etc)
- This process is replaced on the ready queue and the next process selected by the scheduling algorithm
- The new process's operating system and processor state is restored
- The new process continues (to this process it looks like a block call has just returned, or as if an interrupt service routine (not a signal handler) has just returned

CPU Switch From Process to Process

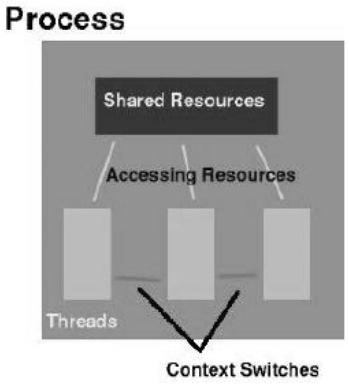


The CPU can be switched from process to process

Let us now discuss Context switch versus Process switch

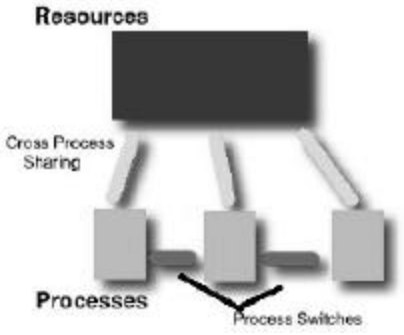
The difference between a context switch and a process switch.

Context switch



A context switch is where you replace the current context (contents of the CPU registers) with a new context. A context switch is usually hardware supported. This means that at least part of a context switch is performed by the hardware. This means that a context switch is very, very fast. A context switch usually happens when an interrupt occurs. Just because a context switch has occurred it doesn't mean that a process switch will occur.

Process switch



A process switch is where the current running process is replaced with a new process.

A process switch usually has some very limited hardware support. Most of the activity in a process switch must be performed by software, the operating system. Some of the decisions which have to be made during a process switch can be quite complex. This means that a process switch can be very slow. A process switch always includes at least one context switch, usually two. Remember, operating systems are interrupt driven, a process switch happens in response to an interrupt. That's one context switch, current process to interrupt handler. Eventually, the operating system returns the CPU to the new running process. The second context switch, operating system to new current process.

Threads

Threads are lightweight processes. They improve performance by weakening the process abstraction. A process is one thread of control executing one program in one address space. A thread may have multiple threads of control running different parts of a program in one address space.

Because threads expose multitasking to the user (cheaply) they are more powerful, but more complicated. Thread programmers have to explicitly address multithreading and synchronization (which is the topic of our next unit).

User Threads

User threads are implemented entriely in user space. The programmer of the thread library writes code to synchronize threads and to context switch them, and they all run in one process. The operating system is unaware that a thread system is even running.

User-level threads replicate some amount of kernel level functionality in user space. Examples of user-level threads systems are Nachos and Java (on OSes that don't support kernel threads).

Becuase the OS treats the running process like any other there is no additional kernel overhead for user-level threads. However, the user-level threads only run when the OS has scheduled their underlying process (making a blocking system call blocks all the threads.)

Kernel Threads

Some OS kernels support the notion of threads and schedule them directly. There are system calls to create threads and manipulate them in ways similar to processes. Synchronization and scheduling may be provided by the kernel.

Kernel-level threads have more overhead in the kernel (a kernel thread control block) and more overhead in their use (manipulating them requires a system call). However the abstraction is cleaner (threads can make system calls independently).

Examples include Solaris LWPs, and Java on machines that support kernel threads (like solaris).

Review Exercise

1. Discuss the role of a process in process management.

_____

_____

_____

_____

_____

2. Draw a process state transition diagram using five states and explain the interpretation of each transition.

_____

_____

_____

_____

_____

3. Explain how the READY and BLOCKED queues would represent the presence of processes in these states.

_____

_____

_____

_____

_____

_____

_____

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems / Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

What is a Process?

A container to run software in

Notes

LESSON 8:

Objectives

- You will be able to know what operations are performed on Processes?
- You will Understand about Process Scheduling Criteria.

Now I will start with an Operations on Processes

You hopefully now know that the mix of processes present in the system is dynamic, with the operating system attempting to manage the mix in order to obtain good resource use efficiency and good user reactions! I have yet to address the issues involved in process creation and deletion.

To manage the process mix, the operating system must be able to:

- Setup a process i.e. create it by marshalling all necessary resources and placing it in the new queue; and
- Delete a process i.e. reallocating the various resources back to the operating system's internal inventory data structures

Process Creation

- Process creation is obviously critical to the operating system, starting right at the boot up stage where it launches its own daemons and other service provider processes.
- At the core is a *process creation* system call, which obtains resources either from the operating system or from the resources already allocated to the process making the call - these resources are organized into a PCB and this new process enters the new queue.
- Using UNIX as an example operating system, we say that a child processes has been created. A child process will possess a prototype of int main() so it will return a result to its parent.
- A hierarchical (tree) structure of processes is created.
- Once a parent creates a child process, a number of execution possibilities exist:
- The parent may immediately enter a wait state for the child to finish — on UNIX, see the man pages for {wait, waitpid, wait4, wait3};
- The parent could immediately terminate;
- Both may continue to execute.
- If the parent happens to terminate before the child has returned its value, then the child will become a zombie process and may be listed as such in the process status list!
- Once a parent creates a child process, a number of memory possibilities exist:
- The child can have a duplicate of the parent's address space - as each process continues to execute, their data spaces will presumably diverge;
- The child can have a completely new program loaded into its address space;
- In UNIX, the pid t fork (void) system call can be used to create a new child with a duplicate address space.

Let us now discuss Process Creation using different models

There are two main models of process creation - the fork/exec and the spawn models. On systems that support fork, a new process is created as a copy of the original one and then explicitly executes (exec) a new program to run. In the spawn model the new program and arguments are named in the system call, a new process is created and that program runs directly.

Fork is the more flexible model. It allows a program to arbitrarily change the environment of the child process before starting the new program. Typical fork pseudo-code looks like:

```
if ( fork() == 0 ) {
    /* Child process */
    change standard input
    block signals for timers
    run the new program
}
else {
    /* Parent process */
    wait for child to complete
}
```

Any parameters of the child process's operating environment that must be changed must be included in the parameters to spawn, and spawn will have a standard way of handling them. There are various ways to handle the proliferation of parameters that results, for example AmigaDOS (R) uses tag lists - linked lists of self-describing parameters - to solve the problem.

The steps to process creation are similar for both models. The OS gains control after the fork or spawn system call, and creates and fills a new PCB. Then a new address space (memory) is allocated for the process. Fork creates a copy of the parent address space, and spawn creates a new address space derived from the program. Then the PCB is put on the run list and the system call returns.

An important difference between the two systems is that the fork call must create a copy of the parent address space. This can be wasteful if that address space will be deleted and rewritten in a few instruction's time. One solution to this problem has been a second system call, vfork, that lets the child process use the parent's memory until an exec is made. We'll discuss other systems to mitigate the cost of fork when we talk about memory management.

Which is "better" is an open issue. The tradeoffs are flexibility vs. overhead, as usual

I will explain you the various steps used for Process Termination

- Once a process executes its final instruction, a call to exit() is made.

- Even if the user did not program in a call to exit(), the compiler will have appended one to int main().
- The operating system then flushes any I/O buffers.
- All resources such as physical and virtual memory, I/O buffers, and open files are deallocated and returned to the operating system.
- The final result of the process from its int main() is returned to the parent, with a call to wait() if necessary.
- A parent may terminate execution of children processes.
- Reasons for a call to abort() include the child exceeding its allocated resources, the process may no longer be required, or the parent is exiting and the operating system may not allow a child to exist without a parent.

Scheduling Queues

As processes enter the system, they are put into a *job queue*. This queue consists of all processes residing on mass storage awaiting allocation of main memory. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the *ready queue*. This list is generally a linked-list. A ready-queue header will contain pointers to the first and last process control blocks in the list. Each PCB has a pointer field that points to the next process in the ready queue.

There are also other queues in the system. When a process is allocated the CPU, it executes for awhile and eventually either quits or waits for the occurrence of a particular event, such as the completion of an I/O request. In the case of an I/O request, such a request may be to disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. Thus, the process may have to wait for the disk. The list of processes waiting for a particular I/O device is called a *device queue*. Each device has its own device queue.

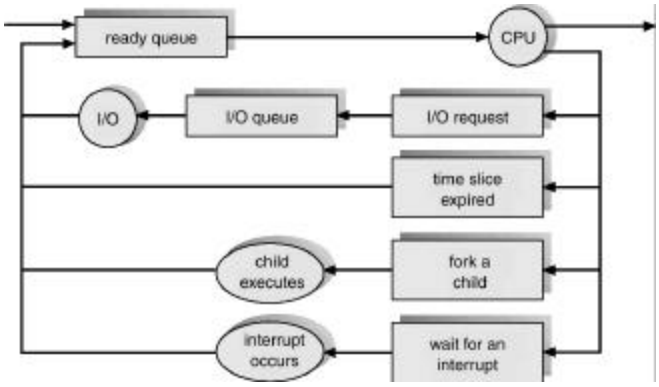
A common representation for a discussion of process scheduling is a *queueing diagram*, shown below. Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

A new process is initially put into the ready queue. It waits in the ready queue until it is selected for execution and is given the CPU. Once the process is allocated the CPU and is executing, one of 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 process and wait for its' termination.
- The process could be forcibly removed from the CPU, as a result of an interrupt, and be put back into the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back into the ready queue. A process continues this cycle until it terminates, at which time it exits from the system.

Now you will learn about the representation of Process Scheduling



CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.
- CPU scheduling decisions may take place when a process:
- Switches from running to waiting state.
- Switches from running to ready state.
- Switches from waiting to ready.
- Terminates.

Scheduling under 1 and 4 is non-preemptive.

All other scheduling is preemptive.

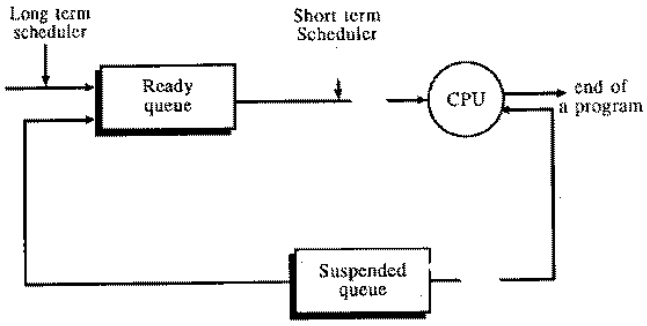
Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency –** time it takes for the dispatcher to stop one process and start another running.

Types of Schedulers

In this subsection we describe three types of schedulers: **long term, medium term** and **short term schedulers** in terms of its objectives, operating environment and relationship to other schedulers in a complex operating system environment.

Long term scheduler: Sometimes it is also called job scheduling. This determines which job shall be admitted for immediate processing.



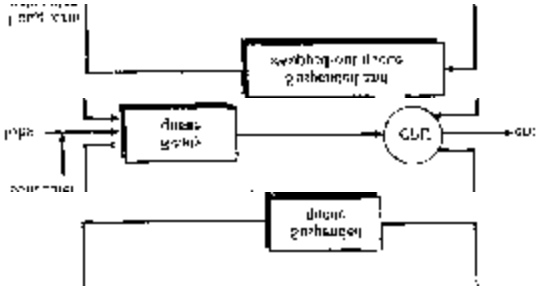
Long term and short term schedule

There are always more processes than it can be executed by CPU operating System. These processes are kept in large storage devices like disk for later processing. The long term scheduler selects processes from this pool and loads them into memory. In memory these processes belong to a ready queue. Queue is a type of data structure which has been discussed in course 4. Figure 3 shows the positioning of all three type of schedulers. The short term scheduler (also called the CPU scheduler) selects from among the processes in memory which are ready to execute and assigns the CPU to one of them. The long term scheduler executes less frequently.

If the average rate of number of processes arriving in memory is equal to that of departuring the system then the long- term scheduler may need to be invoked only when a process departs the system. Because of longer time taken by CPU during execution, the long term scheduler can afford to take more time to decide which process should be selected for execution. It may also be very important that long term scheduler should take a careful selection of processes i.e. processes should be combination of CPU and I/O bound types. Generally, most processes can be put into any of two categories: CPU bound or I/O bound. If all processes are I/O bound, the ready queue will always be empty and the short term scheduler will have nothing to do. If all processes are CPU bound. no process will be waiting for I/O operation and again the system will be unbalanced. Therefore, the long term scheduler provides good performance by selecting combination of CPU bound and I/O bound process.

Medium term scheduler:

Most of the processes require some I/O operation. In that case, it may become suspended for I/O operation after running a while. It is beneficial to remove these process (suspended) from main memory to hard disk to make room for other processes. At some later time these process can be reloaded into memory and continued where from it was left earlier. Saving of the suspended process is said to be swapped out or rolled out. The process is swapped in and swap out by the medium term scheduler. The figure 4 shows the positioning of the medium term scheduler.



Medium term scheduler

The medium term scheduler has nothing to do with the suspended processes. But the moment the suspending condition is fulfilled, the medium term scheduler get activated to allocate the memory and swap in the process and make it ready for commenting CPU resources. In order to work properly, the medium term scheduler must be provided with information about the memory requirement of swapped out processes which is usually recorded at the time of swapping and stored in die related process control block. In term of the process state transition diagram (figure 1) the medium term scheduler controls suspended to ready transition of swapped processes.

The short term scheduler:

It allocates processes belonging to ready queue to CPU for immediate processing. Its main objective is to maximize CPU requirement. Compared to the other two scheduler it is more frequent It must select a new process for execution quite often because a CPU executes a process only for few millisecond before it goes for I/O operation. Often the short term scheduler executes at least once very 10 millisecond. If it takes 1 millisecond to decide to execute a process for 10 millisecond, the 1/(10+1) = 9% of the CPU is being wasted simply for scheduling the work. Therefore. it must be very fast.

In terms of the process state transition diagram it is in charge of ready to running state transition

Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)

Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Discussion

- Discuss in brief various operations of Processes

- Explain the need of a dispatcher

- Briefly explain the Process Scheduling Criteria

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems / Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective /Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

Notes

Objectives

Today I will teach you the Scheduling Concepts and various Scheduling Algorithms.

Scheduling Concepts

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The idea is quite simple. 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 normally sit idle while the process waited for the completion of the event. In a multiprogramming system, several processes are kept in memory at a time. When one process has to wait, the operating system takes the CPU away from that process and gives it to another process. This pattern continues. Every time one process has to wait, another process may take over use of the CPU.

The benefits of multiprogramming are increased CPU utilization and higher throughput, which is the amount of work accomplished in a given time interval.

Scheduling Criteria

- The aim of a scheduler algorithm is to allocate the CPU time resource in some optimal manner. The definition of optimality determines the end result — we can consider:
- **CPU utilization** i.e. the proportion of time that the CPU is doing work;
- Throughput i.e. the number of processes (or jobs) completed per unit time (not useful unless the jobs are similar in complexity);
- **Turnaround time** i.e. the total elapsed time from when the job was submitted to when it was complete, including execution time, IO wait time, ready-to-run queue wait time, and all other overheads;**CPU burst**: the amount of time the process uses the processor before it is no longer ready

Types of CPU bursts:

- short bursts-process I/O bound (i.e. vi)

Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue to be allocated the CPU. There are several scheduling algorithms which you will examine in this section.

A major division among scheduling algorithms is that whether they support **pre-emptive or non-preemptive scheduling** discipline. A scheduling discipline is non-preemptive if once a process has been given the CPU, the CPU cannot be taken away from that process. A scheduling discipline is pre-emptive if the CPU can be taken away.

Preemptive scheduling is more useful in high priority process which requires immediate response. For example in Real time system the consequence of missing one interrupt could be dangerous.

In non-preemptive systems, jobs are made to wait by longer jobs, but the treatment of all processes is fairer. The decision whether to schedule preemptive or not depends on the environment and the type of application most likely to be supported by a given operating system

First-Come, First-Served Scheduling

By far the simplest CPU scheduling algorithm is the *first-come, first- served (FCFS)* algorithm. With this scheme, the process that requests the CPU is allocated the CPU first. The implementation of the FCFS policy is easily managed with a First-In-First-Out (FIFO) queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the ready queue. The FCFS scheduling is simple to write and understand.

The FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it wants to release the CPU, either by terminating or by requesting I/O. The FCFS algorithm is particularly troublesome for time-sharing systems. Where it is important that each user get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

Example:

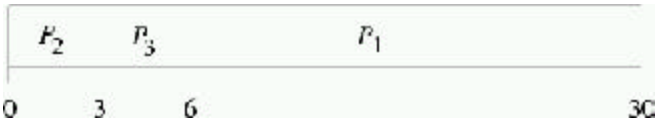
Process	Burst Time
P_1	24
P_2	3
P_3	3

Suppose that the processes arrive in the order: P 1 , P 2 , P 3 The Gantt chart for the schedule is:

- Waiting time for P1 = 0; P2 = 24; P3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

Suppose that the processes arrive in the order: P2, P3 , P1.

- The Gantt chart for the schedule is:



- Waiting time for P1 = 6; P2 = 0; P3 = 3
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case.

- Convoy effect: short process behind long process

Shortest-Job-First Scheduling

A different approach to CPU scheduling is the shortest-job-first (SJF) algorithm.(The term shortest process first is not used because most people and textbooks refer to this type of scheduling discipline as shortest-job-first) This algorithm associates with each process the length of the next CPU burst. When the CPU is available, it is assigned to the process that has the next smallest CPU burst. If two processes have the same length CPU burst, FCFS scheduling is used to break the tie.

The SJF algorithm is provably *optimal*, in that it gives the minimum average waiting time for a given set of processes. Given two processes, with one having a longer execution time than the other, it can be shown that moving a short process before a long process decreases the waiting time of the long process. Consequently the *average* waiting time decreases.

The real difficulty with the SJF algorithm knows the length of the next CPU request. For a long-term (job) scheduling in a batch system, you can use the process time limit. Thus, users are motivated to estimate the process time limit accurately, since a lower value may mean faster response. (Too low a value will cause a "time-limit-exceeded" error and require resubmission.) SJF scheduling is used frequently in process scheduling.

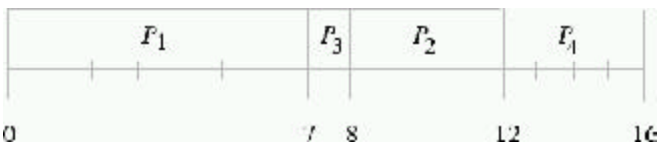
Although the SJF algorithm is optimal, that is, not other algorithm can deliver better performance; it cannot be implemented at the level of short-term CPU scheduling. There is no way to know the length of the next CPU burst. One approach is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but you may be able to predict its value. We expect that the next CPU burst will be similar in length to the previous ones. Thus, by computing an approximation of the length of the next CPU burst, you can pick the process with the shortest predicted CPU burst.

The SJF algorithm may be either 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 will preempt 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.

Example

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4 ·

SJF (nonpreemptive)



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

Priority Scheduling

The SJF algorithm 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. Equal-priority processes are scheduled in FCFS order.

An SJF algorithm is simply a priority algorithm where the priority(p) is the inverse of the (predicted) next CPU burst(t): $p = 1/t$. The larger the CPU burst, the lower the priority and vice versa.

Note that you discuss scheduling in terms of *high* and *low* priority. Priorities are generally some fixed range of numbers, such as 0 to 7, or 0 to 4095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In the text, we assume that low numbers represent high priority.

Priority scheduling can be either 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 the 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. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. Generally, one of two things will happen. Either the process will eventually be run (At 2 am. Sunday, when the system is finally lightly loaded) or the computer system will eventually crash and lose all unfinished low-priority processes.

A solution to the problem of indefinite blockage of low-priority processes is *aging*. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 0(high) to 127(low), you could decrement a waiting process's priority by 1 every minute. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed. In fact, it would take no more than 2 hours and 7 minutes for a priority 127 processes to age to priority 0 processes.

Round-Robin Scheduling

The round-robin (RR) scheduling algorithm is designed especially for time-sharing systems. A small unit of time, called a time quantum or time-slice is defined. A time quantum is generally from 10 to 100 milliseconds. 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, you keep the ready queue as a first-in, first-out (FIFO) queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first

process, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, it the CPU burst of the currently running process is greater than 1 time quantum; the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the *tail* of the ready queue. The CPU scheduler will then select the next process from the ready queue.

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row. If a process's CPU burst exceeds 1 time quantum, that process os *preempted* and is put back in the ready queue. The RR scheduling algorithm is inherently preemptive.

The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is very large (infinite), the RR policy is the same as the FCFS policy. If the time quantum is very small (say 10 milliseconds), the RR approach is called *processor sharing*, and appears (in theory) to the users as though each of *n* processes has its own processor running at 1/*n* the speed of the real processor.

For operating systems, you need to consider the effect of context switching on the performance of RR scheduling. Let use assume that you have only 1 process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead. If the quantum is 6 time units, however, the process will require 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then 9 context switches will occur, slowing the execution of the process accordingly.

Thus, you want the time quantum to be large with respect to the context switch time. If the context switch time is approximately 5 percent of the time quantum, then about 5 percent of the CPU time will be spent in context switch.

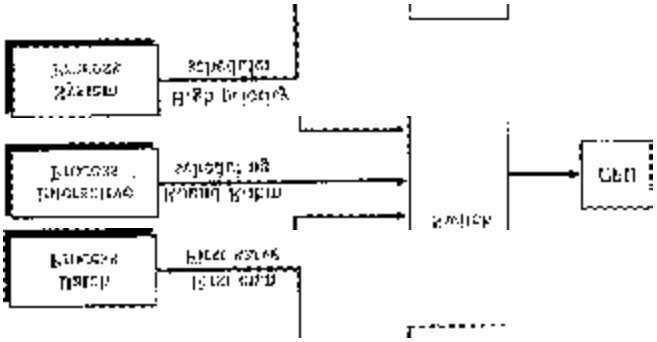
Summary of CPU Scheduling implementations

FCFS	inherently non-preemptive
SJF	preemptive or non-preemptive
Priority	preemptive or non-preemptive
Round-Robin	inherently preemptive

Multilevel Queue Scheduling

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

A multilevel queue scheduling algorithm partitions the ready queue into separate queues as shown below. Processes are permanently assigned to one queue, generally based on some property of the process, such as memory size or process type. Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes. A RR algorithm might schedule the foreground process queue, while the background an FCFS algorithm schedules queue. In addition, there must be scheduling between the queues. This is commonly a fixed-priority preemptive scheduling. For example, the foreground queue may have an absolute priority over the background queue.



Multiple queue Scheduling

Multilevel Feedback Queue Scheduling

Normally, in a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on entry to the system. Processes do not move between queues. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but is inflexible.

Multilevel feedback queue scheduling, however, 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 to a higher-priority queue. This is a form of aging that would prevent starvation.

In general, a multifeedback queue scheduler is defined by the following parameters:

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher- priority queue
- The method used to determine when to demote a process to a lower-priority queue
- The method used to determine which queue a process will enter when that process needs service

The definition of a multilevel feedback queue scheduler makes it the most general CPU scheduling algorithm. It can be configured to match a specific system under design. Unfortunately, it also

requires some means of selecting values for all the parameters to define the best scheduler. Although a multilevel feedback queue is the most general scheme, it is also the most complex.

Discussions

- What would be the effect, using the FCFS scheme, if the running process got stuck in an infinite CPU loop?

_____

_____

_____

_____

_____

_____

- With respect to the Round Robin scheduling scheme, discuss the factors, which determine the ideal value for the time quantum.

_____

_____

_____

_____

_____

_____

- The following series of processes with the given estimated run-times arrives in the READY queue in the order shown. For FCFS and SJF scheduling policies, calculate the waiting time and the wait-time / run-time ratio of each process. Comment on results.

Job	Est. run time
1	10
2	50
3	2
4	100
5	5

_____

_____

_____

_____

_____

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems /Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

Objectives

Hello ,today you will learn about the following -

- Scheduling Mechanism.
- Various Process Scheduling algorithms.

Scheduling Mechanisms

Time multiplexing of the CPU by the multiple processes simultaneously loaded into RAM. Enqueuer. Ready list (ready queue) of pointers to process descriptors. Context switcher saves all register values of currently running process in an area in its process descriptor. Dispatcher chooses process to run next from the ready queue based on the policy in effect. Context switcher loads the registers from the process descriptor of the process chosen by the dispatcher.

Voluntary release of the CPU when a process yields the CPU, requests a resource that cannot be immediately granted, or makes an IO system call. Danger of infinite loops.

Involuntary release of the CPU. Limits negative effect of process infinite loop to just that process. Interval timer device generates interrupts periodically. The interrupt handler code for this device can call the CPU scheduler to schedule some other process to run. Preemptive scheduling.

Context switches take 5-10 microseconds, depending on memory speed and how many registers there are in the CPU.

Context switch time is overhead and slows the system down. The hardware determines context switch time. The time a process spends in the ready queue before running on the CPU slows down that process. The scheduling policy determines this time, not the hardware. If the process is interactive, the user might sense this time if it is long.

Scheduling Policies (Strategies)

Factors: predictable performance, equitable sharing, optimizes performance for certain classes of jobs (batch, interactive, real time).

Lots of theoretical work done assuming a collection of processes in the ready queue, no more processes show up in the ready queue, and process total CPU needs and IO needs are known in advance. This is all unrealistic so this theoretical work is of theoretical interest only.

Definitions: *service time* is the total CPU time needed by a process to complete, *wait time* is the time spent in the ready queue before getting the CPU for the first time, *turnaround time* is the total time from first entering the ready queue (process creation) to leaving the running state for the last time (process termination).

Batch systems try to minimize average turnaround time (maximize throughput or jobs completed per minute). Timesharing systems try to minimize the wait time (also called the response time).

Non preemptive Strategies

First-come-first-served (FCFS).

Suppose five processes have CPU needs of 350, 125, 475, 250, 75 and are run in that order. The average turnaround time is (350 + (350+125) + (350+125+475) + (350+125+475+250) + (350+125+475+250+75)) * (1/5) = 4250/5 = 850. The average wait time is (0 + (350) + (350+125) + (350+125+475) + (350+125+475+250)) * (1/5) = 2975/5 = 595.

Shortest-job-next (SJN).

The five processes with CPU needs of 350, 125, 475, 250, 75 will be run in the order 75, 125, 250, 350, 475 The average turnaround time is (75 + (75+125) + (75+125+250) + (75+125+250+350) + (75+125+250+350+475)) * (1/5) = 2800/5 = 560. The average wait time is (0 + (75) + (75+125) + (75+125+250) + (75+125+250+350)) * (1/5) = 1525/5 = 305. SJN minimizes average wait time at the expense of increased variance of waiting times. Starvation of large jobs is possible if new arrivals to the ready queue with small CPU needs are run first even though the larger jobs have been waiting longer.

Priority scheduling

External factors are used to determine which process gets the CPU next, for example, faculty before students, deans before faculty, etc.

Deadline scheduling.

A program digitizing music and writing a CD must be scheduled carefully so that the buffer of digitized music to be written does not empty. If that were to happen, the CD is ruined.

Preemptive Strategies

Round robin (time slicing).

Widely used. Goal is to provide equitable CPU sharing in a timesharing interactive environment at the expense of considerable context switch overhead.

Suppose five processes have CPU needs of 350, 125, 475, 250, 75, are run in that order for time slices of 50. Assume first that the context switch time is zero. The average turnaround time is (1100 + 550 + 1275 + 950 + 475) * (1/5) = 4350/5 = 870, comparable to FCFS. The average wait time is (0 + 50 + 100 + 150 + 200) * (1/5) = 500/5 = 100, favorably low.

Now suppose the context switch time is not zero but 10 (time slice is still 50). The average turnaround time is (1320 + 660 + 1535 + 1140 + 565) * (1/5) = 5220/5 = 1044, a substantial increase. The average wait time is (0 + 60 + 120 + 180 + 240) * (1/5) = 600/5 = 120.

Multiple-level queues.

Foreground ready queue and background ready queue. Foreground queue is scheduled round robin. Background ready queue is not scheduled unless foreground ready queue is empty and is scheduled round robin with a much larger time slice if the foreground queue is empty.

Multiple-level feedback queues.

CPU-bound ready queue and IO-bound ready queue. OS categorizes processes as they run and switches processes between the two ready queues dynamically. The IO-bound ready queue is scheduled before the CPU-bound queue to keep the IO devices busy.

There might be a real-time ready queue that is scheduled before the other two.

Discussion

- Now you explain the concept of a priority used in scheduling. Why is priority working usually chosen for real-time processes?

_____
_____
_____
_____
_____
_____
_____
_____

- Give your Comments on the principal disadvantages of each of these scheduling methods:
- FCFS
- SJF
- RR

_____
_____
_____
_____
_____
_____
_____
_____

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill / Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems / Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

Today I will be covering the following objectives.

- Introduction to Cooperating process.
- You will be able to know about Inter-Process Communication
- Basic concept of Inter-Process Communication and Synchronization

Basic Concepts of Concurrency

Concurrent Process: I discussed the concept of a process earlier in this unit. The operating system consists of a collection of such processes which are basically two types:

Operating system processes: Those that execute system code and the rest being user processes, those that execute user's code. All of these processes can potentially execute in concurrent manner. Concurrency refers to a parallel execution of a program.

A concurrent program specifies two or more sequential programs (a sequential program specifies sequential execution of a list of statements) that may be executed concurrently as parallel processes. For example, an airline reservation system that involves processing transactions from many terminals has a natural specifications as a concurrent program in which each terminal is controlled by its own sequential process. Even when processes are not executed simultaneously, it is often easier to structure as a collection of cooperating sequential processes rather than as a single sequential program.

A simple batch operating system can be viewed as 3 processes -a reader process, an executor process and a printer process. The reader reads cards from card reader and places card images in an input buffer. The executor process reads card images from input buffer and performs the specified computation and store the result in an output buffer. The printer process retrieves the data from the output buffer and writes them to a printer Concurrent processing is the basis of operating system which supports multiprogramming.

The operating system supports concurrent execution of a program without necessarily supporting elaborate form of memory and file management. This form of operation is also known as multitasking. Multiprogramming is a more general concept in operating system that supports memory management and file management features, in addition to supporting concurrent execution of programs.

Cooperating Process

Concurrent process executing in the operating system may be either independent process or cooperating process. A process is independent if cannot affect or affected by another process executing in the system. Any process that doesn't share any data with any other process is independent. A process is cooperating if it can affect or be affected by the process executing in the system. Any process that shares data with other process is a cooperating process.

Advantage of Cooperating process

- **Information sharing**: Several users may be interested in the same piece of information.
- **Computation speedup**: If we want particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with others.
- **Modularity**: Dividing the system functions into separate process or threads.
- **Convenience**: Even an individual user may have many task on which to work at one time.

Let us consider the producer-consumer problem, A producer is the process that is consumed by a consumer process. A compiler produce a assembly code that is consume by the assembler. A producer is one item and the consumer is the another item. The producer and consumer must be synchronized so that consumer does not consume an item that has yet been produced.

This can be done in following ways

- **unbounded buffer** places no practical limit on the size of the buffer.
- **bounded buffer** assumes that there is a fixed buffer size.

- Bounded Buffer - Shared-Memory Solution

Using Shared data

```
const int n = 5;    //Buffer Size
int item;           //may be of any data type
int buffer[n];      //array to hold items
int in = 0;         //indexes for placement and
int out = 0;        //reading of items in buffer
```

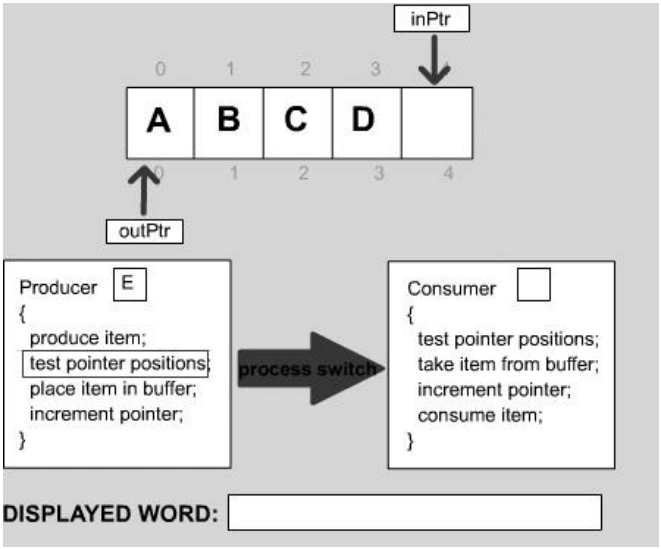
Producer process

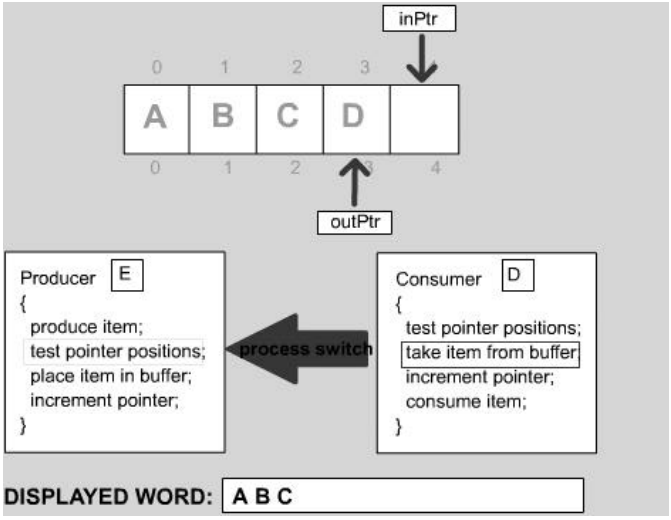
```
·           for (;;)
  {
    /*Produce Item  */
    nextp = nextp + 1;
      /*Test Pointer Position*/
    while(((in+1) % n)== out)  //if in pointer catches up
    {                          //to out pointer, wait for out
      /*do nothing*/           //pointer to move on
    }

    /*Place Item In Buffer*/
    buffer[in] = nextp;

    /*Increment Pointer*/
    in = (in+1)%n;
  }
```

Consumer process

```
for (;;)
{
```

```
/*Test Pointer Position*/
while (in == out)       //if out pointer catches up
{                       //to in pointer, wait for in
 /*do nothing*/         //pointer to move on
}

/*Take Item From Buffer*/
nextc = buffer[out];

/*Increment Pointer*/
out = (out+1)%n;

/*Consume Item*/
cout << nextc ;
}
```





Now I will explain you Inter-process Communication (IPC)

There are a number of applications where processes need to communicate with each other. Some examples of inter-process communication include:

When a process prints a table it communicates with operating system processes.An airline agent runs a program (processes) in the reservation system, which communicates with others about:

What flights and seats are available.Which process has access to critical information (semaphores).

Two user processes on mhc communicate when they implement the “talk” function.Knowledge sources in a “blackboard system” communicate their hyphotheses.

Processes can communicate by passing information to each other via shared memory or by message passing.

Why Do Processes Intercommunicate?

- Often a problem is broken into several stages, each handled by a process, that passes information to the next stage.
- Sometimes a package is broken up into several parts (e.g for an accounting package: inventory, credits, debits, invoicing, payroll). Each part will need to pass/obtain information to/from another part (e.g sales affect inventory etc.).
- There are many methods of intercommunicating information between processes.

Files

- Files are the most obvious way of passing information. One process writes a file, and another reads it later. It is often used for IPC.

Processes can communicate by passing information to each other via shared memory or by message passing.

Shared Memory

When processes communicate via shared memory they do so by entering and retrieving data from a single block of physical memory that designated as shared by all of them. This memory may be a single bit or a vast array. Each process has direct access to this block of memory (see Figure).

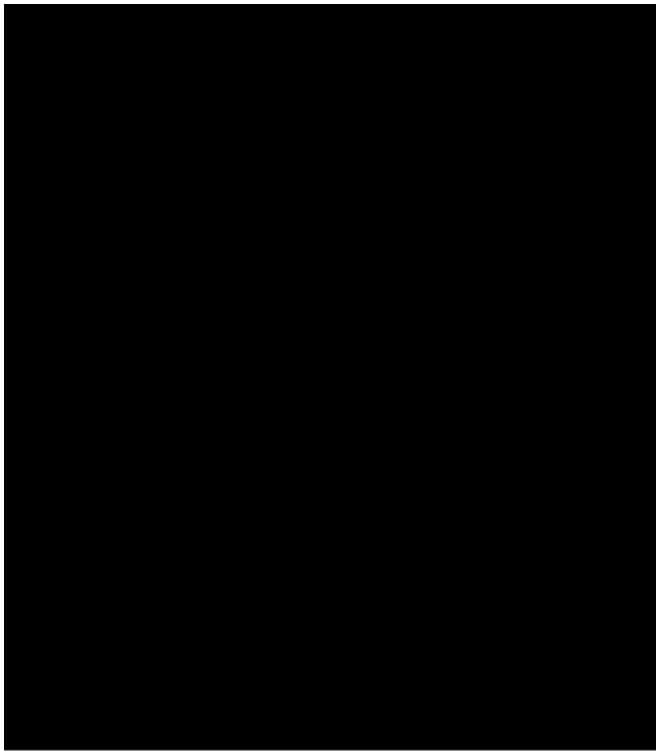


Figure when processes communicate via shared memory; they do so by having direct access to the same block of physical memory.

This block of memory may be a section of RAM. In this case each process might view this block of memory as an array. Clearly, we must be aware that this is a shared resource and, as a consequence, it is subject to the critical section problem. We will talk more about critical sections later… but this introduces the issue:

Only one program can have access to a particular array element at a time. Otherwise trouble is possible. In some cases, it is possible for every process to inspect the element at the same time, but no process should have access to that element while another process is in a section of the code that may change that element based on the value of that element. This section of code is called a **critical section**.

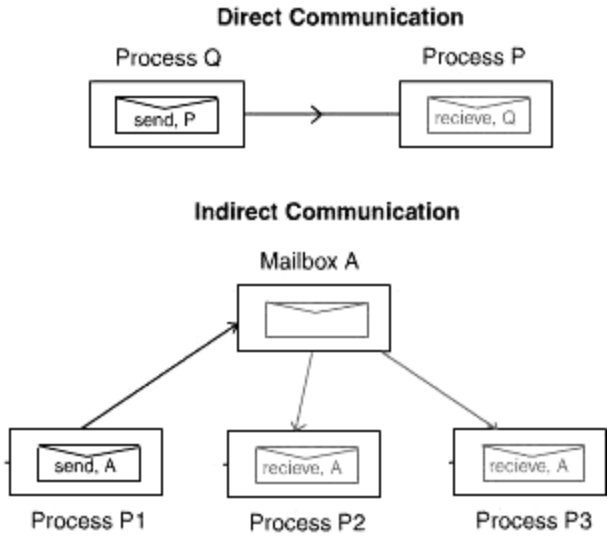
Another example of shared memory is when processes share the same disk file or directory. This is the same situation, essentially. Shared files are used for applications where the amount of memory to be shared is very large (eg. large databases) and the method of memory access not random or not quickly changing. Shared RAM is the choice when communicating large amounts of data when the access tends to be randomly organized.

Examples of the use of shared memory:

Semaphores. All processes share the integer used for a semaphore. **Buffers**. A process doing I/O shares a buffer with the OS. **Blackboard**. The "blackboard" in a "blackboard system" is a shared bank of memory.

Message Passing

Message passing is a more indirect form of communication. Rather than having direct access to a block of memory, processes communicate by sending and receiving packets of information called **messages**. These messages may be communicated **indirectly or directly**. **Indirect message** passing is done via a mailbox. **Direct message** passing is done via a link between the two communicating processes.



In both cases the messages themselves are sent via the operating system. The processes do not have **direct access** to any memory used in the message passing process.

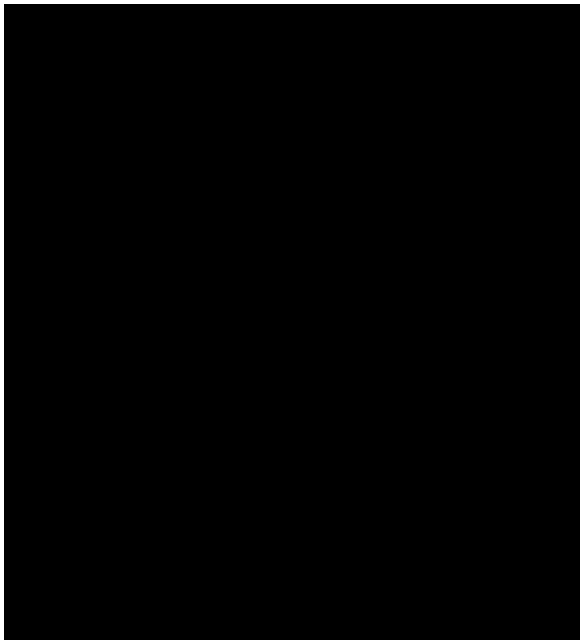


Figure Processes may communicate by sending messages to each other via a mailbox scheme. In this kind of communication individual processes do not have direct access to the memory used as mailboxes.

In the indirect scheme processes create mailboxes and use them much as we use the mailbox at home (see Figure). The sending process sends a message to a mailbox and the receiving process receives the message by extracting it from the mailbox. This communication process is implemented using a set of commands:

create(A). This command creates a mailbox A.
send(A, messag). Send message to mailbox A
receive(A, message). Receive a message from mailbox A.

Direct message passing process requires that a direct link be established between the two communicating processes (see Figure 5.3). These links are, of course, buffered so the link is still indirect in this sense. The communication path is direct, however. Direct message passing is implemented using the following commands:

create-link (P,Q). Creates a link between processes P and Q.
send(P, message). Sends a message to process P via the link.
receive(Q, message). Receives a message from process Q via the link.

Both UNIX and Windows NT support indirect communications. Both include, for example, pipes (named and unnamed) and socket based communications. Pipes are usually associated with communications between processes within one computer system … although Windows NT does support pipes between processes in different computer systems.

Socket based communications can take place between processes in the same or different computer systems, as long as the two computer systems are connected by a TCP / IP network connection.

Anonymous pipe communication calls (examples).

Unix
pipe(...) - creates a pipe.
read(...) - read from a pipe
write(...) - write to a pipe
close(...) - close one of the pipe sections.
Windows NT
CreatePipe(...) - creates a pipe with two handles.
SetStdHandle(...) - establishes a handle as the "standard handle"
GetStdHandle(...) - retreives the standard handle.
ReadFile(...) - reads from the input named by a handle.
WriteFile(...) - writes to the output named by a handle.
CloseHandle(...) - closes the resource referred to by a handle.

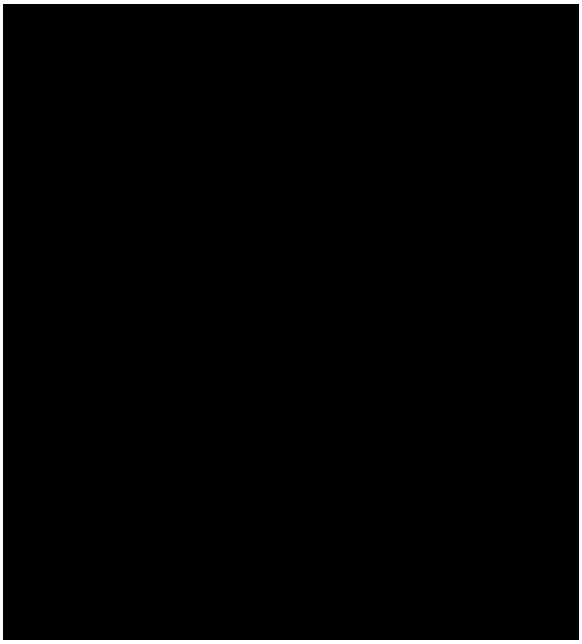


Figure Direct message passing takes place over direct links between processes, rather than via mailboxes.

Message passing can be used, among other things, for communications between processes running in different machines connected together via a network. Shared memory is not possible in this situation.

Let us discuss Basic Concepts of Inter-process Communication and Synchronization

In order to cooperate concurrently executing processes must communicate and synchronize. Inter-process communication is based on the use of **shared variables** (variables that can be referenced by more than one process) or **message passing**.

Synchronization is often necessary when processes communicate. Processes are executed with unpredictable speeds. Yet to communicate one process must perform some action such as setting the value of a variable or sending a message that the other detects. This only works if the events perform an action or detect an action are constrained to happen in that order. Thus one can view synchronization as a set of constraints on the ordering of events. The programmer employs a synchronization mechanism to delay execution of a process in order to satisfy such constraints.

To make this concept clearer, consider the batch operating system again. A shared buffer is used for communication between the leader process and the executor process. These processes must be synchronized so that, for example, the executor process never attempts to read data from the input if the buffer is empty. The next sections are mainly concerned with these two issues.

Mutual Exclusion

Processes frequently need to communicate with other processes. When a user wants to read from a file, it must tell the file process what it wants, then the file process has to inform the disk process to read the required block.

Processes that are working together often share some common storage that one can read and write. The shared storage may be in main memory or it may be a shared file. Each process has segment of code, called a critical section, which accesses shared memory or files. The key issue involving shared memory or shared files is to find way to prohibit more than one process from reading and writing the shared data at the same time. What we need is mutual

Exclusion :

some way of making sure that if one process is executing in its critical section, the other processes will be excluded from doing the same thing. Now I present algorithm to support mutual exclusion. This is applicable for two processes only.

Buffering

1· Queue of messages attached to the link; implemented in one of three ways.

2. Zero capacity 0 messages Sender must wait for receiver (rendezvous).

3. Bounded capacity finite length of n messages Sender must wait if link full. Unbounded capacity infinite length Sender never waits

Review Exercise

What is Concurrent Process?

_____

_____

_____

_____

_____

_____

_____

Explain the Basic for Inter-process Communication

_____

_____

_____

_____

_____

_____

_____

_____

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems / Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

Objectives that I will be covering today are as

- You will be able to know about Process Synchronization.
- You will be able to concept and functions of Process Synchronization.

Process Synchronization

- Background
- The Critical Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Critical Regions
- Monitors
- Synchronization in Solaris 2
- Atomic Transaction

Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Shared memory solution to bounded buffer problem allows at most n - 1 items in buffer at the same time. A solution, were all *N* buffers are used is not simple.
    - Suppose that we modify the producer consumer code by adding a variable *counter*, initialized to 0 and incremented each time a new item is added to the buffer.

Bounded-Buffer

Producer/Consumer Problem - One process produces items (e.g. disk requests, characters, print files) and another process consumes.

Assume a fixed "buffer" space.
Producer must stop when buffer is full.
Consumer must wait until buffer is nonempty.
Algorithm

· **Shared data type item**

```
char item;      //can be of any data type
char buffer[n];
int in = 0;
int out = 0;
int counter = 0;
```

· **Producer process**

```
do
{
 /*Produce Item */
 nextp = nextp + 1 ;

 /*Test Counter*/
 while(counter ==n)
 {
  /*do nothing*/
 }

 /*Place Item In Buffer*/
 buffer[in] = nextp;

 /*Increment Pointer*/
 in = (in+1)%n;

 /*Increment Counter*/
 counter = counter+1;
} while(true);
```

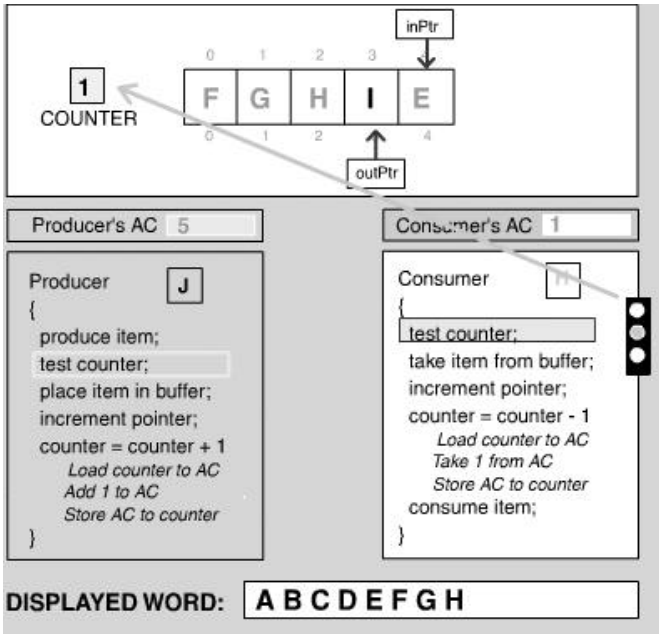
· **Consumer process**

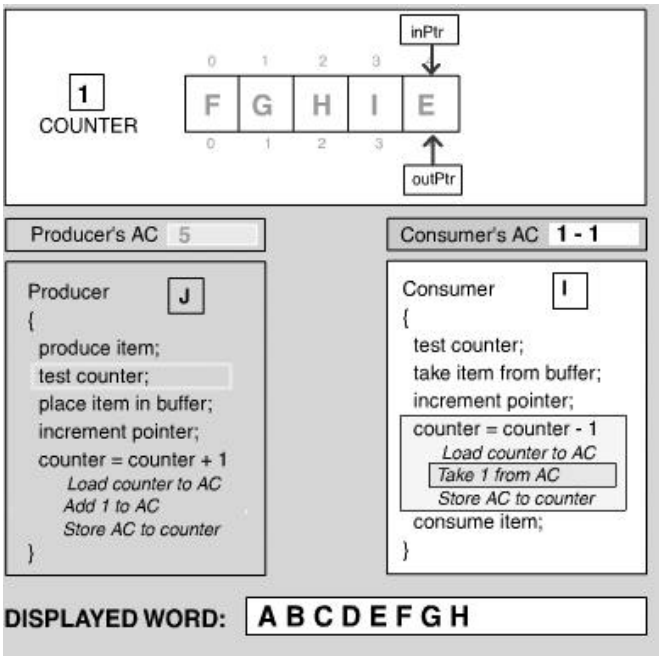
```
do
{
 /*Test Counter*/
 while (counter==0)
 {
  /*do nothing*/
 }

 /*Take Item From Buffer*/
 nextc = buffer[out];

 /*Increment Pointer*/
 out = (out+1)%n;

 /*Decrement Counter*/
 counter = counter - 1;

 /*Consume Item*/
 cout << nextc ;
} while (true)
```

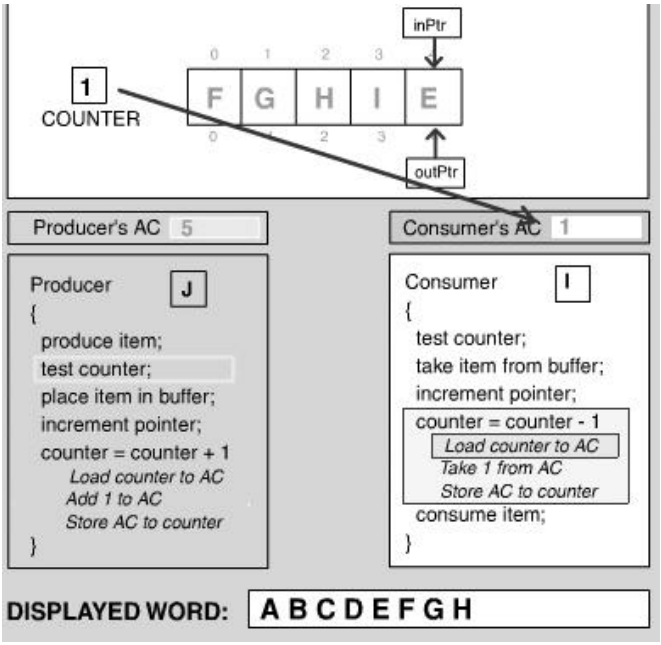
- The statements:
    - counter = counter +1;
    - counter = counter 1;

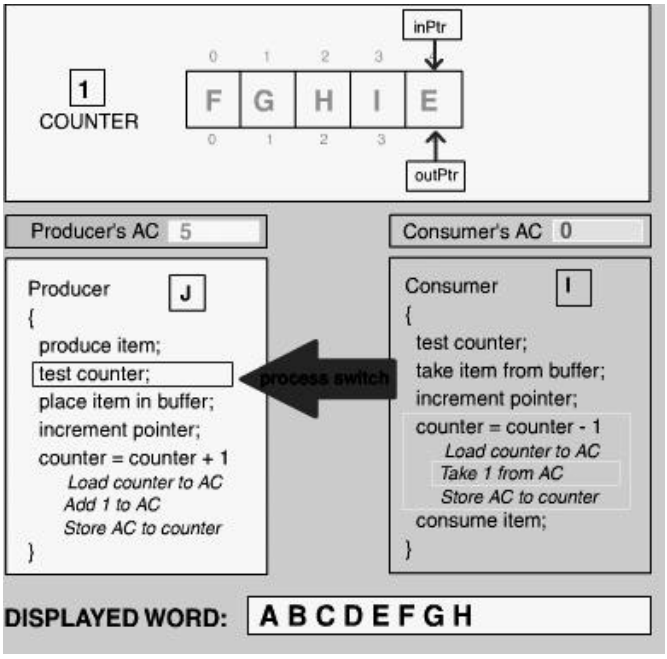
  must be executed *atomically*.

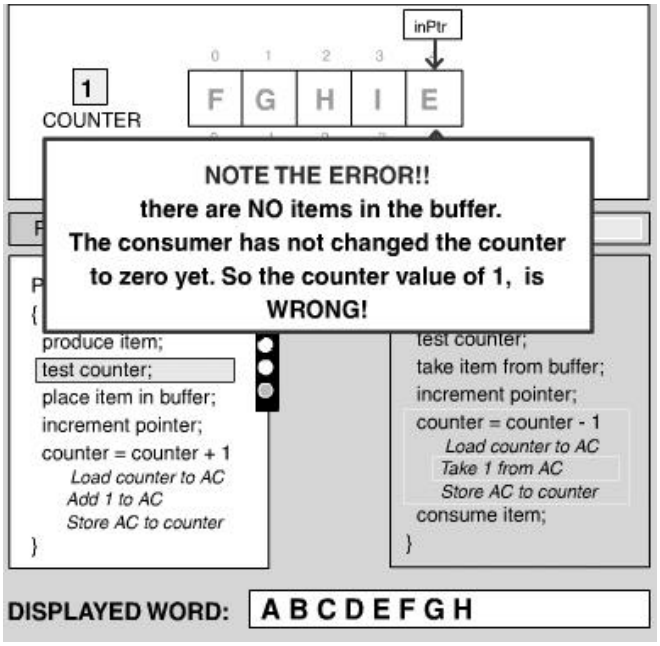
The bounded-buffer with counter solution suffers from problems associated with atomicity and mutual exclusion.

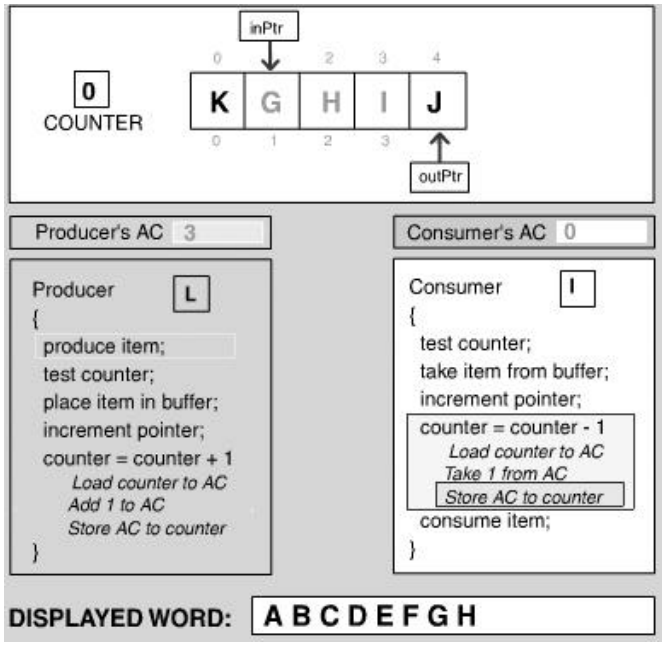

inPtr
outPtr
1
COUNTER
F G H I E
Producer's AC 5
Consumer's AC 1
Producer J
{
produce item;
test counter;
place item in buffer;
increment pointer;
counter = counter + 1
Load counter to AC
Add 1 to AC
Store AC to counter
}
Consumer I
{
test counter;
take item from buffer;
increment pointer;
counter = counter - 1
Load counter to AC
Take 1 from AC
Store AC to counter
consume item;
}
DISPLAYED WORD: A B C D E F G H

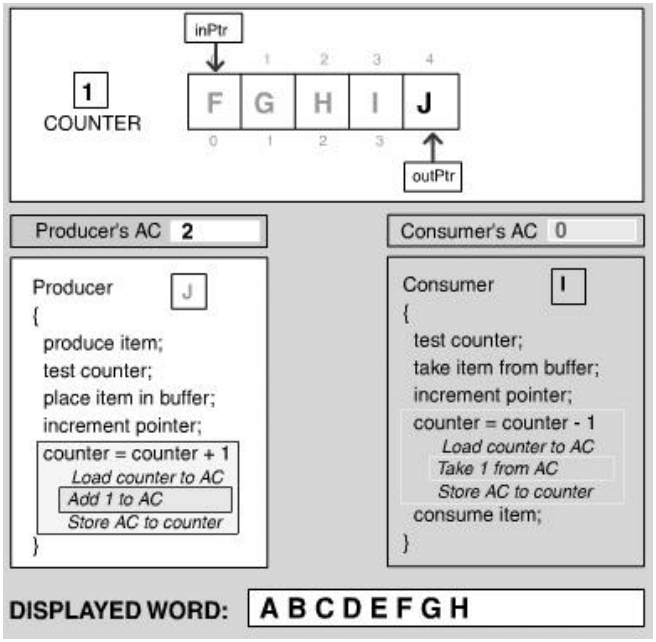

inPtr
outPtr
1
COUNTER
F G H I E
Producer's AC 5
Consumer's AC 1 - 1
Producer J
{
produce item;
test counter;
place item in buffer;
increment pointer;
counter = counter + 1
Load counter to AC
Add 1 to AC
Store AC to counter
}
Consumer I
{
test counter;
take item from buffer;
increment pointer;
counter = counter - 1
Load counter to AC
Take 1 from AC
Store AC to counter
consume item;
}
DISPLAYED WORD: A B C D E F G H

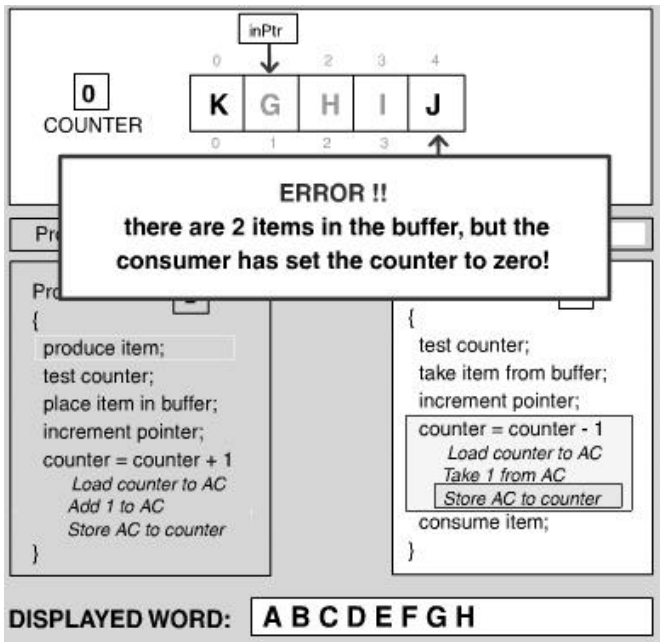

inPtr
outPtr
1
COUNTER
F G H I E
Producer's AC 5
Consumer's AC 1
Producer J
{
produce item;
test counter;
place item in buffer;
increment pointer;
counter = counter + 1
Load counter to AC
Add 1 to AC
Store AC to counter
}
Consumer I
{
test counter;
take item from buffer;
increment pointer;
counter = counter - 1
Load counter to AC
Take 1 from AC
Store AC to counter
consume item;
}
DISPLAYED WORD: A B C D E F G H


inPtr
outPtr
1
COUNTER
F G H I E
Producer's AC 5
Consumer's AC 0
Producer J
{
produce item;
test counter;
place item in buffer;
increment pointer;
counter = counter + 1
Load counter to AC
Add 1 to AC
Store AC to counter
}
process switch
Consumer I
{
test counter;
take item from buffer;
increment pointer;
counter = counter - 1
Load counter to AC
Take 1 from AC
Store AC to counter
consume item;
}
DISPLAYED WORD: A B C D E F G H



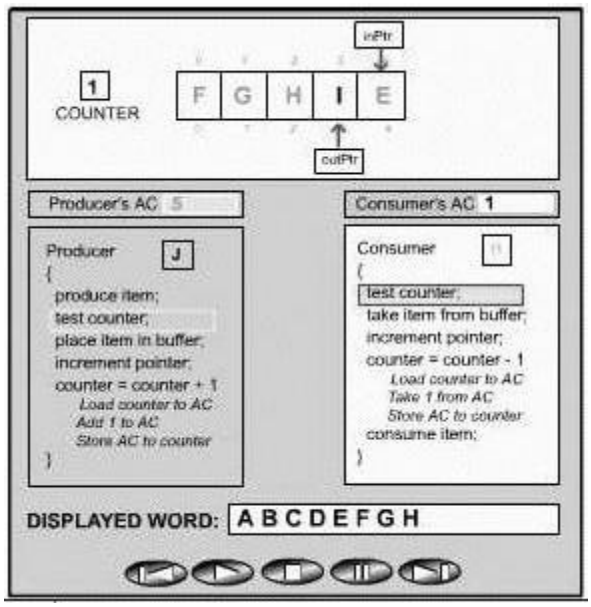






- Which uses the bounded-buffer with counter solution to show problems with atomicity and mutual exclusion?

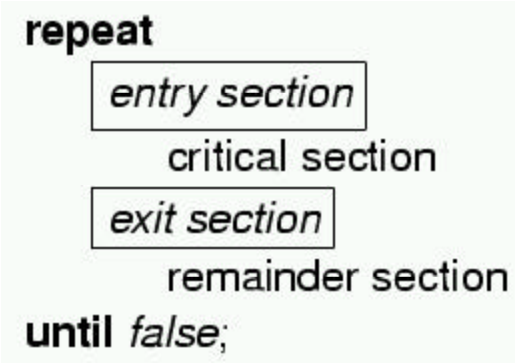
Critical Section



Consider a system consisting of several processes, each having a segment of code called a critical section, in which the process may be changing common variables, updating tables, etc. The important feature of the system is that when one process is executing its critical section, no other process is to be allowed to execute its critical section. Execution of the critical section is mutually exclusive in time.

The critical section problem is to design a protocol that these processes can use to cooperate safely. Each process must request permission to enter its critical section (entry section). The critical section may be followed by the exit section.

- *n* processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem - ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
- Structure of process P_i



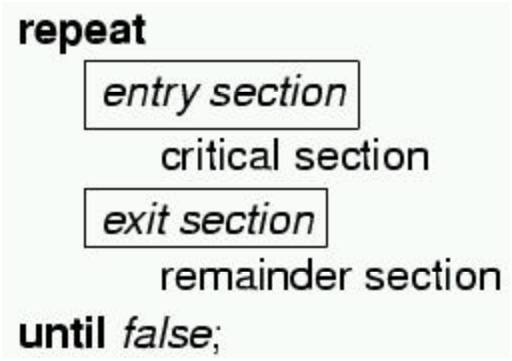
Solution to Critical Section Problem

1. **Mutual Exclusion**. If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress**. If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting**. A bound must exist 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.

Number 1 just says the obvious, that no two processes can be executing their critical section at the same time. Number 2 says that the choosing process (who will enter the critical section) should not depend on processes currently executing their critical section - the system still has to work even if nobody is executing the critical section. Number 3 says that no process should have to wait forever to execute its critical section.

Two-Process Synchronization

- Assume that each process executes at a nonzero speed.
- No assumption concerning relative speed of the n processes.
- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)



- Processes may share some common variables to synchronize their actions.

Algorithm 1

- Shared variables:

```
int turn;   //turn can have a value of either 0 or 1
//if turn = i, P(i) can enter it's critical section
```

- **Process P_i**

```
do
{
 while (turn != i)
 {
  /*do nothing*/
 }
 critical section
 turn = j;
 remainder section
}
```

while (true)

· **Satisfies mutual exclusion, but not progress.**

Algorithm 2

· Shared variables

```
boolean flag[2];
flag[0] = flag[1] = false;
// if flag[i] == true, P(i) ready to
// enter its critical section
```

initially flag[0] = flag[1] = false.

flag[i] = true ==> P_i ready to enter its critical section

· **Process P_i**

```
do
{
 flag[i]= true;
 while (flag[j])
 {
  /*do nothing*/
 }
 critical section
 flag[i] = false;
 remainder section
} while (true)
```

· Does not satisfy the mutual exclusion requirement.

Algorithm 3

- Combined shared variables of algorithms 1 and 2.
- Process P_i

```
do
{
 flag[i] = true;
 turn = j;
 while (flag[j] && turn == j)
 {
  /*do nothing*/
 }
 critical section
 flag[i] = false;
 remainder section
}
while (true)
```

- Meets all three requirements; solves the critical section problem for two processes.

Let us now discuss what is Bakery Algorithm

Critical section for n processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...
- Notation <= lexicographical order (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 = , . . . , n - 1
- Shared data

```
boolean choosing[n]; //initialise all to false
int number[n];       //initialise all to 0
```

· **Data structures are initialized to false and 0, respectively**

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

  for(int j = 0; j < n; j++)
  {
   while (choosing[j]== true)
   {
     /*do nothing*/
   }

    while ((number[j]!=0) &&
        (number[j],j)< (number[i],i))
   {
   /*do nothing*/
   }
  }

  do critical section
  number[i] = 0;
  do remainder section
 }while (true)
```

Synchronization Hardware

· Test and modify the content of a word atomically

```
boolean Test_and_Set(boolean& target)
 {
  boolean test = target;
  target = true;
  return test;
 }
```

Mutual Exclusion with Test and Set

· Shared data:

```
boolean lock = false;
```

· **Process P_i**

```
do
{
 while (Test-and-Set(lock))
 {
  /*do nothing*/
 }

 critical section
 lock = false;
```

remainder section
}while (true)

Check Your Progress

What is a critical Section?

_____
_____
_____
_____
_____
_____
_____
_____

What is Process Synchronization?

_____
_____
_____
_____
_____
_____
_____
_____
_____

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems / Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

Objectives

In previous Lesson you learn about Process Synchronization, Synchronization hardware and critical section. Today I will teach you about the Process Synchronization with Semaphores

Semaphore

A non-computer meaning of the word semaphore is a system or code for sending signals, by using arms or flags held in hands, etc. Various positions represent different letters and numbers. These are the things that used to be used on ships to coordinate their motion (before the invention of radios). Presently, you might have seen them used on aircraft carriers to coordinate the onboard activities of airplanes.

In a computer sense, a semaphore is an integer variable that, apart from initialization, is accessed only through two standard *atomic* operations: wait and signal. These operations were originally termed P (for *wait*; from the Dutch *proberen*, to test) and V (for *signal*; from *verhogen*, to increment). The classical definition of *wait* in pseudocode is

Points you should Remember

- Synchronization tool that does not require busy waiting.
- Semaphore S - integer variable
- can only be accessed via two indivisible (atomic) operations

```
wait(s)
{
  while (S<=0)
  {
    /*do nothing*/
  }
  S= S-1;
}

signal(S)
{
  S = S + 1;
}
```

Example: Critical Section for n Processes

- Shared variables
  - **semaphore** mutex;
  - initially

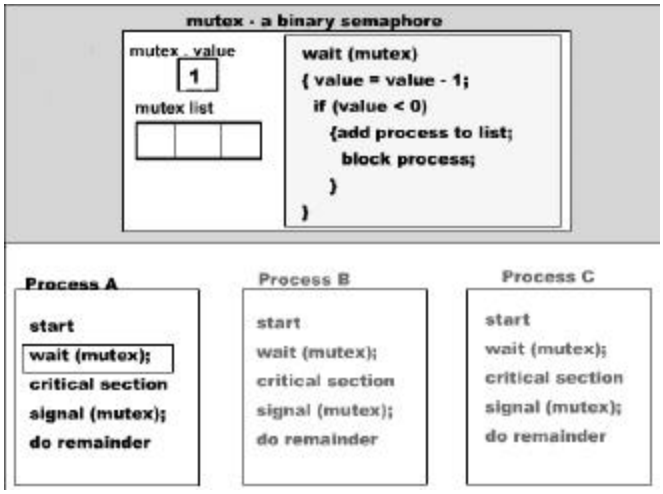
mutex = 1

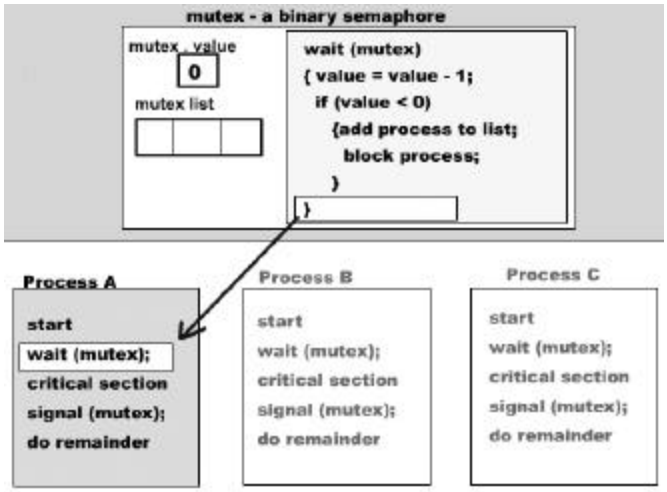
- Process P_i

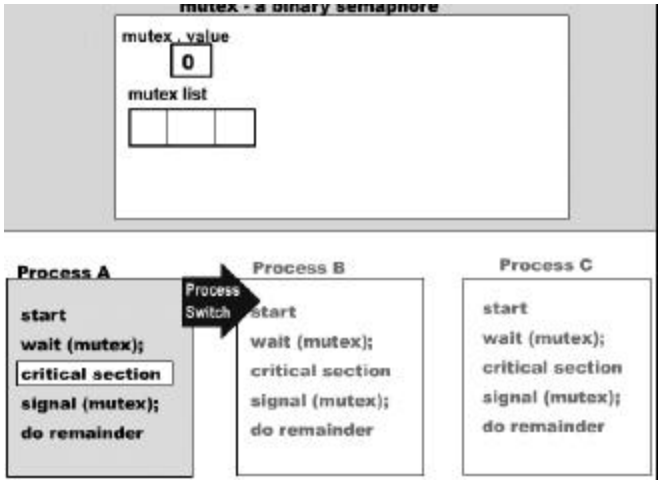
```
do
    {
      wait (mutex);
      critical section
      signal (mutex);
      remainder section
    }
    while(true)
```

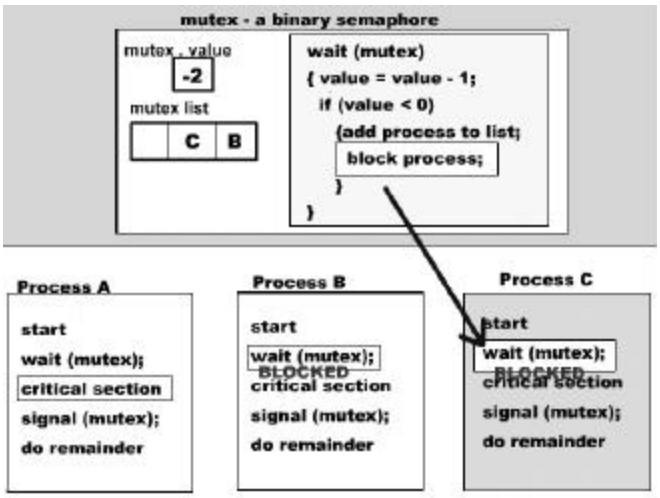
Mutual Exclusion with Semaphores

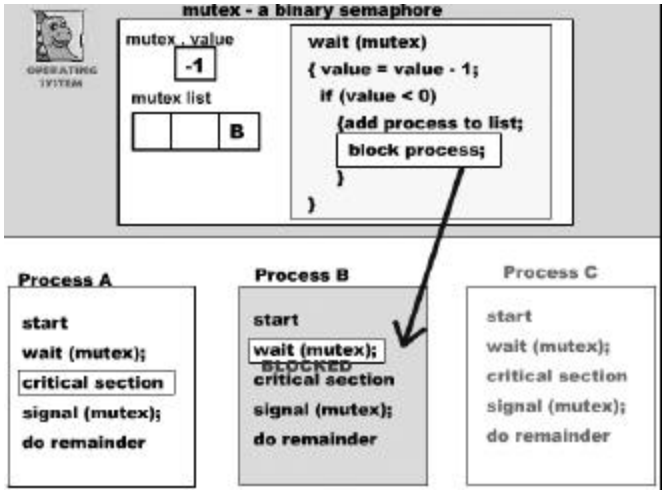
- which shows how to use semaphores for mutual exclusion.

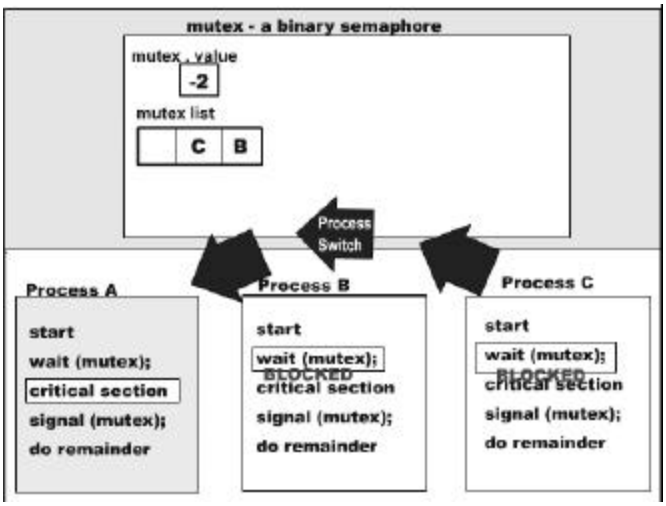


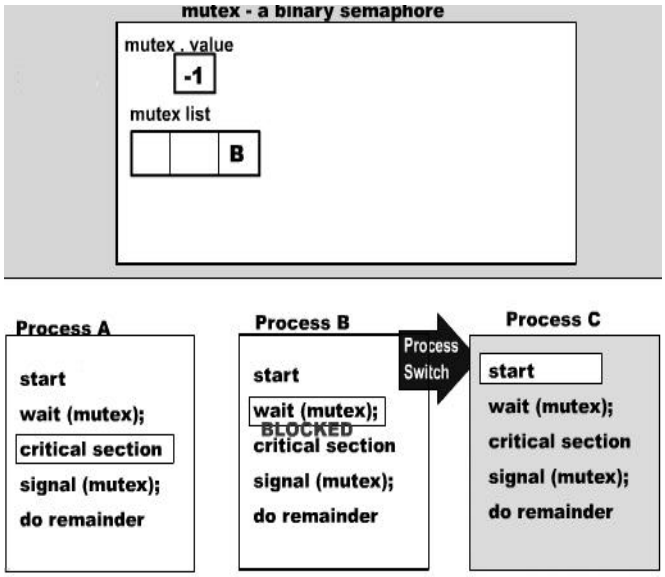


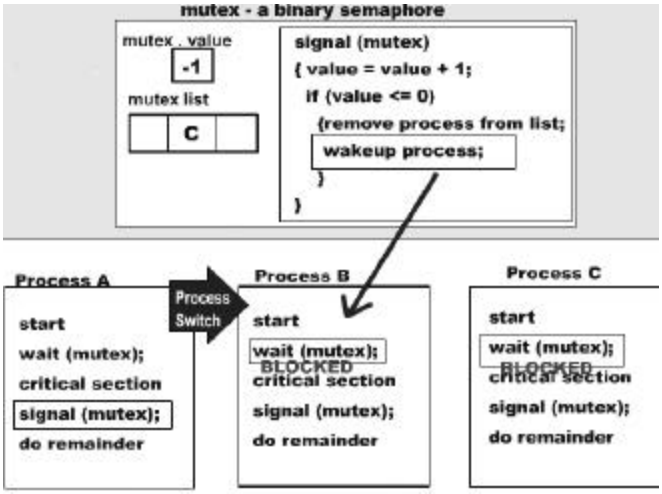

mutex - a binary semaphore
mutex . value
0
mutex list
Process A
start
wait (mutex);
critical section
signal (mutex);
do remainder
Process Switch
Process B
start
wait (mutex);
critical section
signal (mutex);
do remainder
Process C
start
wait (mutex);
critical section
signal (mutex);
do remainder

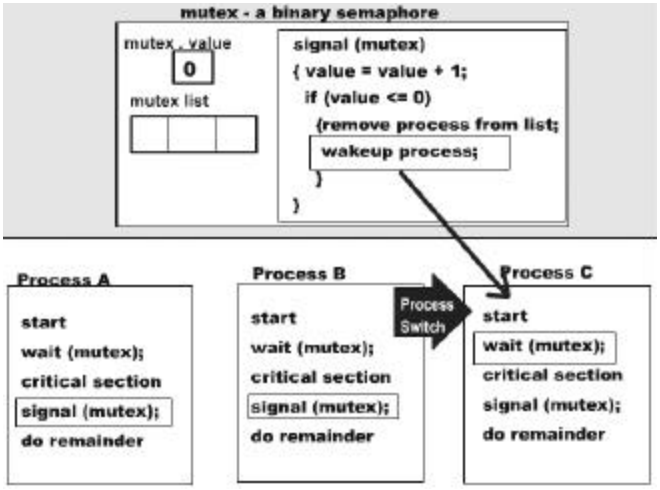

mutex - a binary semaphore
mutex . value
-2
mutex list
C
B
wait (mutex)
{ value = value - 1;
if (value < 0)
{add process to list;
block process;
}
}
Process A
start
wait (mutex);
critical section
signal (mutex);
do remainder
Process B
start
wait (mutex);
BLOCKED
critical section
signal (mutex);
do remainder
Process C
start
wait (mutex);
BLOCKED
critical section
signal (mutex);
do remainder


mutex - a binary semaphore
mutex . value
-1
mutex list
B
wait (mutex)
{ value = value - 1;
if (value < 0)
{add process to list;
block process;
}
}
Process A
start
wait (mutex);
critical section
signal (mutex);
do remainder
Process B
start
wait (mutex);
BLOCKED
critical section
signal (mutex);
do remainder
Process C
start
wait (mutex);
critical section
signal (mutex);
do remainder


mutex - a binary semaphore
mutex . value
-2
mutex list
C
B
Process Switch
Process A
start
wait (mutex);
critical section
signal (mutex);
do remainder
Process B
start
wait (mutex);
BLOCKED
critical section
signal (mutex);
do remainder
Process C
start
wait (mutex);
BLOCKED
critical section
signal (mutex);
do remainder


mutex - a binary semaphore
mutex . value
-1
mutex list
B
Process A
start
wait (mutex);
critical section
signal (mutex);
do remainder
Process B
start
wait (mutex);
BLOCKED
critical section
signal (mutex);
do remainder
Process Switch
Process C
start
wait (mutex);
critical section
signal (mutex);
do remainder


mutex - a binary semaphore
mutex . value
-1
mutex list
C
signal (mutex)
{ value = value + 1;
if (value <= 0)
{remove process from list;
wakeup process;
}
}
Process A
start
wait (mutex);
critical section
signal (mutex);
do remainder
Process Switch
Process B
start
wait (mutex);
BLOCKED
critical section
signal (mutex);
do remainder
Process C
start
wait (mutex);
BLOCKED
critical section
signal (mutex);
do remainder



Semaphore Implementation

- Define a semaphore as a record/structure

```
struct semaphore
{
  int value;
  List *L;    //a list of processes
}
```

- Assume two simple operations:
  - block suspends the process that invokes it.
  - wakeup(P) resumes the execution of a blocked process P.
- Semaphore operations now defined as

```
wait(S)
{
  S.value = S.value -1;
  if (S.value <0)
  {
    add this process to S.L;
    block;
  }
}

signal(S)
{
  S.value = S.value + 1;
  if (S.value <= 0)
  {
    remove a process P from S.L;
    wakeup(P);
  }
}
```

Semaphore as General Synchronization Tool

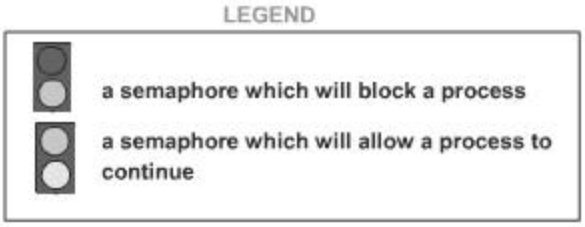
- Execute B in P_j only after A executed in P_i
- Use semaphore *flag* initialized to 0
- Code:
- P_i

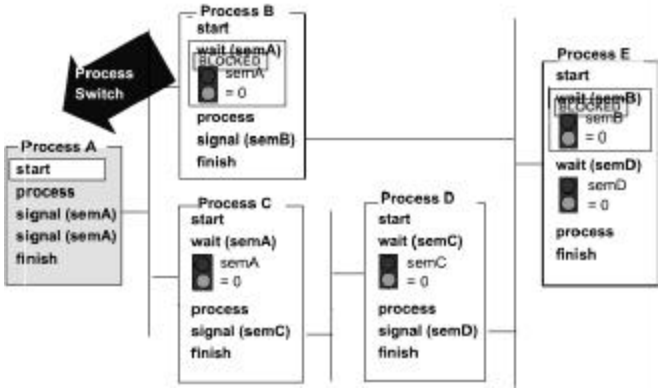
  A
  signal(flag)
- P_j

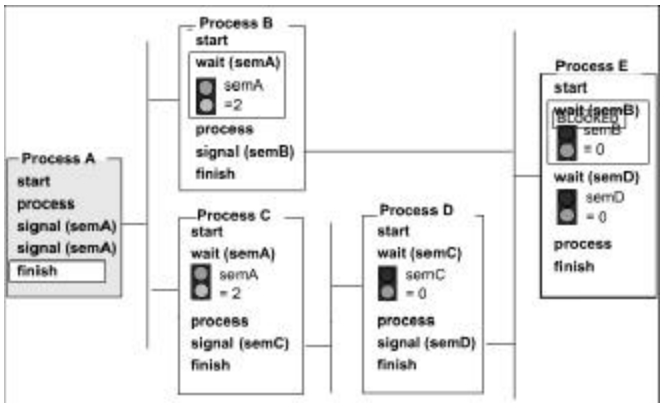
  .
  .
  .
  wait(flag)
  B

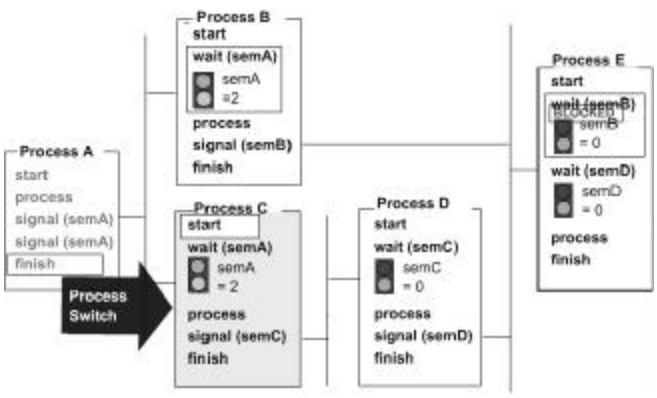
Semaphores as Process Synchronization

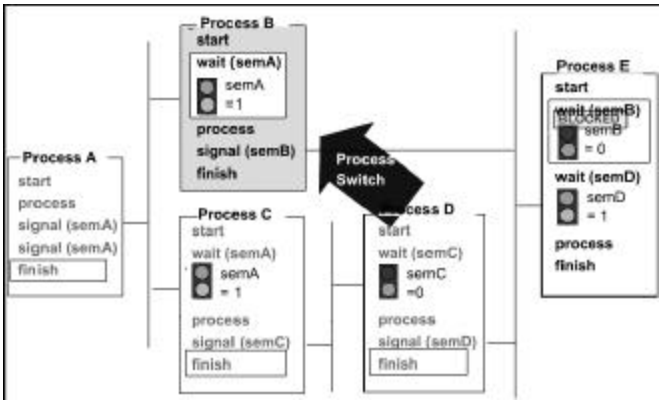
- showing how semaphores can be used for process synchronization.

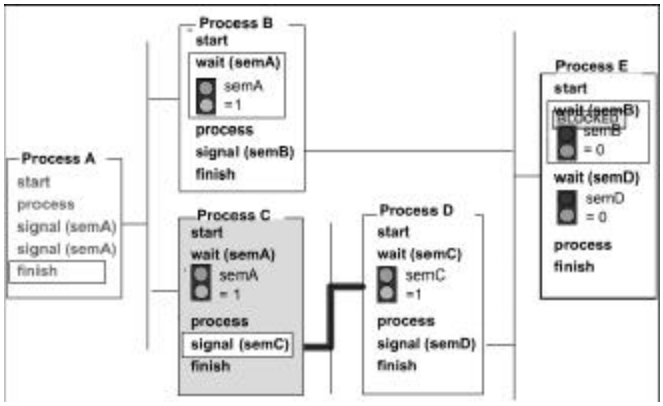


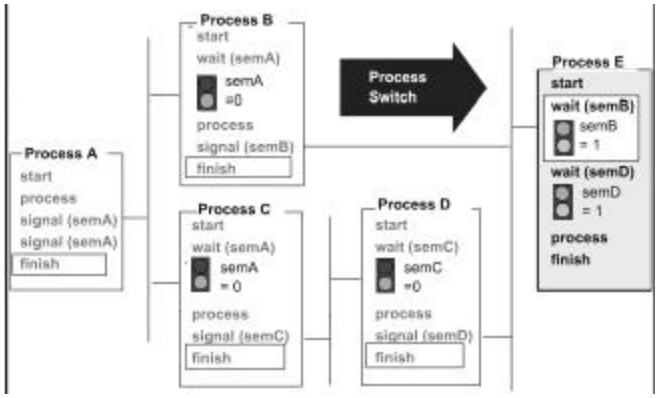


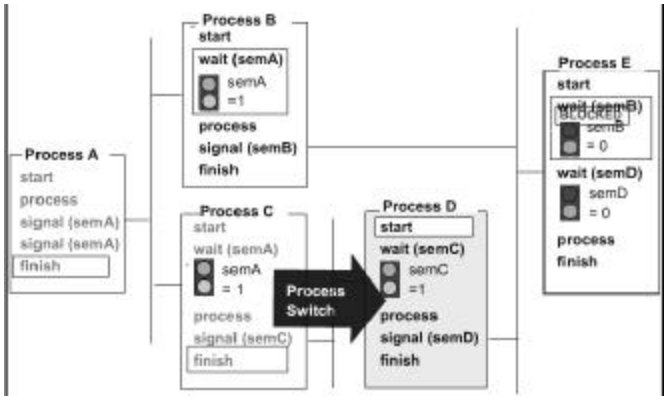


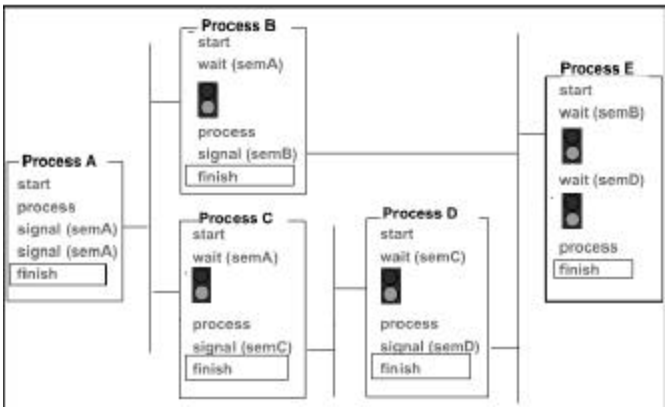












Let us discuss Deadlock and Starvation

- Deadlock - two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- Let S and Q be two semaphores initialized to 1

P_0	P_1

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

Starvation -
indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Check Your Progress

What is a semaphore? What am its drawbacks?

_____
_____
_____
_____
_____
_____
_____

What is a DeadLock ? How to break DeadLocks ?

_____
_____
_____
_____
_____
_____
_____

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems /Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

Objectives

Hello students, In previous Lesson you learnt about the Process Synchronization, Synchronization hardware critical section and Semaphores. Today I will teach you various types of Semaphores, Monitors and Atomic transactions.

Two Types of Semaphores

- Counting semaphore - integer value can range over an unrestricted domain.
- Binary semaphore - integer value can range only between 0 and 1; can be simpler to implement.
- Can implement a counting semaphore S as a binary semaphore.

Implementing S (Semaphore) as a Binary Semaphore

- Data structures:

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

- **Initialization:**

```
S1 = 1;
S2 = 0;
C = initial value of semaphore S;
```

- **wait operation**

```
wait( S1 );
C = C - 1;
if (C<0)
{
 signal( S1 );
 wait( S2 );
}
else
 signal( S1 );
```

- **signal operation**

```
wait( S1 );
C = C+1;
if (C <= 0)
 signal( S2 );
signal( S1 );
```

Let us discuss classical Problems of Synchronization

- BoundedBuffer Problem
- Readers and Writers Problem
- DiningPhilosophers Problem

Bounded Buffer Problem

- Shared data

```
char item;          //could be any data type
  char buffer[n];
  semaphore full = 0;     //counting semaphore
  semaphore empty = n;    //counting semaphore
  semaphore mutex = 1;    //binary semaphore
  char nextp, nextc;
```

- **Producer process**

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

- Consumer process

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

Readers-Writers Problem

- Shared data

```
semaphore mutex = 1;
  semaphore wrt = 1;
  int readcount = 0;
```

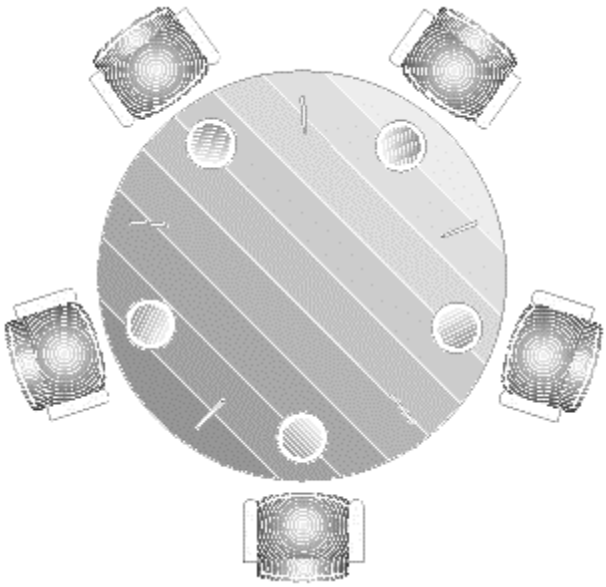
- Writer process

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

- Reader process

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

Dining Philosophers Problem



- Shared data

semaphore chopstick[5];
chopstick[] = 1;

- Philosopher i:

```
do
   {
    wait (chopstick[i]);
    wait (chopstick[i+1 mod 5]);
    eat;
    signal (chopstick [1]);
    signal (chopstick [i+1 mod 5]);
    think;
   }
   while (true)
```

Critical Regions

- High level synchronization construct
- A shared variable v of type T, is declared as:
- **var** v: **shared** T
- Variable v accessed only inside statement:
- **region** v **when** B **do** S

where B is a Boolean expression.

While statement S is being executed, no other process can access variable v.

- Regions referring to the same shared variable exclude each other in time.
- When a process tries to execute the region statement, the Boolean expression B is evaluated. If B is true, statement S is executed. If it is false, the process is delayed until B becomes true and no other process is in the region associated with v.

Example - Bounded Buffer

- Shared variables:
- Producer process inserts nextp into the shared buffer

CODE GOES HERE

- Consumer process removes an item from the shared buffer and puts it in nextc

CODE GOES HERE

Implementation: region x when B do S

- Associate with the shared variable x, the following variables:

CODE GOES HERE

Mutually exclusive access to the critical section is provided by mutex.

- If a process cannot enter the critical section because the Boolean expression B is false, it initially waits on the first delay semaphore; moved to the second delay semaphore before it is allowed to reevaluate B.
- Keep track of the number of processes waiting on first delay and second delay , with first count and second count respectively.
- The algorithm assumes a FIFO ordering in the queuing of processes for a semaphore.
- For an arbitrary queuing discipline, a more complicated implementation is required.

Code Goes Here

Monitors

High level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```
class monitor
   {
    variable declarations

    P(1)
    {...}

    P(2)
    {...}

    P(n)
    {...}

    Initialization code
   }
```

- To allow a process to wait within the monitor, a condition variable must be declared, as:

condition x, y;

- Condition variable can only be used with the operations 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.

Dining Philosophers Example

```
monitor dining-philosophers
 {
  enum state {thinking, hungry, eating};
  state 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 k)
  {
    if ((state[k+4 % 5] != eating) && (state[k]==hungry)
                          && state[k+1 % 5] != eating))
    {
      state[k] = eating;
      self[k].signal;
    }
  }

  init
  {
    for (int i = 0; i< 5; i++)
      state[i] = thinking;
  }
}
```

Monitor Implementation Using Semaphores

· Variables

```
semaphore mutex = 1;
semaphore next = 0;
int next-count;
```

· Each external procedure F will be replaced by

```
wait(mutex);
...
 body of F;
...
if (next-count > 0)
  signal(next);
else
  signal(mutex);
```

· Mutual exclusion within a monitor is ensured.

· For each condition variable x, we have:

```
semaphore x-sem = 0;
   semaphore x-count = 0;
```

- The operation x.wait can be implemented as:

```
x-count = x-count + 1;
    if (next-count > 0)
     signal(next);
    else
      signal(mutex);
    wait(x-sem);
    x-count = x-count - 1;
```

- The operation x.signal can be implemented as:

```
if (x-count > 0)
{
  next-count = next-count + 1;
  signal(x-sem);
  wait(next);
  next-count = next-count - 1;
}
```

- Conditional wait construct: x.wait(c);
  - c - integer expression evaluated when the wait operation is executed.
  - value of c (priority number) stored with the name of the process that is suspended.
  - when x.signal is executed, process with smallest associated priority number is resumed next.
- Check two conditions to establish correctness of system:
  - User processes must always make their calls on the monitor in a correct sequence.
  - Must ensure that an uncooperative process does not ignore the mutual exclusion gateway provided by the monitor, and try to access the shared resource directly , without using the access protocols.

Atomic Transactions

- Transaction - program unit that must be executed atomically; that is, either all the operations associated with it are executed to completion, or none are performed.
- Must preserve atomicity despite possibility of failure.
- We are concerned here with ensuring transaction atomicity in an environment where failures result in the loss of information on volatile storage.

Log Based Recovery

- Write ahead log - all updates are recorded on the log, which is kept in stable storage; log has following fields:
  - transaction name
  - data item name, old value, new value
- The log has a record of <T_i **starts**>, and either
- <T_i **commits**> if the transactions commits, or
- <T_i **aborts**> if the transaction aborts.
- Recovery algorithm uses two procedures:

- **undo**(T_i) - restores value of all data updated by transaction T_i to the old values. It is invoked if the log contains record <T_i **starts**>, but not <T_i **commits**>.
- **redo**(T_i) - sets value of all data updated by transaction T_i to the new values. It is invoked if the log contains both <T_i **starts**> and <T_i **commits**>.

Checkpoints - Reduce Recovery Overhead

1. Output all log records currently residing in volatile storage onto stable storage.
2. Output all modified data residing in volatile storage to stable storage.
3. Output log record < **checkpoint**> onto stable storage.

- Recovery routine examines log to determine the most recent transaction T_i that started executing before the most recent checkpoint took place.
  - Search log backward for first < **checkpoint**> record.
  - Find subsequent <T_i **start**> record.
- **redo** and **undo** operations need to be applied to only transaction T_i and all transactions T_j that started executing after transaction T_i.

Concurrent Atomic Transactions

- Serial schedule - the transactions are executed sequentially in some order.
- Example of a serial schedule in which T_0 is followed by T_1 :

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

- Conflicting operations - O_i and O_j conflict if they access the same data item, and at least one of these operations is a write operation.
- Conflict serializable schedule - schedule that can be transformed into a serial schedule by a series of swaps of non-conflicting operations.

Example of a Concurrent Serializable Schedule

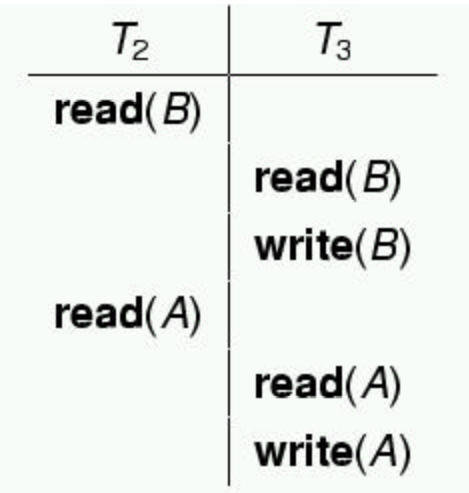
T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

- Locking protocol governs how locks are acquired and released; data item can be locked in following modes:
- **Shared**: If T_i has obtained a shared mode lock on data item Q, then T_i can read this item, but it cannot write Q.
- **Exclusive**: If T_i has obtained an exclusive mode lock on data item Q, then T_i can both read and write Q.

Two-phase locking protocol

- **Growing phase**: A transaction may obtain locks, but may not release any lock.
- **Shrinking phase**: A transaction may release locks, but may not obtain any new locks.
- The two-phase locking protocol ensures conflict serializability , but does not ensure freedom from deadlock.
- Timestamp ordering scheme - transaction ordering protocol for determining serializability order.
- With each transaction T_i in the system, associate a unique fixed timestamp, denoted by TS(T_i).
- If T_i has been assigned timestamp TS(T_i), and a new transaction T_j enters the system, then TS(T_i) < TS(T_j).
- Implement by assigning two timestamp values to each data item Q.
- **Wtimestamp**(Q) - denotes largest timestamp of any transaction that executed **write**(Q) successfully.
- **Rtimestamp**(Q) - denotes largest timestamp of any transaction that executed **read**(Q) successfully.

Schedule Possible under Timestamp Protocol



- There are schedules that are possible under the two-phase locking protocol but are not possible under the timestamp protocol, and vice versa.

The timestamp ordering protocol ensures conflict serializability; conflicting operations are processed in timestamp order.

Check Your Progress

What are various type of Semaphores?

_____
_____
_____
_____
_____

What is Atomic Transaction?

_____
_____
_____
_____
_____

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems / Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

Objectives

In this lecture, you will learn about the concept of **deadlocks.**

You learnt in the earlier lectures that a process is a program in execution and from the operating system's point of view; it is a unit of **resource allocation**.

You may wonder what these resources are. The resources can be physical or logical. Physical resources could be disk, printers etc. while logical resources could be files. The CPU, memory and other processes could also be resources that are shared.

The processes compete for these resources and sometimes get into a state of deadlock.

Overall Picture

In a multiprogramming environment where several processes compete for resources, a situation may arise where a process is waiting for resources that are held by other waiting processes. This situation is called a **deadlock**

Introduction

Generally, a system has a finite set of resources (such as memory, IO devices, etc.) and a finite set of processes that need to use these resources.

A process which wishes to use any of these resources makes a request to use that resource. If the resource is free, the process gets it. If it is used by another process, it waits for it to become free.

The assumption is that the resource will eventually become free and the waiting process will continue on to use the resource. But what if the other process is also waiting for some resource?

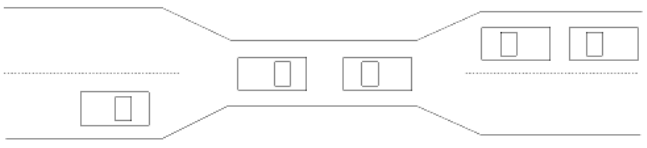
"A set of processes is in a deadlock state when every process in the set is waiting for an event that can only be caused by another process in the set."

Imagine a situation where two processes have acquired a lock on two tape drivers (or hard drives, etc.), but need two such resources to proceed with execution (for example, to copy something from one to the other). Each is waiting for the other process to release the other tape drive, which will never happen, since the other is also waiting for the same thing.

Well, what is this state of deadlock?

Let me explain with an example.

Bridge Crossing Example



- Traffic only in one direction.
- 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.

Let me explain this with an example:

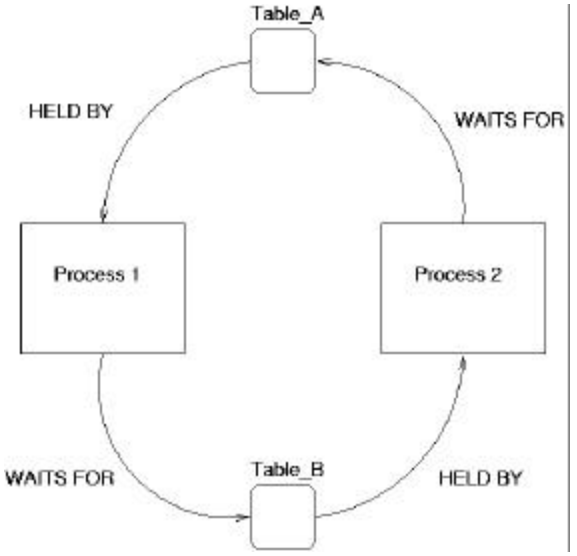
Suppose that a system has one tape and one printer device and two processes P1 and P2. Both P1 and P2 require the tape and printer devices for their functioning. The processes make their resource requests in the following sequence:

1. P1 requests tape
2. P2 requests printer
3. P1 requests printer
4. P2 requests tape

The set of processes {P1 and P2} is now deadlocked. Why?

The first two requests can be granted straightaway. P1 holds the tape device and P2 holds the printer device. Now when P1 asks for the printer, it is blocked because the printer is not currently available. P2 is similarly blocked when it asks for the tape. P1 comes out of the blocked state when P2 releases the resource that it holds. Similarly P2 comes out of its blocked state when P1 releases the resource it holds. Hence the two processes are deadlocked.

The diagram below illustrates the concept:



What are the consequences of deadlocks?

- Response times and elapsed times of processes suffer.

- If a process is allocated a resource R1 that it is not using and if some other process P2 requires the resource, then P2 is denied the resource and the resource remains idle.

How do you characterize deadlocks?

Deadlocks are **undesirable** because processes **never finish executing** and system **resources are tied up**

What are the conditions under which deadlocks can occur in a system?

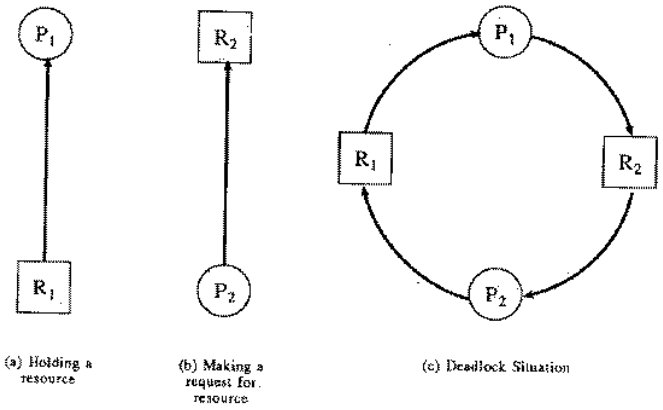
A deadlock situation can arise if the following four conditions hold simultaneously in a system:

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 the resource, the requesting process must be delayed until the resource has been released.

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.

No Preemption: Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

Circular Wait: A set {P0, P1, P2, …, Pn} of waiting processes must exist such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn-1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0.



(a) Holding a resource (b) Making a request for resource (c) Deadlock Situation

All four conditions *MUST* hold for a deadlock to occur.

Resource-Allocation Graph

The idea is to have a graph. The graph has two different types of nodes, the process nodes and resource nodes (process represented by circles, resource node represented by rectangles). For different instances of a resource, there is a dot in the resource node rectangle. For example, if there are two identical printers, the printer resource might have two dots to indicate that we don't really care which is used, as long as we acquire the resource.

The edges among these nodes represent resource allocation and release. Edges are directed, and if the edge goes from resource to process node that means the process has acquired the resource. If the edge goes from process node to resource node that means the process has requested the resource.

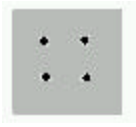
We can use these graphs to determine if a deadline has occurred or may occur. If for example, all resources only have one instance (all resource node rectangles have one dot) and the graph is circular, then a deadlock *has* occurred. If on the other hand some resources have several instances, then a deadlock *may* occur. If the graph is not circular, a deadlock cannot occur (the *circular wait* condition wouldn't be satisfied).

A set of vertices *V* and a set of edges *E*.

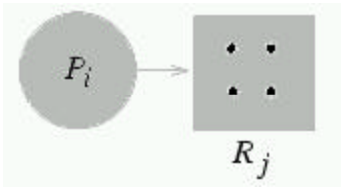
· *V* is partitioned into two types:

- $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the processes in the system.
- $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system.
- request edge - directed edge $P_i \longrightarrow R_j$
- assignment edge - directed edge $R_j \longrightarrow P_i$
- Process

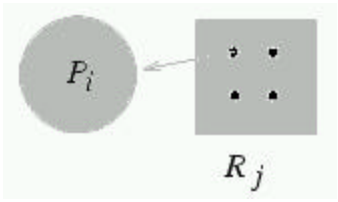
- Resource type with 4 instances



- P_i requests instance of R_j

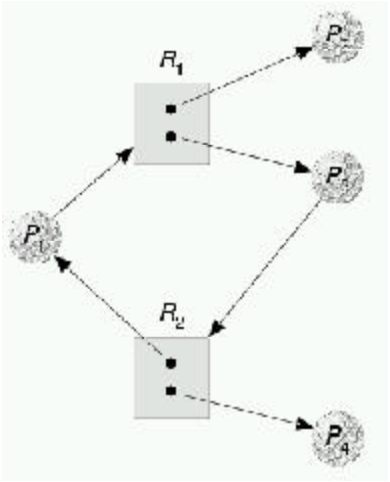


- P_i is holding an instance of R_j



Example of a Graph with No Cycles

Example of a Graph with a Cycle



Basic Facts

- If graph contains no cycles
  - No deadlock.
- If graph contains a cycle —>
  - If only one instance per resource type, then deadlock.
  - If several instances per resource type, possibility of

There are four strategies used for dealing with deadlocks.

1. Ignore the problem
2. Detect deadlocks and recover from them
3. Avoid deadlocks by carefully deciding when to allocate resources.
4. Prevent deadlocks by violating one of the 4 necessary conditions.

Ignoring the problem-The Ostrich Algorithm

The "put your head in the sand approach".

- If the likelihood of a deadlock is sufficiently small and the cost of avoiding a deadlock is sufficiently high it might be better to ignore the problem. For example if each PC deadlocks once per 100 years, the one reboot may be less painful that the restrictions needed to prevent it.
- Clearly not a good philosophy for nuclear missile launchers.
- For embedded systems (e.g., missile launchers) the programs run are fixed in advance so many of the questions Tanenbaum raises (such as many processes wanting to fork at the same time) don't occur.

Methods for Handling Deadlocks

You can deal with the deadlock issue in several ways:

1. You can use specific protocols to prevent or avoid deadlocks (preventing it).
2. You can detect the deadlock and recover from it (recovering after it has occurred).
3. You can totally ignore the deadlock problem (pretend it doesn't exist). This is what most operating systems do (including UNIX and Windows).

Preventing deadlocks is ensuring that at least one of the necessary four deadlock conditions cannot occur. Avoiding a deadlock is knowing which resources the process will use beforehand and only run the process when all the resources are available for the process to use (not make it wait for individual resources one by one - which may cause a deadlock).

If you do not provide for deadlock prevention or deadlock avoidance, the system may enter into deadlock state. At this point, we may employ some deadlock detection scheme and a recovery (if there is indeed a deadlock).

If we don't prevent nor recover from it, the system will eventually have deadlocks. This is the strategy used by most operating systems. Luckily, deadlocks are usually rare, and the systems that are affected usually suffer from other freezing problems (process not releasing control in a non-preemptive environment, program errors, etc.) that make deadlocks seem unimportant. [ie: if you reboot your Windows every day because something gets messed up, you won't mind rebooting your Windows every year because of a deadlock].

Check Your Progress

List several examples of deadlocks that are not related to computer system environment. Describe some of its characteristics?

_____

_____

_____

_____

_____

_____

What is Deadlock?

_____

_____

_____

_____

_____

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems / Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

Notes

Objectives

In previous lecture, you have learnt about the concept of **deadlocks,** In this lecture you will learn about deadlock prevention and what causes deadlocks.

What are the methods of handling deadlocks?

Deadlocks can be handled in many ways. These are as follows:

- Deadlock prevention
- Deadlock Avoidance
- Deadlock Detection and Recovery

In this lecture, we will discuss about deadlock prevention and in the next lecture we will look at the other two.

So what is deadlock prevention?

Deadlock prevention involves a set of methods for ensuring that at least one of the four necessary conditions cannot hold.

Let me explain each of these conditions in detail:

Mutual Exclusion: If no resource were ever assigned exclusively to a single process, we would never have deadlocks.

Suppose that two processes are allowed to write on the printer at the same time. This would lead to chaos. By spooling the printer output, several processes can generate output at the same time. The only process that actually requests the physical printer is the **printer daemon**. Since the printer daemon never requests for any other resources, deadlock can be eliminated for the printer.

The bottom-line:

"Avoid assigning a resource when that is not absolutely necessary."

Hold and wait: All processes are required to request all their resources before starting execution. If everything is available, process will be allocated whatever it needs and can run to completion. If one or more resources are busy, nothing will be allocated and process would just wait.

Disadvantages:

- Many processes do not know how many resources they will need until they have started running.
- Resources will not be used optimally with this approach
- Starvation may result

Here is an alternative to overcome this disadvantage:

A process can request resources only if it has none. This means it should first temporarily release all resources it currently holds.

Can you give us an example?

A process copies data from tape drive to disk file, sorts the disk file and then prints the results on a printer.

If all resources (tape drive, disk drive and printer) are requested at the beginning, then the process must initially request tape drive, disk file and printer. It will then hold the printer for its entire execution even though it needs it only at the end.

Alternatively: Initially request tape drive and disk file. Copy from tape drive to disk file. Then release both. Request again for disk file and printer. Copy from disk to printer. Release disk and printer.

No Preemption: Process holds some resources. It requests another resource, which cannot be immediately allocated to it. All resources currently being held are preempted. Preempted resources added to available list of resources for which the process is waiting. Process is restarted only when it can regain all its old resources as well as new ones that it is requesting.

Here is an example:

Process P1 requests for some resources.

Check if they are available.

If yes, allocate

If No,

Check if they are allocated to some other process that is waiting for additional resources.

If yes, preempt desired resources from waiting process and allocate to requesting process.

Circular Wait: To prevent circular wait, impose total ordering of all resources types. Each process, then, should request resources in an increasing order of enumeration.

Suppose R = {R1, R2 … Rm} is a set of resource types.

Assign to each resource type, a unique integer.

Define a one-to-one function 'F' such that R—>N where N is a set of natural numbers.

For example, F (Tape drive) = 1, F (Disk drive) = 5, F (Printer) = 12.

Each process requests resource in increasing order of enumeration.

A process can initially request any number of instances of resource type Ri.After that, process can request instances of resource type Rj if and only if F (Rj) > F (Ri).

Whenever a process requests an instance of resource type Rj, we must ensure that it has released any resources Ri such that F (Ri) >= F (Rj).

Consider another example:

Let i > j. If 'i' is allocated to A, then A cannot request 'j' because i > j.

Let i < j. If 'j' is allocated to B, then B can not request 'i'

The function F should be defined according to the normal order of usage of the resources in a system.

Check Your Progress

What are various method of Deadlock prevention?

_____

_____

_____

_____

Reference Books:

Author Dahmke, Mark.
Main Title Microcomputer Operating Systems / Mark Dahmke.
Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-
Main Title An Introduction To Operating Systems / Harvey M. Deitel.
Edition Rev. 1st Ed.
Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-
Main Title Fundamentals Of Operating Systems / A.M. Lister.
Edition 3rd Ed.
Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)
Main Title Introduction To Computer Systems / N.A.B. Gray.
Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.
Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.
Edition 2nd Ed.
Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.
Main Title Operating Systems / William Stallings.
Edition 6th Ed.
Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-
Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.
Edition 2nd Ed.
Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.
Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.
Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.
Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.
Edition 6th Ed.
Publisher Reading, Mass. : Addison Wesley Longman, C1998.

Notes

Objectives

In the last lecture, you learnt about the deadlocks, their characterization and deadlock prevention methods. And in this lecture, you will learn other methods of handling deadlocks such as deadlock detection & recovery.

Detecting Deadlocks with single unit resources

Consider the case in which there is **only one** instance of each resource.

- So a request can be satisfied by only one specific resource.
- In this case the 4 necessary conditions for deadlock are also sufficient.
- Remember we are making an assumption (single unit resources) that is often invalid. For example, many systems have several printers and a request is given for "a printer" not a specific printer. Similarly, one can have many tape drives.
- So the problem comes down to finding a directed cycle in the resource allocation graph. Why? Answer: Because the other three conditions are either satisfied by the system we are studying or are not in which case deadlock is not a question. That is, conditions 1, 2, 3 are conditions on the system in general not on what is happening right now.

To find a directed cycle in a directed graph is not hard. The algorithm is in the book. The idea is simple.

1. For each node in the graph do a depth first traversal (hoping the graph is a DAG (directed acyclic graph), building a list as you go down the DAG.
2. If you ever find the same node twice on your list, you have found a directed cycle and the graph is not a DAG and deadlock exists among the processes in your current list.
3. If you never find the same node twice, the graph is a DAG and no deadlock occurs.
4. The searches are finite since the list size is bounded by the number of nodes.

Detecting Deadlocks with multiple unit resources

This is more difficult.

- The figure on the right shows a resource allocation graph with multiple unit resources.
- Each unit is represented by a dot in the box.
- Request edges are drawn to the box since they represent a request for any dot in the box.
- Allocation edges are drawn from the dot to represent that this unit of the resource has been assigned (but all units of a resource are equivalent and the choice of which one to assign is arbitrary).
- Note that there is a directed cycle in black, but there is no deadlock. Indeed the middle process might finish, erasing the magenta arc and permitting the blue dot to satisfy the rightmost process.
- The book gives an algorithm for detecting deadlocks in this more general setting. The idea is as follows.

7. look for a process that might be able to terminate (i.e., all its request arcs can be satisfied).
8. If one is found pretend that it does terminate (erase all its arcs), and repeat step 1.
9. If any processes remain, they are deadlocked.

· We will soon do in detail an algorithm (the Banker's algorithm) that has some of this flavor.

Detection-Algorithm Usage

Detection algorithms need to be executed to detect a deadlock. The frequency and time when you run such algorithm is dependent on how often you assume deadlocks occur and how many processes they may effect.

If deadlocks may happen often, you run the detection often. If it affects many processes, you may decide to run it often so that less processes are affected by the deadlock.

You could run the algorithm on every resource request, but considering that deadlocks are rare, it is not very efficient use of resources. You could run the algorithm from time to time, say every hour, or when CPU utilization crosses some threshold, or at random times during the system execution lifetime.

Deadlock Recovery

You can recover from a deadlock via two approaches: we either kill the processes (that releases all resources for killed process) or take away resources.

Process Termination

When recovering from a deadlock via process termination, you have two approaches. You can terminate all processes involved in a deadlock, or terminate them one by one until the deadlock disappears.

Killing all processes is costly (since some processes may have been doing something important for a long time) and will need to be re-executed again.

Killing a process at a time until deadlock is resolved is also costly, since we must rerun deadlock detection algorithm every time you terminate a process to make sure we got rid of the deadlock.

Also, some priority must be considered when terminating processes, since you don't want to kill an important process when less important processes are available. Priority might also include things like how many resources are being held by that process, or how long has it executed, or how long it has to go before it completes, or how many resources it needs to complete its job, etc.

Resource Preemption

This approach takes resources from waiting processes and gives them to other processes. Obviously, the victim process cannot continue regularly, and you have a choice of how to handle it. We can either terminate that process, or roll it back to some previous state so that it can request the resources again.

Again, there are many factors that determine which process you choose as the victim.

Note: that if the system has resource preemption, by definition, a deadlock cannot occur. The type of resource preemption you are talking about here is non-normal preemption that only occurs when a deadlock detection mechanism detected a deadlock.

Check Your Progress

Explain Deadlock Detection process?

_____

_____

_____

_____

_____

How to recover from Deadlock?

_____

_____

_____

_____

_____

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems / Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

Objectives

In the last lecture, you learnt about deadlocks, their characterization and deadlock detection & prevention methods.In todays lecture, you will learn other methods of handling deadlocks such as deadlock avoidance and detection & recovery.

How can you avoid Deadlocks (Deadlock Avoidance)

It turns out that there are algorithms you can use to avoid deadlock. The principle is as follows:

Whenever you are about to make an allocation, you can run this algorithm and see if making that allocation would lead to a deadlock.

This seems like the ideal method but has some important drawbacks.

Let us see what these drawbacks are:

1. The algorithms are not that fast and there is a lot of overhead in running them before every resource allocation. The OS might be allocating resources hundreds of times a second.
2. The algorithms assume that processes know their maximum resource needs apriori. But this is often not the case.
3. They (processes) assume that they know what resources are available in the system. Hardware can go down and resources can become unavailable. This could lead to unpredictable deadlock situations.

In short, in deadlock avoidance method, the OS must be given in advance additional information concerning which resources a process will request and use during its lifetime.

Let me give you an example:

A system has one tape drive and one printer.

- Process P will request first tape drive and then the printer before releasing both.
- Process Q will request first printer and then tape drive.
- This information should be available apriori so that OS can decide for each request, whether or not process should wait.

So, in deadlock avoidance, in order to decide whether current request can be satisfied or must wait to avoid a possible future deadlock, for each request, the system should consider:

- Resources that are currently available.
- Resources that are currently allocated.
- Future requests and releases of each process.

So how does the algorithm actually work?

Here are some salient points of the algorithm:

- The deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that there can never be a circular-wait condition.
- A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid deadlock.
- A system is in a safe state only if there exists a safe sequence.
- A safe state is not a deadlocked state but a deadlocked state is an unsafe state.
- An unsafe state may lead to a deadlock.

Let me explain this with an example:

Consider a system that has **ten** instances of resource type R, and three processes A, B and C. The resource allocation and demand is as follows:

'A' has 3 instances of R but may need 9 eventually.

'B' has 2 instances of R but may need 4 eventually.

'C' has 2 instances of R but may need 7 eventually.

Currently allocated resources: 3 + 2 + 2 = 7
Currently available resources: 10 - 7 = 3

Is this a safe state?

To answer this question, you must find out if there is at least one sequence of scheduling that allows all processes to complete. Let us check out the following:

- The scheduler runs process **B** first which requests **two additional** resources and gets it. After this we are left with 1 resource [10 – (3 + 4+ 2)]. Eventually, B completes and releases all the four resources it holds. The resource pool now has 1 + 4 = **5 resources**.
- The scheduler runs process **C** next which requests **five additional** resources and gets it. After this we are left with 0 resource [10 – (3 + 0+ 7)]. Eventually, C completes and releases all the seven resources it holds. The resource pool now has 0 + 7 = **7 resources**.
- The scheduler runs process **A** next, which requests **six additional** resources and gets it. After this we are left with 1 resource [10 – (9 + 0+ 0)]. Eventually, A completes and releases all the nine resources it holds. The resource pool now has 1 + 9 = **10 resources**.

So, you can see that by scheduling the processes in the sequence **B C A,** the OS ensures that all processes run to completion. Hence we conclude that the initial state of the system is **safe**.

Can you have a formal definition of a safe sequence?

A safe sequence is a sequence of processes <P1, P2, P3, …, Pn> for the current allocation state if for each Pi, the resources that Pi can still request can be satisfied by the currently available resources plus the resources held by all Pj with j <= i .

The Banker's Algorithm for deadlock avoidance:

One of the most popular deadlock avoidance algorithms is the Banker's Algorithm proposed by Dijkstra in 1965.

It is a scheduling algorithm that can avoid deadlocks. The name was chosen because this algorithm could be used in a banking system to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all its customers.

- A town banker deals with a group of customers to whom he has granted lines of credits.
- There are four customers: A, B, C and D, which are analogous to four processes.
- The credit unit is like the resource
- The banker himself is the OS
- Assume each credit unit = Rs. 1000.

Not all customers need their maximum credit immediately. Hence only 10 credit units are reserved.

Process	**Current**	**Max.**	Free = 10
A	0	6	
B	0	5	
C	0	4	
D	0	7	

How does the algorithm work?

When a new process (customer) enters the system, it (he) must declare the maximum number of instances of each resource type (credit units) that it (he) may need. This number may not exceed the total number of resources (credit units) in the system. When a user (customer) requests a set of resources (credit unit), 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.

Consider current allocation to various processes is as shown below.

Process	Current	Max.	Free = 2
A	1	6	
B	1	5	
C	2	4	
D	4	7	

Would the System be in a safe state?

'C' requests 2 additional units and gets them. It then runs to completion and frees all the resources it has.

Process	Current	Max.	Free = 4
A	1	6	
B	1	5	
C	0	-	
D	4	7	

Now either 'B' or 'D' can request and run to completion. Assume 'B' requests 4 additional units and gets them. It then runs to completion and frees all its resources.

Process	Current	Max.	Free = 5
A	1	6	
B	0	-	
C	0	-	
D	4	7	

Now 'D' runs and requests 3 additional resources and gets them. It then runs to completion and releases all its resources.

Process	Current	Max.	Free = 9
A	1	6	
B	0	-	
C	0	-	
D	0	-	

Finally 'A' runs and requests 5 additional resources and gets them. It then runs to completion and releases all its resources.

Process	Current	Max.	Free = 10
A	0	-	
B	0	-	
C	0	-	
D	0	-	

Here is the complete banker's algorithm:

The banker's algorithm requires the following data structures to be defined:

Available: A vector of length m indicates the number of available resources of each type. If **Available** [j] = k, there are k instances of resource type Rj available.

Max: An n * m matrix that defines the maximum demand of each process.

If **Max** [I,j] = k, then process Pi may request at most k instances of resource type Rj.

Allocation: An n * m matrix defines the number of resources of each type currently allocated to each process. If **Allocation** [I, j] = k, then process Pi is currently allocated k instances of resource type Rj.

Need: An n * m matrix indicates the remaining resource need of each process. If **Need** [I,j] = k, then Pi may need k more instances of resource type Rj to complete its task..

Note that Need [I, j] = Max [I, j] – Allocation [I,j].

Having defined the data structures, the algorithm now proceeds in two phases:

- Safety Algorithm
- Resource Request Algorithm

Safety Algorithm

As discussed earlier, the safety algorithm is for finding out whether or not a system is in a safe state. It is described below:

3. Let **work** and **finish** be vectors of length m and n respectively. Initialize **work** = **Available** and **Finish**[I] = false for all I = 1, 2, …, n.
4. Find an I such that both
   - Finish[I] = false
   - $Need_i$ <=work

   If no such I exists, go to step 4.

5. work = **work** + $\mathbf{allocation}_i$

finish[I] = true

go to step 2

6. If **finish**[I] = true for all I, then the system is in a safe state.

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

Resource Request Algorithm

Having determined that the system is safe, this algorithm grants the requested resources to the process.

Let $Request_i$ be the request vector for process P_i. If $Request_i[j] = k$, then process Pi wants k instances of resource type Rj. When this request is made, the following actions are taken:

1. If **request** $_I$ <= **need** $_I$, then go to step 2. Otherwise raise an error condition because 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:

a. Available = **available** – **request** $_I$

b. Allocation = **allocation** + **request** $_I$

c. Need $_I$ = **Need** $_I$ – **request** $_I$

4. Call the Safety algorithm. If the state is safe, then transaction is completed and process P_i is allocated the resources. If the new state is unsafe, then P_i must wait and the old resource allocation state is restored.

Exercise:

Consider a system with five processes P_0 through P_4 and three resource types A, B and C. A has 10 instances, B has five instances and C has seven instances. Consider the following snapshot of the system:

Process	**Allocation**			**Max**			**Available**		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

Assume that the system is in a safe state. (prove it!).

Suppose now that process P1 requests one additional instance of resource type A and two instances of resource type C. So $Request_1 = (1,0,2)$. Apply banker's algorithm to determine if the request can be granted.

Deadlock detection and recovery

If a system does not employ either deadlock prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. The system should first detect the deadlock and then try to recover from it.

Deadlock in a system can be detected by finding a cycle in a graph of resource requests.

What is this graph of resource requests?

It is also known as a DRAG (Directed resource allocation graph). The DRAG is a:

- Directed Graph
- Consists of set of vertices V and a set of edges E
- V is partitioned into two sets:
- P (set of all processes) = {P1, P2, P3, … , Pn}
- R (set of all resources) = {R1, R2, R3, … , Rm}
- A directed edge from process Pi to resource Rj is denoted by: Pi à Rj (known as **request edge)**
- A directed edge from resource Rj to process Pi is denoted by: Rjà Pi (known as **assignment edge**)
- Each process Pi is represented by a **circle** and each resource type Rj is represented by a **square**.
- Multiple instances of a resource type are represented by **'dots'**

The notation Rj ——> Pi indicates:

"Process Pi has been allocated an instance of resource type Rj."

An assignment edge always points to a Circle (Pi) in the DRAG and must also designate one of the 'dots'.

When a process releases a resource, the assignment edge is deleted.

The notation Pi ——> Rj indicates:

"Process Pi has requested for an instance of resource type Rj and is currently waiting."

Request Edge always points to a 'Square' (Rj) in a DRAG.

When a process Pi requests an instance of a resource Rj, a request edge is inserted into the DRAG and after the request is fulfilled, the edge is converted into an assignment edge.

If a DRAG has no cycles, no process is deadlocked.

If a DRAG has a cycle, deadlock may exist.

If each resource type has only one instance, then a cycle implies that deadlock occurred. (Necessary and sufficient condition)

If each resource type has several instances, then a cycle does not necessarily imply a deadlock has occurred.

Example Of A Drag

There are two minimal cycles: P1—>R1—>P2—>R3—>P3—>R2—>P1 and P2—>R3—>P3—>R2—>P2.

Processes P1, P2, P3 are deadlocked because:

P2 is waiting for R3, which is held by P3.

P3 is waiting for R2 which is held by P1 and P2 and

P1 is waiting for R1, which is held by P2

Once you detect a deadlock using the DRAG, how do you recover from it?

Once you have discovered a deadlock, you have to figure out how to break it. This involves preempting a resource, which might mean canceling a process and starting it over.

Deadlock detection and recovery is the optimistic solution to the problem. You assume deadlock is unlikely, but detect it and recover from it when it occurs rather than spending resources trying to prevent it or avoid it.

Review Questions:

1. List three examples of deadlocks that are not related to computer-system environment.

2. Is it possible to have a deadlock involving only one single process? Explain your answer.
3. Consider the following snapshot of a system:

Process	Allocation	Max	Available
A B C D	A B C D		A B C D
P0	0 0 1 2	0 0 1 2	1 5 2 0
P1	1 0 0 0		1 7 5 0
P2	1 3 5 4		2 3 5 6
P3	0 6 3 2		0 6 5 2
P4	0 0 1 4		0 6 5 6

a) What is the content of the matrix **need**?

b) Is the system in a safe state?

c) If a request from the process P1 arrives for (0,4,2,0) can the request be granted immediately?

Check Your Progress

Explain Deadlock Avoidance Procedure?

_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems / Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

7.1 List types of resources we might consider in deadlock problems on computers.

Answer: CPU cycles, memory space, files, I / O devices, tape drives, printers.

7.2 Define deadlock.

Answer: A situation where every process is waiting for an event that can be triggered

only by another process.

7.3 What are the four necessary conditions needed before deadlock can occur?

Answer:

a. At least one resource must be held in a nonsharable mode.

b. A process holding at least one resource is waiting for more resources held by other processes.

c. Resources cannot be preempted.

d. There must be a circular waiting.

7.4 Give examples of sharable resources.

Answer: Read-only files, shared programs and libraries.

7.5 Give examples of nonsharable resources.

Answer: Printer, magnetic tape drive, update-files, card readers.

7.6 List three overall strategies in handling deadlocks.

Answer:

a. Ensure system will never enter a deadlock state.

b. Allow deadlocks, but devise schemes to recover from them.

c. Pretend deadlocks don't happen.

7.7 Consider a traffic deadlock situation

a. Show that the four necessary conditions for deadlock indeed hold in this example.

b. State a simple rule that will avoid deadlocks in this system.

Answer:

a. Each section of the street is considered a resource.

- **Mutual-exclusion** —only one vehicle on a section of the street.
- **Hold-and-wait** — each vehicle is occupying a section of the street and is waiting to move to the next section.
- **No-preemption** — a section of a street that is occupied by a vehicle cannot be taken away from the vehicle unless the car moves to the next section.
- **Circular-wait** —each vehicle is waiting for the next vehicle in front of it to move.

b. Allow a vehicle to cross an intersection only if it is assured that the vehicle will not have to stop at the intersection.

7.8 List the data structures needed for the banker's algorithm.

Answer:

- available vector *Available*(m)
- demand matrix *Max*(n,m)
- allocation matrix *Allocation*(n,m)
- need matrix *Need*(n,m)

7.9 Summarize the banker's algorithm.

Answer:

a. If request for process *i* exceeds its need, error has occurred.

b. If request of process *i* exceeds available resources, process *i* must wait.

c. The system temporarily allocates the resources process *i* wants; if the state is unsafe, the allocation is postponed.

7.10 Summarize the Safety Algorithm.

Answer:

a. Initialize vector *Work* to *Available* and set vector *Finish* to false.

b. Find a process such that *Finish*(i) = false and *Need*(i) *leq Work*.

c. If found, add *Allocation*(i)to*Work*(i), *Finish*(i)to true,and go to step b.

d. If not found, continue here. If *Finish*(i) = true for all processes then state is safe, else it is unsafe.

7.11 How can we determine whether current state is "safe" in systems with only one instance of each resource type?

Answer: State is unsafe if any cycle exists.

7.12 What conditions must exist before a wait-for graph is useful in detecting deadlocks?

Answer: A cycle.

7.13 What does a cycle in a wait-for graph indicate?

Answer: A deadlock.

7.14 Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock-free.

Answer: Suppose the system is deadlocked. This implies that each process is holding one resource and is waiting for one more. Since there are three processes and four resources, one process must be able to obtain two resources. This process requires no more resources and therefore it will return its resources when done.

7.16 What is starvation?

Answer: System is not deadlocked, but at least one process is indefinitely postponed..

7.17 List three options for breaking an existing deadlock.

Answer:

a. Violate mutual exclusion, risking data.

b. Abort a process.

c. Preempt resources of some process.

7.18 What three issues must be considered in the case of preemption?

Answer:

a. Select a victim to be preempted.

b. Determine how far back to rollback the victim.

c. Determine means for preventing that process from being "starved."

Notes

Objectives

In the last two lectures, you learnt about deadlocks, their characterization and various deadlock-handling techniques. In this lecture, you will learn about memory management, swapping and concept of contiguous memory allocation.

Memory Management is also known as Storage or Space Management

What is Memory Management?

Memory management involves

- Subdividing memory to accommodate multiple processes
- Allocating memory efficiently to pack as many processes into memory as possible

When is address translation performed?

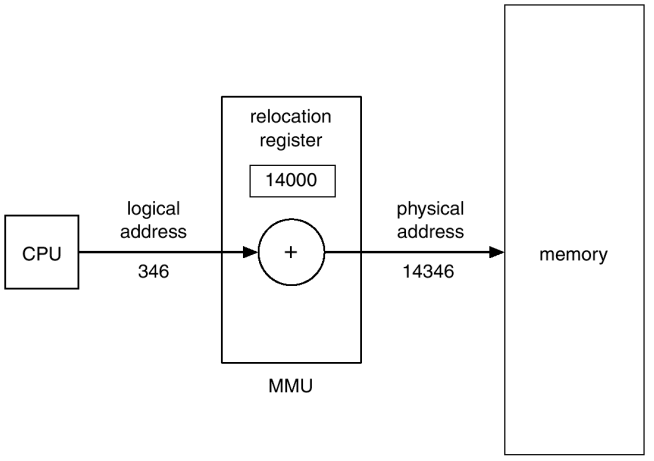
1. At compile time
- Primitive.
- Compiler generates *physical* addresses.
- Requires knowledge of where the compilation unit will be loaded.
- Rarely used (MSDOS .COM files).

2. At link-edit time (the "linker lab")
- Compiler
  - Generates relocatable addresses for each compilation unit.
  - References external addresses.
- Linkage editor
- Converts the relocatable addr to absolute.
- Resolves external references.
- *Misnamed* ld by unix.
- Also converts virtual to physical addresses by knowing where the linked program will be loaded. Linker lab "does" this, but it is trivial since we assume the linked program will be loaded at 0.
- Loader is simple.
- Hardware requirements are small.
- A program can be loaded only where specified and cannot move once loaded.
- Not used much any more.

3. At load time
- Similar to at link-edit time, but do not fix the starting address.
- Program can be loaded anywhere.
- Program can move but cannot be split.
- Need modest hardware: base/limit registers.
- Loader sets the base/limit registers.

4. At execution time
- Addresses translated dynamically during execution.
- Hardware needed to perform the virtual to physical address translation quickly.
- Currently dominates.
- Much more information later.

Now I will explain you MMU - Logical vs. Physical Address Space



- Concept of logical address space bound to a separate physical address space - central to proper memory management
  - Logical (virtual) address – generated by the CPU
  - Physical address – address seen by the memory unit
  - Logical and physical addresses:
- Same in compile-time and load-time address-binding schemes
  - Different in execution-time address-binding scheme
  - Memory Management Unit: HW device that maps virtual to physical address
  - Simplest scheme: add relocation register value to every address generated by process when sent to memory

Dynamic Loading

- Routine is not loaded until it is called.
- Better memoryspace utilization; unused routine is never loaded.
- Useful when large amounts of code are needed to handle infrequently occurring cases.
- No special support from the operating system is required; implemented through program design.

Dynamic Linking

- Linking postponed until execution time.
- Small piece of code, *stub*, used to locate the appropriate memoryresident library routine.
- Stub replaces itself with the address of the routine, and executes the routine.
- Operating system needed to check if routine is in processes' memory address.

Overlays

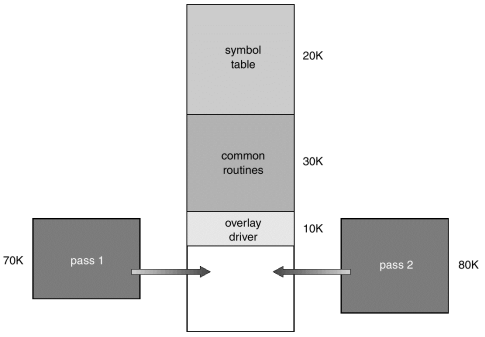
- To handle processes larger than their allocated memory
- Keep in memory only instructions and data needed at any given time
- Implemented by user, no special support needed from OS, programming design is complex

The Need for Memory Management

Overlay for a two-pass assembler:

Pass 1	70KB
Pass 2	80KB
Symbol Table	20KB
Common Routines	30KB
Total	200KB

Two overlays: 120 + 130KB



Main memory is generally the most critical resource in a computer system in terms of the speed at which programs run and hence it is important to manage it as efficiently as possible.

What are the requirements of Memory Management?

The requirements of memory management are:

- Relocation
- Protection
- Sharing
- Logical Organization
- Physical Organization

What is meant by relocation?

- Programmer does not know where the program will be placed in memory when it is executed
- While the program is executing, it may be swapped to disk and returned to main memory at a different location (**relocated**)
- Memory references must be translated in the code to actual physical memory address

What is meant by protection?

- Processes should not be able to reference memory locations in another process without permission
- Impossible to check absolute addresses in programs since the program could be relocated
- Must be checked during execution

Operating system cannot anticipate all of the memory references a program will make

What does sharing mean?

Allow several processes to access the same portion of memory

Better to allow each process (person) access to the same copy of the program rather than have their own separate copy

What does logical organization of memory mean?

- Programs are written in modules
- Modules can be written and compiled independently
- Different degrees of protection given to modules (read-only, execute-only)
- Share modules
- What does physical organization of memory mean?

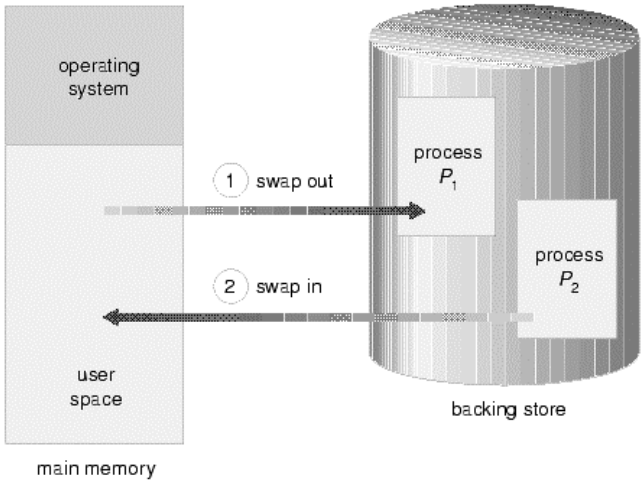
•Memory available for a program plus its data may be insufficient

•Overlaying allows various modules to be assigned the same region of memory

•Programmer does not know how much space will be available

Swapping

Swapping is the act of moving processes between memory and a backing store. This is done to free up available memory. Swapping is necessary when there are more processes than available memory. At the coarsest level, swapping is done a process at a time. That is, an entire process is swapped in/out.



What are the various memory management schemes available?

There are many different memory management schemes. Selection of a memory management scheme for a specific system depends

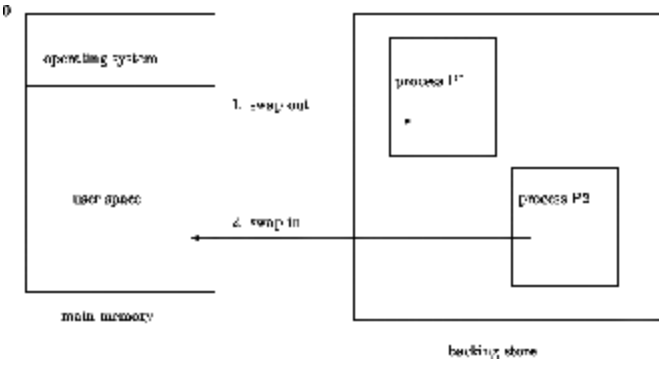
on many factors, especially the hardware design of the system. A few of the schemes are given here:

- Contiguous, Real Memory Management System
- Non-contiguous Real Memory Management System
- Non-contiguous, Virtual Memory Management System

In this lecture, you will learn about the **contiguous memory management scheme**. You will also learn about virtual memory and concept of **swapping**.

First let me explain what swapping means. You are all aware by now that for a process to be executed, it must be in the memory. Sometimes, however, a process can be swapped (removed) temporarily out of the memory to a backing store (such as a hard disk) and then brought back into memory for continued execution.

Let me explain with an example:



Consider a multiprogramming environment with a round-robin CPU scheduling algorithm. When a quantum (time-slice) expires, the memory manger will start to **swap out** processes that just finished, and to **swap in** another process to the memory space that has been freed. In the meantime, the CPU scheduler will allocate a time slice to some other process in memory. Thus when each process finishes its quantum, it will be swapped with another process.

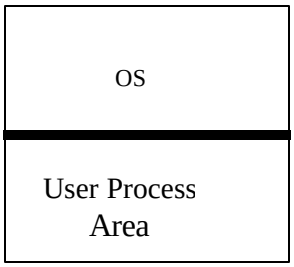
Are there any constraints on swapping?

Yes, there are. If you want to swap a process, you must be sure that it is completely idle. If a process is waiting for an I/O operation that is asynchronously accessing the user memory for I/O buffers, then it cannot be swapped.

Having learnt about the basics of memory management and concept of swapping, we will now turn our attention to the **contiguous memory management** scheme.

What is the meaning of the term contiguous?

Contiguous literally means adjacent. Here it means that the program is loaded into a series of adjacent (contiguous) memory locations. In contiguous memory allocation, the memory is usually divided into two partitions, one for the OS and the other for the user process.



At any time, only one user process is in memory and it is run to completion and then the next process is brought into the memory. This scheme is sometimes referred to as the

Single Contiguous Memory Management.

What are the advantages and disadvantages of this scheme?

First, let us look at the advantages:

- Starting physical address of program is known at compile time
- Executable machine code has absolute addresses only. They need not be changed/translated at execution time
  - Fast access time as there is no need for address translation·
- Does not have large wasted memory
- Time complexity is small·

The disadvantage is that, it does not support multi-programming and hence no concept of sharing.

What about protection?

Since there is one user process and the OS in the memory, it is necessary to protect the OS code from the user code. This is achieved through two mechanisms:

- Use of Protection Bits
- Use of Fence Register

Protection Bits:

- One bit for each memory block
- The memory block may belong to either user process or the OS
- Size of memory block should be known
- The bit is 0 if the word belongs to OS

The bit is 1 if the word belongs to user process

A mode bit in the h/w indicates if system is executing in **privileged mode** or **user mode.**

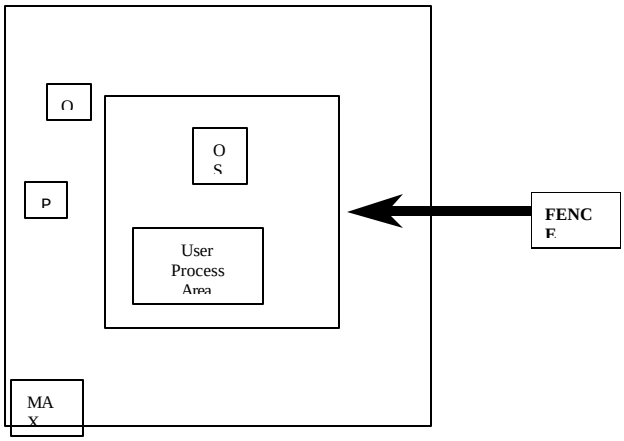
- If mode changes, the h/w mode bit is also changed automatically.
- If user process refers to memory locations inside OS area, then the protection bit for the referred word is 0 and the h/w mode bit is 'user mode'. Thus user process is prevented from accessing OS area.
- If OS makes a reference to memory locations being used by a user process then the mode bit = 'privileged' and the protection bit is not checked at all.·

Y

Current process	mode bit	prot. bit	access	status
USER		u-mode	0	OS
USER		u-mode	1	user
OS		p-mode	1	user
OS		p-mode	0	OS

Fence Register:

- Similar to any other register in the CPU
- Contains address of the 'fence' between OS and the user process (see Fig. 2)
- Fence Register value = P
- For every memory reference, when final address is in MAR (Memory Address Register), it is compared with Fence Register value by h/w thereby detecting protection violations



In a multi-programming environment, where more than one process is in the memory, we have the fixed-partition scheme.

In this scheme,

- Main memory is divided into multiple partitions
- Partitions could be of different sizes but 'fixed' at the time of system generation
- Could be used with or without 'swapping' and 'relocation'
- To change partition sizes, system needs to be shut down and generated again with a new partition size

Review Questions:

1. What is Memory Management?

_____
_____
_____
_____
_____
_____

2. What is various type of addressing?

_____
_____
_____
_____

_____
N
_____
Y
Y
Y

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems / Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

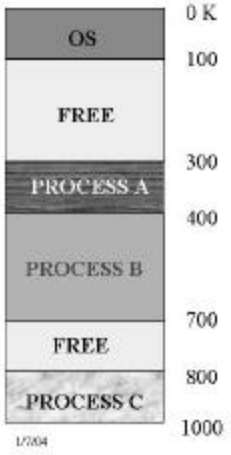
Objectives

In the last lecture, you have learnt about memory management, swapping and concept of contiguous memory allocation. In this lecturer you are going to learn about how OS manages the memory partitions.

So how does the OS manage or keep track of all these partitions?

In order to manage all the partitions,

- The OS creates a Partition Description Table (PDT)
- Initially all the entries in PDT are marked as 'FREE'
- When a partition is loaded into one of the partitions, the 'status' column is changed to 'ALLOC'



OS/012

Figure 3

The Partition Description Table

Pt.ID	PARTITION		
	Start Address	Size	Status
0	0	100	Alloc
1	100	200	Free
2	300	100	Alloc
3	400	300	Alloc
4	700	100	Free
5	800	200	Alloc

- The PCB of each process contains the Id. of the partition in which the process is running

How are the partitions allocated to various processes?

The sequence of steps leading to allocation of partitions to processes is given below:

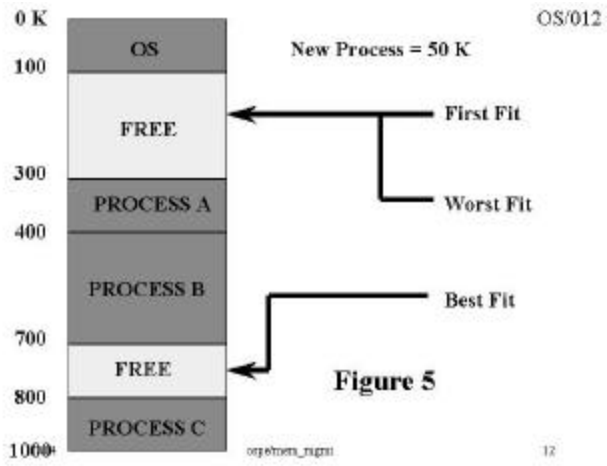
- The long-term scheduler of the process manager decides which process is to be brought into memory next.
- It finds out the size of the program to be loaded by consulting the Information Manager of the OS (the compiler keeps the size of the program in the header of the executable code)
- It then makes a request to the 'partition allocate routine' of the memory manager to allocate free partition of appropriate size.
- It now loads the binary program in the allocated partition (address translation may be necessary)
- It then makes an entry of the partition-id in the PCB before the PCB is linked to chain of ready processes by using the Process Manager module.
- The routine in the Memory Manager now marks the status of that partition as allocated.
- The Process Manager eventually schedules the process

Can a process be allocated to any partition?

The processes are allocated to the partitions based on the allocation policy of the system. The allocation policies are:

- First Fit
- Best Fit
- Worst Fit
- Next Fit

Let me explain this with a simple example:



OS/012

Figure 5

Refer to figure above.

Free partitions are 1 and 4.

So, which partition should be allocated to a new process of size 50K?

First Fit and **Worst Fit** will allocate Partition 1 while **Best Fit** will allocate Partition 4.

Do you know why?

In first fit policy, the memory manager will choose the first available partition that can accommodate the process even though its size is more than that of the process.

In worst fit policy, the memory manager will choose the largest available partition that can accommodate the process.

In best-fit policy, the memory manager will choose the partition that is just big enough to accommodate the process

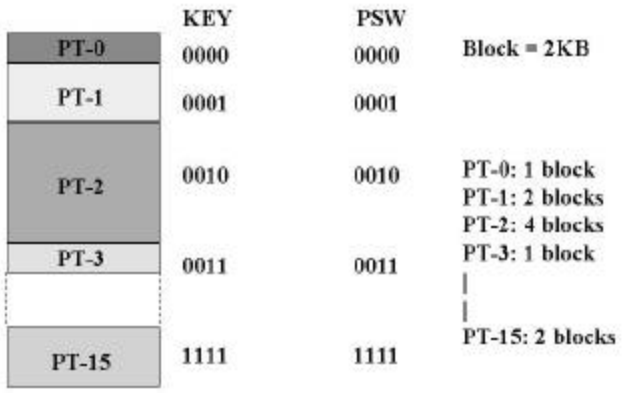
Are there any disadvantages of this scheme?

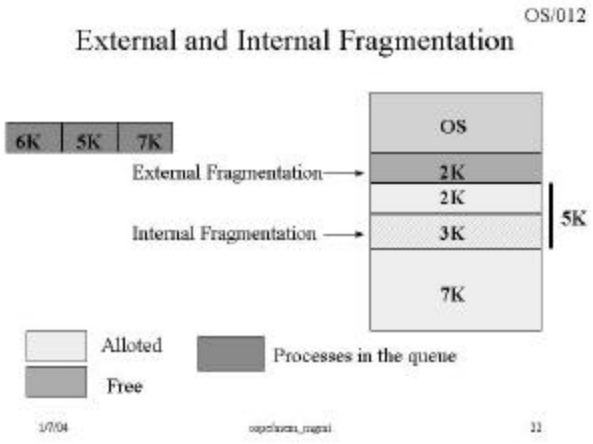
Yes. This scheme causes wastage of memory, referred to as fragmentation.

Let me explain with an example:

Suppose there is a process, which requires 20K of memory. There is a partition of size 40K available. Assuming that the system is following the First-fit policy, then this partition would be allocated to the process. As a result, 20K of memory within the partition is unused. This is called **internal fragmentation**.

Now consider the same 20K process. This time, though there are three partitions of 10K, 5K and 16K available. None of them are large enough to accommodate the 20K process. There are no other smaller processes in the queue. Hence these three partitions remain unused. This is waste of memory and is referred to as **external fragmentation**.





How do you ensure protection of processes in such a scheme?

Protection can be achieved in two ways:

- Protection Bits (used by IBM 360/370 systems)
- Limit Register

Protection Bits:

- Divide the memory into 2 KB blocks.
- Each block has 4 bits reserved for protection called the 'key'.
- Size of each partition had to be multiple of such 2K blocks.
- All the blocks associated with a partition allocated to a process are given the same key.

So how does this mechanism work?

Let me explain with the following example:

Consider a physical memory of 64 KB. Assume each block is of 2KB.

- Total No. of blocks = 64/2 = 32 blocks
- 'Key' associated with each block is 4 bits long
- 'Key string' for 32 blocks is therefore 128 bits long
- System administrator defines a max of 16 partitions of different sizes (out of the available 32 blocks)
- Each partition is then given a protection key in the range 0000 to 1111

Now a process is loaded into a partition

- 'Protection key' for the partition is stored in the **PSW** (Program Status Word).
- The process makes a memory reference in an instruction.
- The resulting address and the block are computed.
- The 4-bit protection key for that block is extracted from the protection-key string.
- It is then tallied with the value in PSW.
- If there is a match, then fine!
- Else the process is trying to access an address belonging to some other partition.

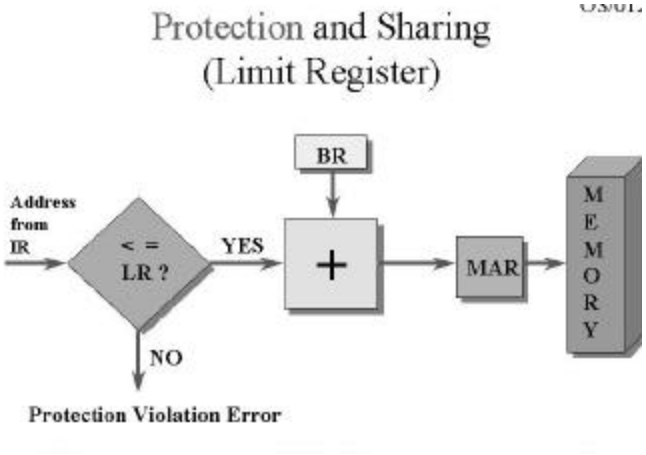
What are the disadvantages of this mechanism?

- Memory wastage due to internal fragmentation
- Limits maximum number of partitions (due to key length)
- Hardware malfunction may generate a different address but in the same partition - scheme fails!!

Limit Register:

- The Limit Register for each process can be stored in the PCB and can be saved/restored during context switch.
- If the program size were 1000, logical addresses generated would be 0 to 999
- The Limit Register therefore is set to 999

- Every 'logical' or 'virtual' address is checked to ensure that it is <= 999 and then added to base register. If not, then hardware generates an error and process is aborted.



Review Questions:

1. Explain the difference between Internal and external fragmentation.

_____

_____

_____

_____

_____

_____

2. Explain the following allocation algorithms:

a. First-Fit b. Best-Fit c. Worst-fit

_____

_____

_____

_____

_____

_____

3. What are the advantages and disadvantages of the fixed partition memory management scheme?

_____

_____

_____

_____

_____

_____

_____

Reference Books:

Author Dahmke, Mark.
Main Title Microcomputer Operating Systems / Mark Dahmke.
Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-
Main Title An Introduction To Operating Systems / Harvey M. Deitel.
Edition Rev. 1st Ed.
Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-
Main Title Fundamentals Of Operating Systems / A.M. Lister.
Edition 3rd Ed.
Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)
Main Title Introduction To Computer Systems / N.A.B. Gray.
Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.
Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.
Edition 2nd Ed.
Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.
Main Title Operating Systems / William Stallings.
Edition 6th Ed.
Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-
Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.
Edition 2nd Ed.
Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.
Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.
Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.
Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.
Edition 6th Ed.
Publisher Reading, Mass. : Addison Wesley Longman, C1998.

Notes

Objectives

In the last two lectures, you will learnt about memory management, swapping and concept of contiguous memory allocation. In this lecturer you are going to learn multiprogramming environment using fixed and Dynamic partitions.

Multiprogramming With Fixed Partition

In a multiprogramming environment, several programs reside in primary memory at a time and the CPU passes its control rapidly between these programs. One way to support multiprogramming is to divide the main memory into several partitions each of which is allocated to a single process. Depending upon how and when partitions are created, there may be two types of memory partitioning:

(1) Static and

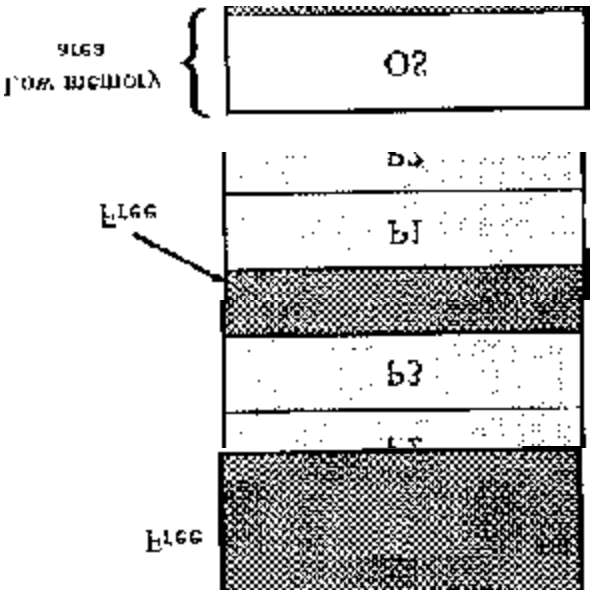
(2) Dynamic.

Static partitioning implies that the division of memory into number of partitions and its size is made in the beginning (during the system generation process) and remain fixed thereafter. In dynamic partitioning, the size and the number of partitions are decided during the run time by the operating system. In this section we will take up static partitioning and multiprogramming with dynamic (variable) partitioning will be discussed in the next section.

In this section, You present several memory management schemes based on contiguous allocation.

The basic approach here is to divide memory into several fixed size partitions where each partition will accommodate only one program for execution. The number of programs (i.e. degree of multiprogramming) residing in memory will be bound by the number of partition When a program terminates, that partition is free for another program waiting in a queue.

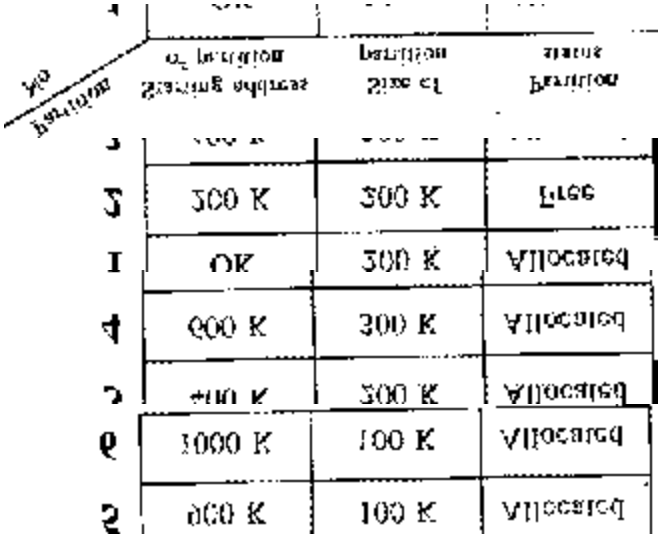
An example of partition memory is shown in figure.



Fixed Size Partition

As shown, the memory is partitioned into 6 regions. The first region Lower area) is reserved for operating system. The remaining five regions are for user programs. Three partitions occupied by programs P1, P2 and P3. Only the first and last one are free and available for allocation.

Once partitions are defined, operating system keeps track of status (whether allocated or free) of memory partitions. Ibis is done through a data structure called partition description table (figure).



Partition Description Table

The two most Common strategies to allocate free partitions to ready processes are: (i) first-fit and (ii) best-fit. The approach followed in the first fit is to allocate the first free partition large enough to accommodate the process. The best fit approach on the other hand allocates the smallest free partition that meets the requirement of the process.

Both strategies require to scan the partition description table to find out free partitions. However, the first-fit terminates after finding the first such partition whereas the best-fit continues searching for the near exact size.

As a result, the first-fit executes faster whereas the best- fit achieves higher utilization of memory by searching the smallest free partition. Therefore, a trade-off between execution speed of first-fit and memory utilization of best-fit must be made.

To explain these two strategies, let us take one example. A new process P4 with size 80K is ready to be allocated into memory whose partition layout is given in figure 2. Using first-fit strategy, P4 will get the first partition leaving 120K of unused memory. Best fit will continue searching for the best possible partition and allocate the last partition to the process leaving just 20K bytes of unused memory.

Wherever a new process is ready to be loaded into memory and if no partition is free, swapping of processes between main memory and secondary storage is done. Swapping helps in CPU utilization by replacing suspend able processes but residing into main memory with ready to execute processes from secondary storages. When the scheduler admits a new process (of high priority) for which no partition is free, a memory manager is invoked to make a partition free to accommodate the process.

The memory manager performs this task by swapping out low priority processes suspended for a comparatively long time in order to load and execute the higher priority process. When the higher priority process is terminated, the lower priority process can be swapped back and continued.

Swapping requires secondary storage device such as fast disk to store the suspended processes from main memory. One problem with swapping process is that it takes lengthy time to access process from secondary storage device. For example, to get an idea of total swap time, assume that user program is 100K words and secondary storage device is a fixed head disk with an average latency of 8m sec and a transfer rate of 250,000 words/second then a transfer of 100K words to or from memory takes:

8msec + (100K words/250,000 words/sec)

= 8msec + 100000 words/250,000 words/sec

= 8msec + 215 sec

= 8msec + 2000/5 msec

= 8msec + 400 msec

= 408 msec (approximately)

Since you must swap in and swap out the total swap time is about 408+408 = 816msec.

The overhead must be considered when deciding whether to swap a process in order to make room for another process.

One important issue concerning swapping is whether the process removed temporarily from any partition should be brought back to the same partition or any partition of adequate size.

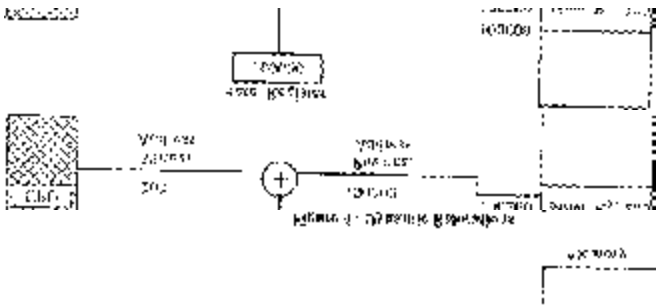
This is dependent upon partitioning policy. The binding of process to a specific partition (static partitioning) eliminates overhead of run time allocation of partition at the expense of lowest utilization of primary memory. On the other hand, systems where processes are not permanently bound to a specific partition (dynamic partition) are much more flexible and utilizes memory more efficiently. The only drawback with dynamic partitioning approach is run time overhead of partition allocation whenever a new process is swapped in.

As said earlier, the loading of a process into a same partition where from it was swapped out into a different partition is dependent upon relocation policy. The term relocation usually refers to the ability to load and execute a given program into an arbitrary memory partition as opposed to fixed set of memory locations specified at program translation time.

Depending upon when and how the addresses translation from the virtual address to actual address (also called physical address) of primary memory, takes place, process or program relocation may be regarded as static relocation and dynamic relocation. There is a difference between virtual and physical address. Virtual address refers to information within a program's address space, while physical address specifies the actual physical memory locations where program and data are stored in memory during execution time.

If the relocation is performed before or during the loading of a program into memory by a relocating linker or a relocating loader, the relocation approach is called static relocation. The static relocation is practically restricted to support only static binding of processes to partition.

Dynamic relocation refers to run-time mapping of virtual address into physical address with the support of some hardware mechanism such as base registers and limit registers. Relocation of memory references at run-time is illustrated in the following figure:

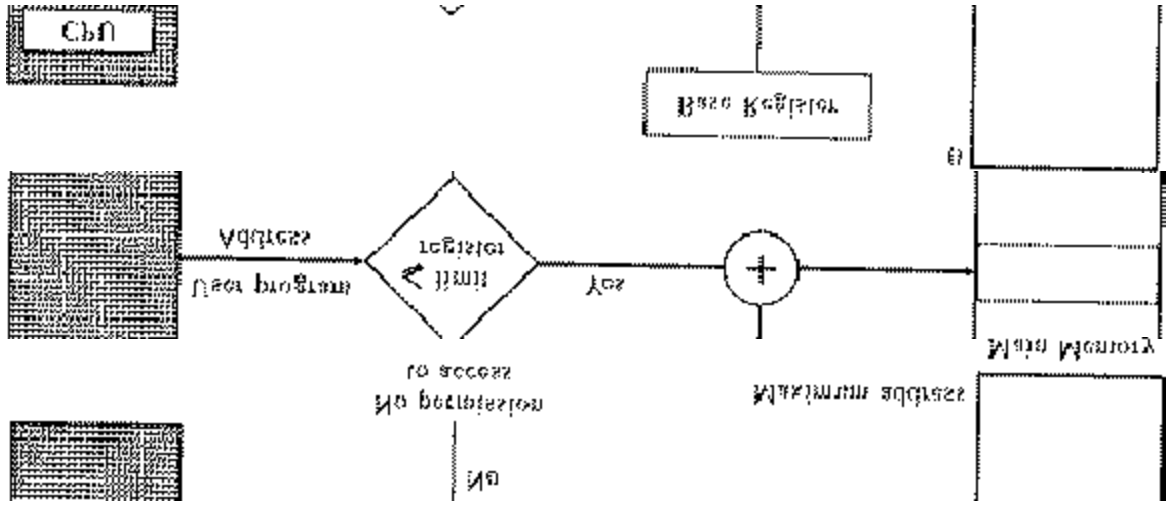


When a process is scheduled, the base register is loaded with the starting address. Every memory address generated automatically has the base register contents added to it before being sent to main memory. Thus, if the base register is 100000 (IOOK), a MOVE R1, 200 which is supposed to load the contents of virtual address 200 (relative to program beginning) into register, effectively turned into a MOVE R1, 100000 + 200, without the instruction itself being modified. The hardware protects the base register to prevent user programs from modifying.

An additional advantage of using a base register for relocation is that a program can be moved anywhere in memory after it has started execution.

Protection and Sharing: Multiprogramming introduces one essential problem of protection. Not only that the operating system must be protected from user programs/processes but each user process should also be protected from maliciously accessing the areas of other processes.

In system that uses base register for relocation, a common approach is to use limit (bound) register for protection. The primary function of a limit register is to detect attempts to access memory location beyond the boundary assigned by the operating system. When a process is scheduled, the limit register is loaded with the highest virtual address in a program. As illustrated in figure 5, each memory access of a running program is first compared with the contents of the limit register. If it exceeds the limit register, no permission is given to the user process. In this way, any attempt to access a memory location beyond the boundary is trapped.



Protection through Limit and Base Register

In addition to protection, a good memory management mechanism must also provide for controlled sharing of data and code between cooperating processes. One traditional approach to sharing is to place data and code in a dedicated common partition. However, any attempt by a participating process to access memory outside of its own participation is normally regarded as a protection violation. In systems with protection keys, this obstacle may be circumvented by changing the keys of all shared blocks upon every process switch in order to grant access rights to currently running process.

Fixed partitioning imposes several restrictions:

- No single program/process may exceed the size of the largest partition in a given system.
- It does not support a system having dynamically data structure such as stack, queue, heap etc.
- It limits the degree of multiprogramming which in turn may reduce the effectiveness of short-term scheduling.

Multiprogramming With Dynamic Partitions

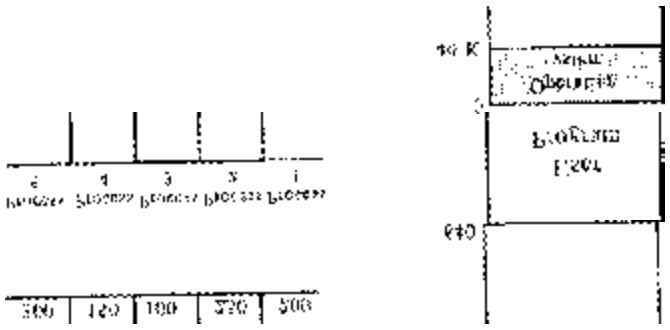
The main problem with fixed size partition is the wastage of memory by programs that are smaller than their partitions (i.e. internal fragmentation). A different memory management approach known as dynamic partitions (also called variable partition) which creates partitions dynamically to meet the requirements of each requesting process. When a process terminates or becomes swapped-out, the memory manager can return the vacated space to the pool of free memory areas from which partition allocations are made.

Compared to fixed partitions, in dynamic partitions, neither the size nor the number of dynamically allocated partition need be limited at any other time. Memory manager continues creating and allocating partitions to requesting processes until all physical memory is exhausted or maximum allowable degree of multiprogramming is reached.

The main difference between the fixed partition and variable partitions is that the number, location and size of partitions vary dynamically in the latter as processes are created and terminated, whereas they are fixed in the former. The flexibility of not being tied to a fixed number of partitions that may be too large or too small for requesting processes, improves memory utilisation but it also complicates the process of allocation and deallocation of memory.

In variable partition, operating system keeps track of which parts of memory are available and which are allocated.

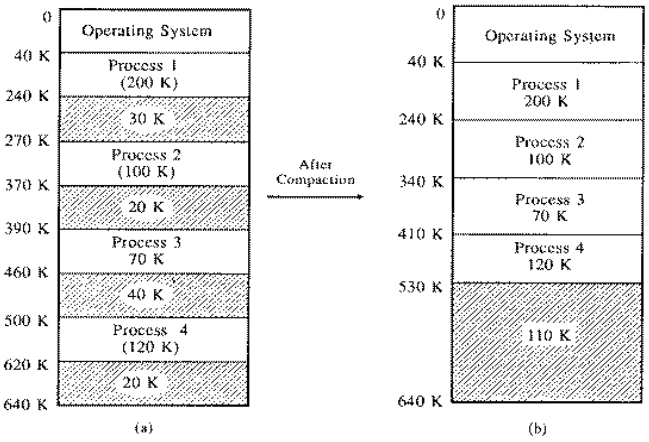
Assume that we have 640K main memory available in which 40K is occupied by operating system program. There are 5 jobs waiting for memory allocation in a job queue (figure). Applying FCFS scheduling policy, Process 1, Process 2 and Process 3 can be immediately allocated in memory. Process 4 cannot be accommodated because there is only 600 - 550 = 50K left for it. This situation is shown in figure (a)



Jobs waiting in a queue for CPU

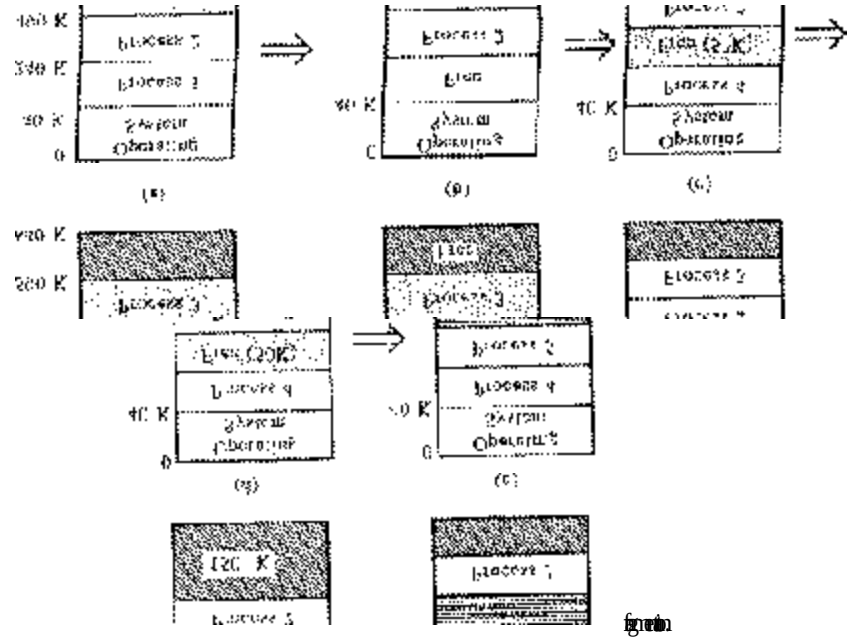
Memory allocation and Job scheduling

Let us assume that after some time Process 1 is terminated, releasing 200K memory space. This is shown in figure (b). After that the control is returned to the Process queue and next process (Process 4) is swapped, in the memory which is illustrated in figure (c). After Process 1, Process 3 gets terminated releasing IOOK memory (figure (d)) but Process 5 cannot be accommodated due to external holes of sizes 30K, 20K, 40K and 20K which have been compacted into one large hole or block of 110 K (figure (b)).



Compaction

Compaction is usually not done because it consumes a lot of CPU time; on a 1M Microcomputer that has the capacity of copying at a rate of 1 megabyte/sec, CPU takes one second to compact memory. It is usually done on a large machine like mainframe or supercomputer because they are supported with a special hardware to perform this task (compaction) at a rate of 40 megabytes/sec or more. Solution of an optimal compaction strategy is quite difficult. This is illustrated with the following example in figure 9.

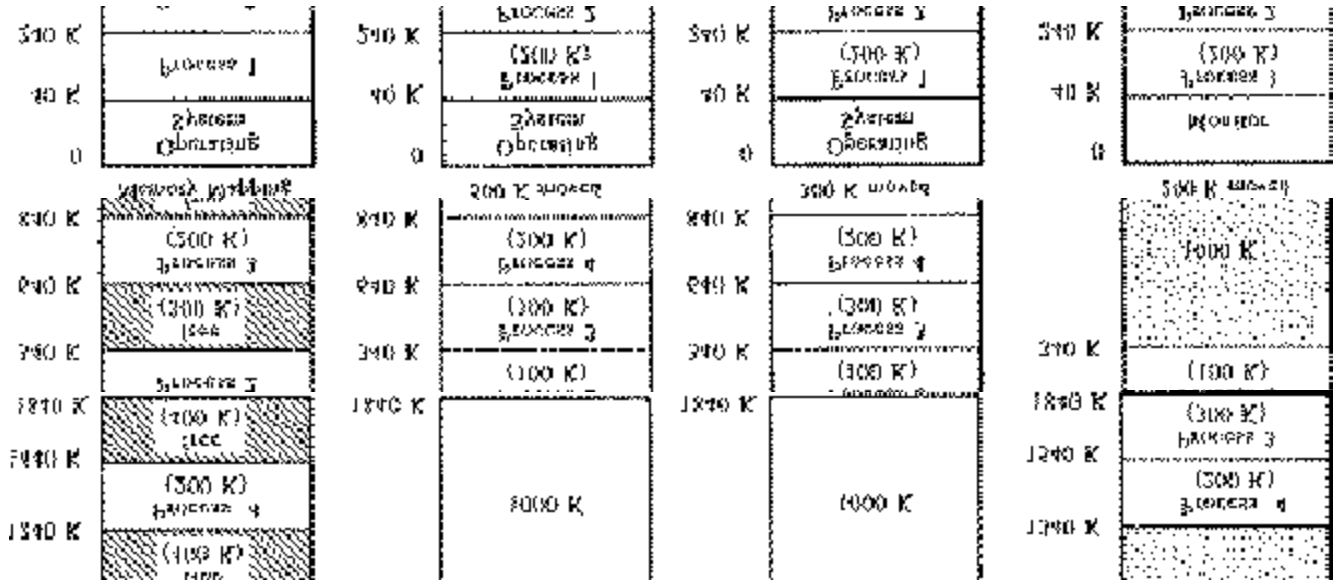


After

the swapping out of Process 2 due to termination, Process 5 will be loaded for execution.

This example illustrates one important problem with variable size partitions. The main problem is external fragmentation. It exists when the size of memory is large enough for a requesting process, but it cannot satisfy a request because it is not contiguous; storage is fragmented into a small number of holes (free spaces). Depending upon the total size of memory and number and size of a program, external fragmentation may be either a minor or a major problem.

One solution to this problem is compaction. It is possible to combine all the holes (free spaces) into a large block by pushing all the processes downward as far as possible. The following figure illustrates the compaction of memory. In figure (a) there are 4



Comparison of some different ways to Compact memory

If you apply simplest algorithm, Process 3 and 4 will be moved at the end, for a total movement of 500K creating a situation in figure (b). If you simply move Process 4 above 3, then you only 300K is moved or if you move Process 3 down below Process 4, then you move only 200K. Please observe that a large memory hole (block) is now in the middle. Therefore, if you have large number of processes, selection of which process or processes for shifting downwards or upwards to meet the requirement for waiting process is quite difficult task.

Advantages:

One advantage with variable partition is that memory utilization is generally better than fixed size partitions, since partitions are created accordingly to the size of process.

Protection and sharing in static and dynamic partitions are quite similar, because of same hardware requirement except for some additional consideration due to compaction of memory during dynamic partitioning.

One advantage of dynamic partitioning is to support processes whose memory requirement increase during their execution. In that case operating system creates a larger partition and moves a process into it. If there is an adjacent free area it simply expands it.

Disadvantages:

Dynamic memory management requires lots of operating system space, time, complex memory management algorithm and bookkeeping operation.

Compaction time is very high. Although internal fragmentation is negligible, external fragmentation may be severe problem imposing a time penalty for compaction.

Review Questions:

1. How to manage in the multiprogramming environment using fixed partitions

_____

_____

_____

_____

_____

_____

2. How to manage in the multiprogramming environment using dynamic partitions

_____

_____

_____

_____

_____

_____

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill / Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems / Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

Notes

Objectives

In the last lecture, you learnt about multiprogramming environment using fixed and Dynamic partitions. In this lecture, you will get to know about non-contiguous memory management scheme and the concept of Paging and Segmentation.

Let us start by defining non-contiguous memory. Non-contiguous memory means that the available memory is not contiguous but is distributed.

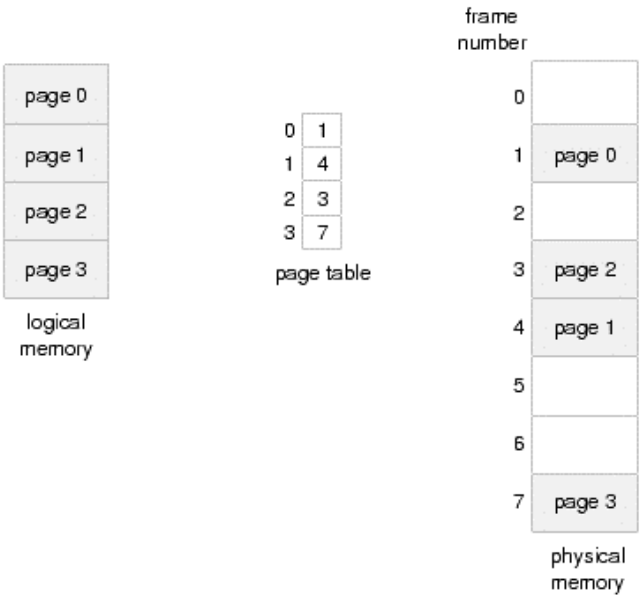
This scheme has the benefit of minimizing external fragmentation. How?

The logical address space of the process is allowed to be non-contiguous, thus allowing a process to be allocated physical memory wherever the later is available. This is achieved using a concept called **paging**.

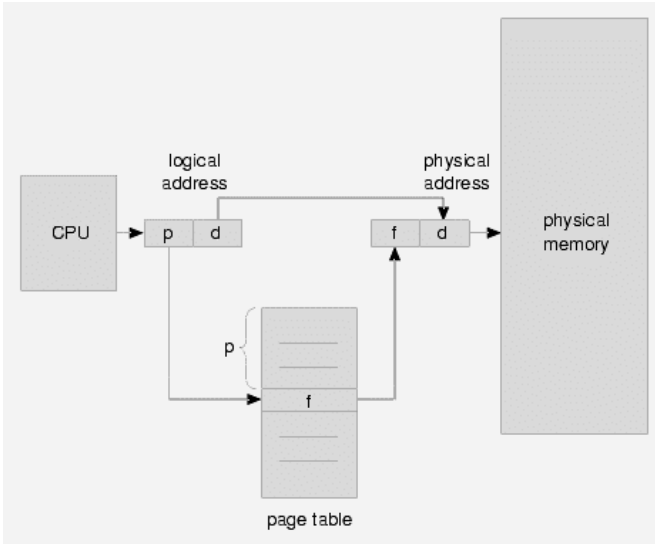
What is paging?

Paging is a process where by

- Physical memory is broken down into fixed size blocks called **page frames**.
- Logical memory is broken down into blocks of the same size as physical memory blocks and is called **pages.**
- When a process is to be executed, its pages are loaded from the **backing store** into any available memory frames.



Does this require any hardware support?
Yes. The picture below shows the support required.



- Every address generated by CPU is divided into two parts: ***page number (p) and page offset (d)***
- p is used as an index into a page map table
- Page table contains base address of each page in physical memory
- The base address is combined with d to define physical memory address that is then put into the MAR
- Compiler generates one-dimensional single address in binary
- Assume 16-bit addresses are used
- The address needs to be partitioned into P (page number) and D (offset)
- How many bits (out of 16) are needed for P and how many for D?
- Answer depends on page size.
- For example if page size is 1024 bytes, then we require 10 bits (210=1024) for D and remaining 6 for P

Here is a complete example illustrating paging:

Let Page size = 4 bytes and available physical memory = 32 bytes
We therefore have 8 frames and 8 pages as follows:

Page 0	addresses	0	-	3
Page 1	addresses	4	-	7
Page 2	addresses	8	-	11
Page 3	addresses	12	-	15
Page 4	addresses	16	-	19
Page 5	addresses	20	-	23
Page 6	addresses	24	-	27
Page 7	addresses 28 – 31			

To access 4 bytes within each page, we need 2 bits
To access 32 bytes of physical memory, we require 5 address lines
Hence, remaining 3 bits are used to access a total of 8 pages

Consider the logical addresses 0, 1, 2, 3 and the page table shown along side:

Logical address 0 is in page 0 at an offset 0 and page 0 is mapped to frame 5.

Since each frame is 4 bytes, the offset (start address) of frame 5 in physical memory is 5 x 4 = 20;

Therefore, logical address 0 corresponds to physical address 20 (5 x 4 + 0).

Similarly logical address 4 corresponds to physical address 24 (6 x 4 + 0).

How does the OS keep track of which pages are loaded into which frames?

The information is kept in a table called the Page Map Table (PMT). The table typically contains the following entries:

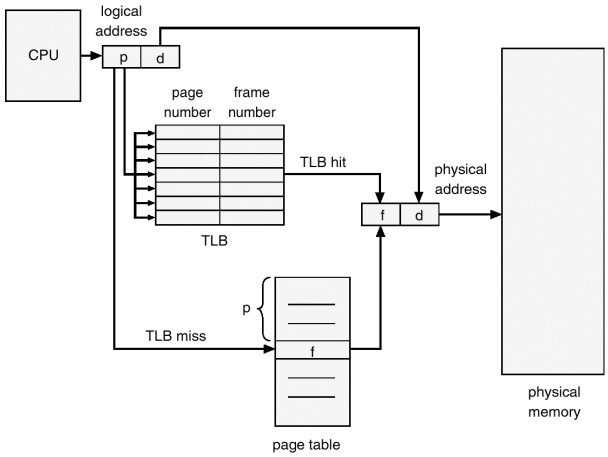
Page number; Frame Number, and the starting address of the Frame. The page table itself is loaded in the memory and its reference is kept in the Process Control Block. There is one page table per process.

How are pages mapped to the frames?

- I will answer this with an example.
- Memory manager keeps track of free page frames. Initially all page frames are free.
- Process Manager requests Information Manager for size of the program.
- Knowing the size, process manager requests memory manager to allocate memory of that size.
- Memory manager calculates number of page frames to be allocated using the formula: program_size/page_frame_size rounded to next integer.
- Memory manager consults list of page frames and if possible allocates to the process.
- Memory manager updates list of free page frames and makes these as 'allocated'. It then creates a PMT for the process.
- Memory manager signals Information Manager to load the process.
- Information manager loads various pages of process address space into allocated physical frames after which the process manager links the PCB for that process in the list of ready processes.
- The PCB maintains a pointer to starting address of PMT.

HW implementation of page table

- Keep page table in memory - dedicated set of registers
- Using Page Table Base Register (PTBR)
  - PTBR points to the page table
  - Page-table length register (PRLR) indicates size of page table
- Every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- **Solution:** use special fast-lookup hardware cache - associative memory or translation look-aside buffers (TLBs)



TLB or Associative Memory

- Associative registers - parallel search

Page #	Frame #

Address translation (A', A'')

- If A' is in associative register, get frame # out.
- Otherwise get frame # from page table in memory.

Effective Access Time

- Associative lookup = ***E*** time unit
- Assure memory cycle time is 1 microsecond
- *Hit ratio* - percentage of times that a page number is found in the associative registers; ratio related to number of associative registers.

Hit ratio = ***A***

Effective Access Time (EAT)

$$EAT = (1 + \boldsymbol{E})\boldsymbol{A} + (2 + \boldsymbol{E})(1 - \boldsymbol{A}) = 2 + \boldsymbol{E} - \boldsymbol{A}$$

Let us now discuss Memory Protection

- Memory protection implemented by associating protection bits with each frame.
- Valid-*invalid* bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page.
  - "invalid" indicates that the page is not in the process' logical address space.

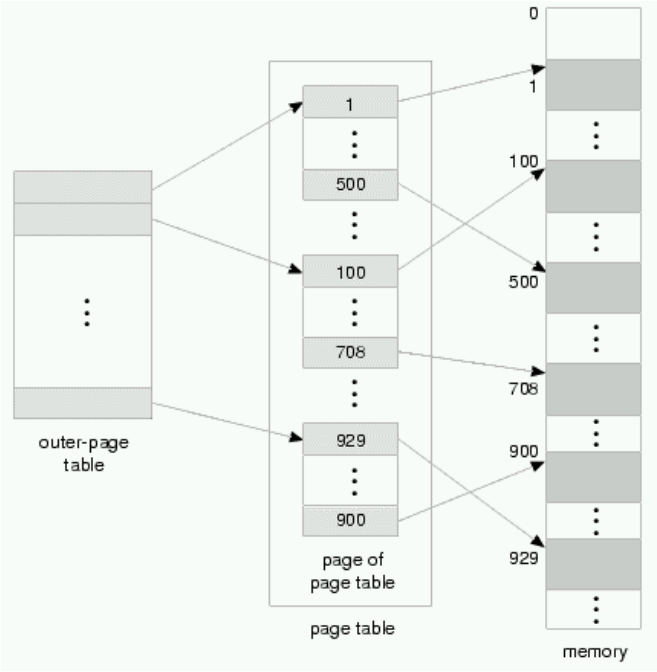
Page Table Structure

Logical address space is still too big – 64-bit architectures!

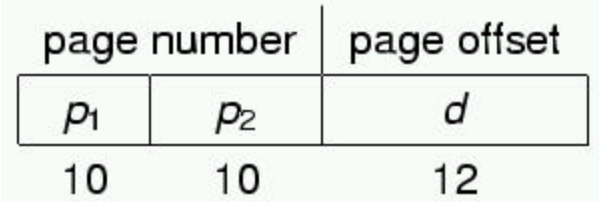
Managing big page tables:

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Table - Two-Level



- A logical address (on 32bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.
- Since the page table is paged, the page number is further divided into:
  - a 10bit page number.
  - a 10bit page offset.
- Thus, a logical address is as follows:

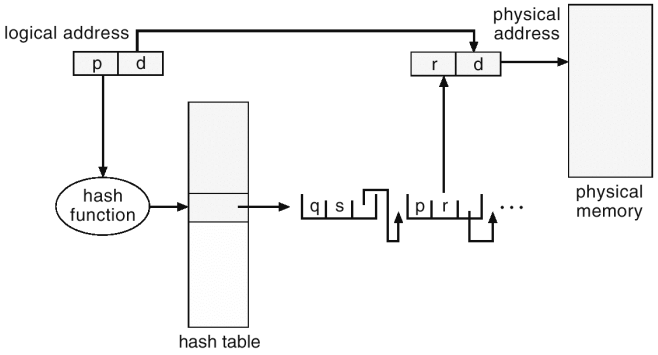


where p1 is an index into the outer page table, and p2 is the displacement within the page of the outer page table.

Hashed Page Tables

- Common in address spaces > 32 bits.
- The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

Having understood the basic concept of paging, we will now look at another concept called segmentation.



Inverted Page Table

- One entry for each real page of memory.
- Entry: ASID + virtual address of page stored in that real memory
- Decreases memory needed to store each page table
- Increases time needed to search table when a page reference occurs
- Use hash table to limit the search

Review Questions:

1. Explain Paging?

2. Describe the mechanism of translating a logical address to physical address in paging.

Reference Books:

Author Dahmke, Mark.
Main Title Microcomputer Operating Systems / Mark Dahmke.
Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-
Main Title An Introduction To Operating Systems / Harvey M. Deitel.
Edition Rev. 1st Ed.
Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-
Main Title Fundamentals Of Operating Systems / A.M. Lister.
Edition 3rd Ed.
Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)
Main Title Introduction To Computer Systems / N.A.B. Gray.
Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.
Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.
Edition 2nd Ed.
Publisher Reading, Mass. : Addison-Wesley, 1985.
Author Stallings, William.
Main Title Operating Systems / William Stallings.
Edition 6th Ed.
Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-
Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.
Edition 2nd Ed.
Publisher Upper Saddle River, Nj : Prentice Hall, C1997.
Author Nutt, Gary J.
Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.
Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.
Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.
Edition 6th Ed.
Publisher Reading, Mass. : Addison Wesley Longman, C199

Notes

Objectives

In the last lecture, you learnt Paging and memory protection scheme. In this lecture, you will get to know about non-contiguous memory management scheme and the concept of Segmentation

Why Segmentation?

1. Pages are of a fixed size

In the paging scheme we have discussed, pages are of a fixed size, and the division of a process's address space into pages is of little interest to the programmer. The beginning of a new page comes logically just after the end of the previous page.

2. Segments are of variable sizes

An alternate approach, called segmentation, divides the process's address space into a number of segments - each of variable size. A logical address is conceived of as containing a segment number and offset within segment. Mapping is done through a segment table, which is like a page table except that each entry must now store both a physical mapping address and a segment length (i.e. a base register and a bounds register) since segment size varies from segment to segment.

3. No (or little) internal fragmentation, but you now have external fragmentation

Whereas paging suffers from the problem of internal fragmentation due to the fixed size pages, a segmented scheme can allocate each process exactly the memory it needs (or very close to it - segment sizes are often constrained to be multiples of some small unit such as 16 bytes.) However, the problem of external fragmentation now comes back, since the available spaces between allocated segments may not be of the right sizes to satisfy the needs of an incoming process. Since this is a more difficult problem to cope with, it may seem, at first glance, to make segmentation a less-desirable approach than paging.

4. Segments can correspond to logical program units

However, segmentation has one crucial advantage that pure paging does not. Conceptually, a program is composed of a number of logical units: procedures, data structures etc. In a paging scheme, there is no relationship between the page boundaries and the logical structure of a program. In a segmented scheme, each logical unit can be allocated its own segment.

1. Example with shared segments

Example: A Pascal program consists of three procedures plus a main program. It uses the standard Pascal IO library for read, write etc. At runtime, a stack is used for procedure activation records. This program might be allocated memory in seven segments:

- One segment for the main routine.
- Three segments, one for each procedure.
- One segment for Pascal library routines.
- One segment for global data.
- One segment for the runtime stack.

2. Several user programs can reference the same segment

Some of the segments of a program may consist of library code shareable with other users. In this case, several users could simultaneously access the same copy of the code. For example, in the above, the Pascal library could be allocated as a shared segment. In this case, each of the processes using the shared code would contain a pointer the same physical memory location.

Segment table user A	Segment table user B	Segment table user C
Ptr to private code	Ptr to private code	Ptr to private code
Ptr to private code	Ptr to shared code	Ptr to private code
Ptr to shared code	Ptr to private code	Ptr to private code
Ptr to private code		Ptr to shared code
Ptr to private code		Ptr to private code

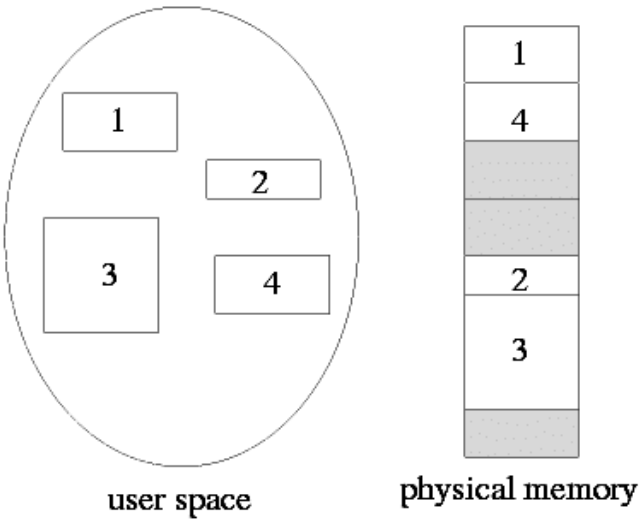
This would not be possible with pure paging, since there is no one-to-one correspondence between page table entries and logical program units.

3. Protection issues

Of course, the sharing of code raises protection issues. This is most easily handled by associating with each segment table entry an access control field - perhaps a single bit. If set, this bit might allow a process to read from the segment in question, but not to write to it. If clear, both read and write access might be allowed. Now, segments that correspond to pure code (user written or library) are mapped read only. Data is normally mapped read-write. Shared code is always mapped read only; shared data might be mapped read-write for one process and read only for others.

What is segmentation?

In paging, the user's view of memory and the actual physical memory are separated. They are not the same. The user's view is mapped onto the physical memory.



- A program is a collection of segments. A segment is a logical unit such as:

main program,

procedure,

function,

method,

object,

local variables, global variables,

common block,

stack,

symbol table, arrays

Segmentation Architecture

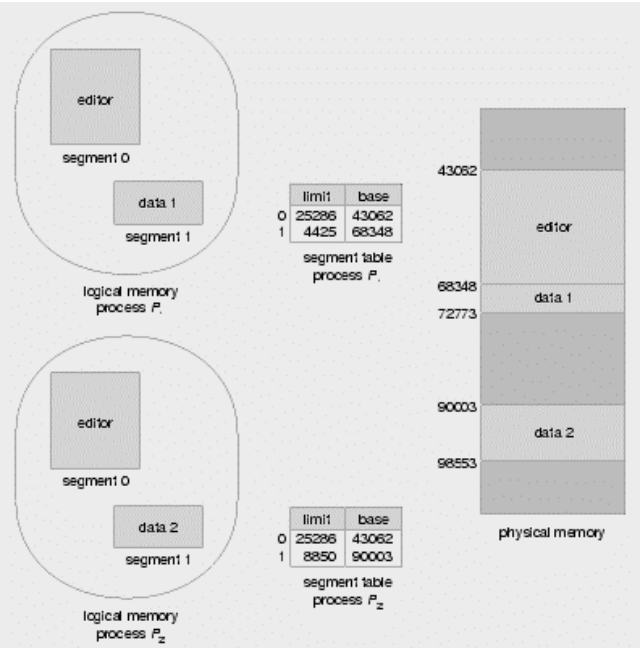
- Logical address consists of a two tuple:

  <segment-number, offset>
- Segment table – maps two-dimensional physical addresses; each table entry has:

  base – starting physical address of segments in memory

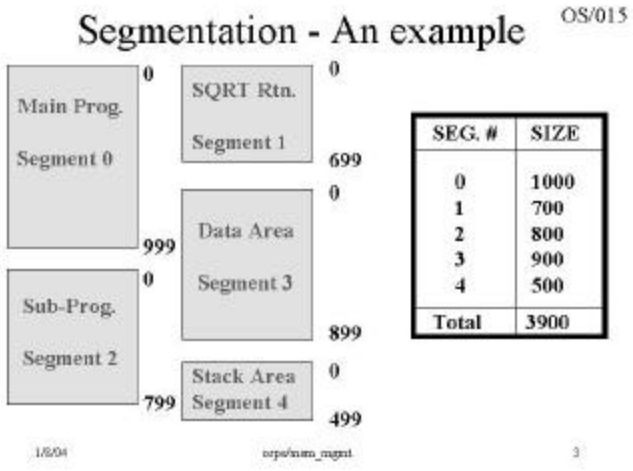
  limit – length of the segment
- Segment-table base register (STBR) points to the segment table's location in memory
- Segment-table length register (STLR) indicates number of segments used by a program; segment number s is legal if s < STLR
- Allocation: first fit/best fit and get external fragmentation
- Protection – easier to map; associated with each entry in segment table:
  - validation bit = 0 Þ illegal segment
  - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level.

- Since segments vary in length, memory allocation is a dynamic storage-allocation problem.
- A segmentation example is shown in the following diagram



What is user's view of memory?

The user of a system does not perceive memory as a linear array of bytes. The user prefers to view memory as a collection of variable sized segments with no necessary ordering among segments.



Let me explain this with an example:

Consider how you think of a program when you are writing it. You think of it as a main program with set of subroutines, procedures, functions, or variables. Each of these modules is referred to by a name. You are not concerned about where in memory these modules are placed. Each of these segments is of

variable length and is intrinsically defined by the purpose of the segment in the program.

Thus segmentation is a memory management scheme that supports this user view of memory. Thus a logical address space is a collection of segments with each segment having a name and length.

What is a 2-d address?

- In paging, a 1-d virtual address and a 2-d address would be exactly same in binary form as page size is an exact power of 2
- In segmentation, segment size is unpredictable. Hence we need to express the address in 2-d form explicitly.
- A system implementing segmentation needs to have a different address format and a different architecture to decode the address

So, what does the segment address consist of?

The address consists of:

- A segment number (S)
- An offset (D) within the segment

So what does the logical address space in segmentation consist of?

- Each segment is compiled with reference to 0 as the starting address for that segment
- Application programmer does not necessarily have to declare different segments in his program explicitly
- The compiler/linkage editor does it on its own as follows:
  - Recognizes different segments
- Numbers them
- Builds segment table
- Produces an executable image by assigning a 2-d address.

How is the segment table constructed?

Refer to the figure on page-4.

Putting all the segments, virtually one after the other, we get the following table:

Virtual Address Range	Virtual Segment Number
0 – 999	0
1000 – 1699	1
1700 – 2499	2
2500 – 3399	3
3400 – 3899	4

Figure-1

From the table, a virtual address 1100 will mean Segment # 1 and displacement 100 (S=1, D=100) and a virtual address 3002 will mean S = 3 and D = 502

How is address translation done in segmentation?

I will explain this with an example:

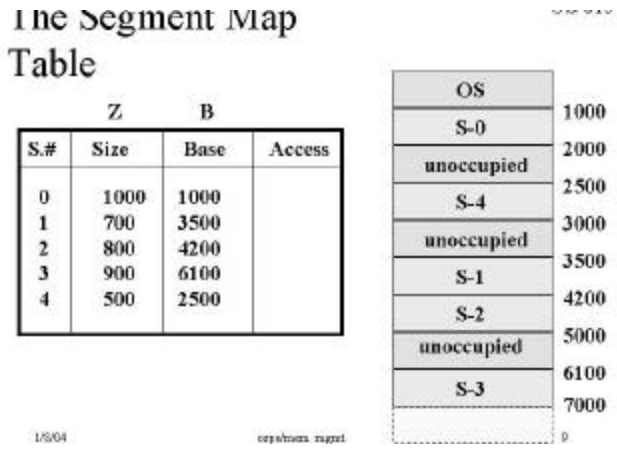
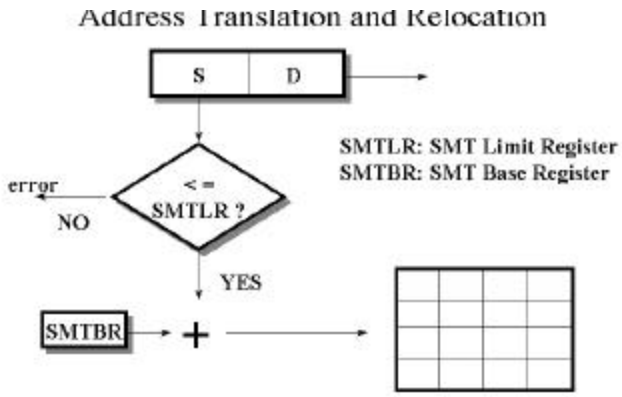


Figure-2

- Let the virtual address to be translated be 2520
- This corresponds to segment 3 (S) and offset = 20 (D) from figure-1
- Segment # is used as index into SMT.
- Info. at this entry is: *seg-size = 900, base = 6100* (figure-2)
- Displacement D is a virtual address within the segment and should not exceed seg-size. In this case, D = 20 is less than seg-size = 900
- OS then checks access rights.
- Effective address is then: Base + D = 6100 + 20 = 6120 which is also the physical address.
- Each process has one SMT

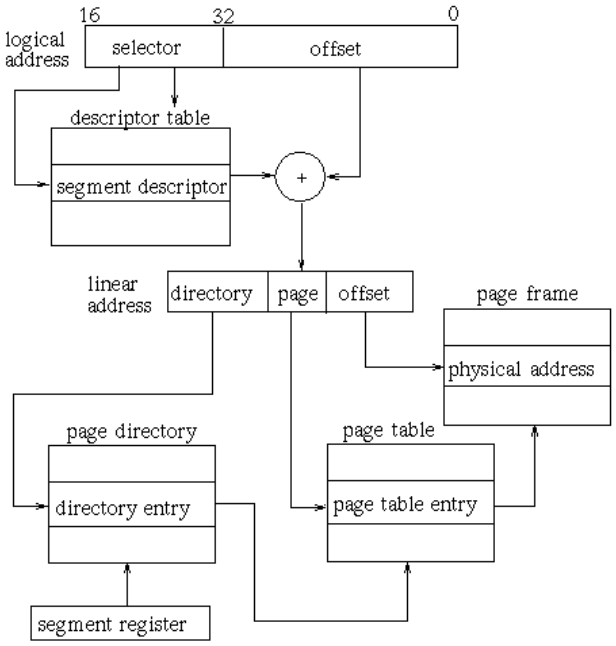
So what is this SMT?

Just as there was a page map table (PMT) associated with each process, there is a segment map table (SMT) per process. The segment table specifies physical base address and address range. Virtual addresses are offsets within a segment, added to base address and checked against address limit. Multiple segments handled by address mode or segment register. A typical SMT is shown in figure-2.



Segmentation with Paging

· The Intel Pentium uses segmentation with paging for memory management, with a two-level paging scheme.



What are the advantages and disadvantages of segmentation?

We will first look at the advantages:

1. The page faults are minimized as the entire segment is present in the memory. Only access violations need to be trapped.
2. No internal fragmentation as the segment size is customized for each process.

And now the disadvantages:

1. Allocation/deallocation sequences result in external fragmentation that needs a periodic pause for compaction to take place

Review Questions:

1. How does paging differ from segmentation?

_____

_____

_____

_____

_____

2. Describe the mechanism of translating a logical address to physical address in segmentation.

_____

_____

_____

_____

_____

_____

_____

_____

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems / Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

Objectives

In the last lecture, you learnt about Paging and Segmentation . In this lecture, you will get to know about Virtual Memory

Virtual memory is a memory management technique that allows the execution of processes that may not be completely in main memory and do not require contiguous memory allocation.

The address space of virtual memory can be larger than that physical memory.

Advantages:

- programs are no longer constrained by the amount of physical memory that is available
- increased degree of multiprogramming
- less overhead due to swapping

Why Do you Need Virtual Memory?

Storage allocation has always been an important consideration in computer programming due to the high cost of main memory and the relative abundance and lower cost of secondary storage. Program code and data required for execution of a process must reside in main memory to be executed, but main memory may not be large enough to accommodate the needs of an entire process. Early computer programmers divided programs into sections that were transferred into main memory for a period of processing time. As the program proceeded, new sections moved into main memory and replaced sections that were not needed at that time. In this early era of computing, the programmer was responsible for devising this overlay system.

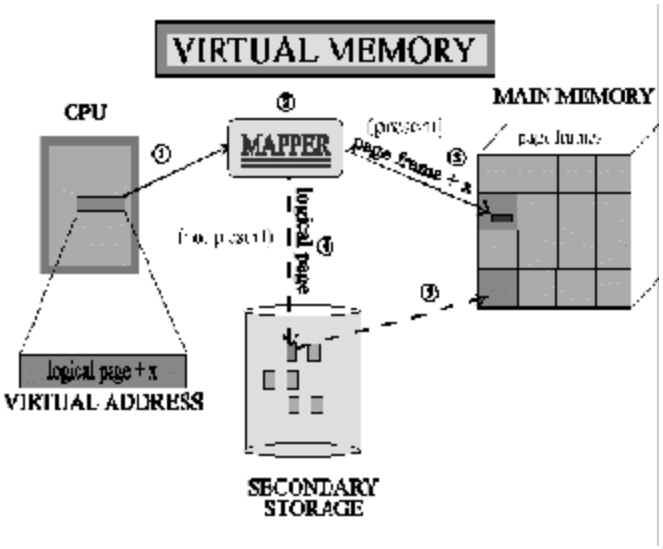
As higher level languages became popular for writing more complex programs and the programmer became less familiar with the machine, the efficiency of complex programs suffered from poor overlay systems. The problem of storage allocation became more complex.

Two theories for solving the problem of inefficient memory management emerged -static and dynamic allocation. Static allocation assumes that the availability of memory resources and the memory reference string of a program can be predicted. Dynamic allocation relies on memory usage increasing and decreasing with actual program needs, not on predicting memory needs.

Program objectives and machine advancements in the '60s made the predictions required for static allocation difficult, if not impossible. Therefore, the dynamic allocation solution was generally accepted, but opinions about implementation were still divided. One group believed the programmer should continue to be responsible for storage allocation, which would be accomplished by system calls to allocate or deallocate memory. The second group supported automatic storage allocation performed by the operating system, because of increasing complexity of storage allocation and emerging importance of multiprogramming. In 1961, two groups proposed a one-level memory store. One proposal called for a very large main memory to alleviate any need for storage allocation. This solution was not possible due to very high cost. The second proposal is known as **virtual memory**

Definition

Virtual memory is a technique that allows processes that may not be entirely in the memory to execute by means of automatic storage allocation upon request. The term virtual memory refers to the abstraction of separating **LOGICAL** memory-memory as seen by the process-from **PHYSICAL** memory-memory as seen by the processor. Because of this separation, the programmer needs to be aware of only the logical memory space while the operating system maintains two or more levels of physical memory space.

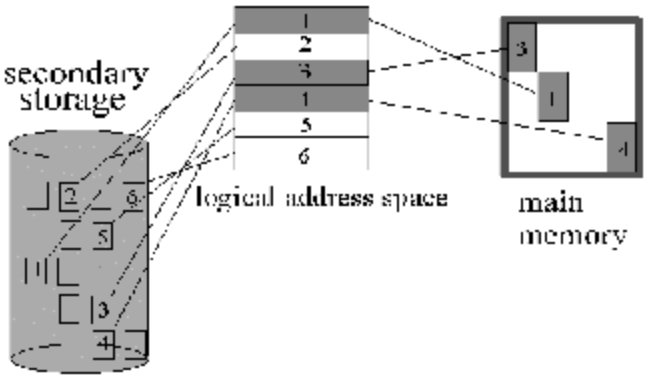


The virtual memory abstraction is implemented by using secondary storage to augment the processor's main memory. Data is transferred from secondary to main storage as and when necessary and the data replaced is written back to the secondary storage according to a predetermined replacement algorithm. If the data swapped is designated a fixed size, this swapping is called **paging**; if variable sizes are permitted and the data is split along logical lines such as subroutines or matrices, it is called **segmentation**. Some operating systems combine segmentation and paging

The diagram illustrates that a program generated address **(1)** or "logical address" consisting of a logical page number plus the location within that page (x) must be interpreted or "mapped" onto an actual (physical) main memory address by the operating system using an address translation function or mapper **(2)**. If the page is present in the main memory, the mapper substitutes the physical page frame number for the logical number **(3)**. If the mapper detects that the page requested is not present in main

memory, a fault occurs and the page must be read into a frame in main memory from secondary storage **(4 , 5)**.

What does the Mapper do?



The mapper is the part of the operating system that translates the logical page number generated by the program into the physical page frame number where the main memory holds the page. This translation is accomplished by using a directly indexed table called the **page table** which identifies the location of all the program's pages in the main store. If the page table reveals that the page is, in fact, not resident in the main memory, the mapper issues a **page fault** to the operating system so that execution is suspended on the process until the desired page can be read in from the secondary store and placed in main memory.

The mapper function must be very fast if it is not to substantially increase the running time of the program. With efficiency in mind, where is the page table kept and how is it accessed by the mapper? The answer involves associative memory.

Virtual memory can be implemented via:

- Demand paging
- Demand segmentation

What is demand paging?

Demand paging is similar to paging with swapping. Processes normally reside on the disk (secondary memory). 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**.

What is 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 one large contiguous address space, the use of the term **swap** is technically incorrect. A swapper manipulates entire processes whereas a **pager** is concerned with the individual pages of a process. It is correct to use the term pager in connection with demand paging.

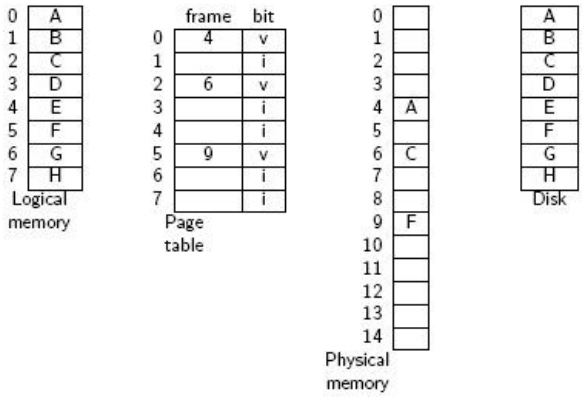
So how does demand paging work?

Whenever a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. So instead of swapping in the whole process, the pager brings only those necessary pages into memory. Here, I would like to add that demand paging requires **hardware support** to distinguish between those pages that are in memory and those that are on the disk.

Let me give you an example. Suppose you need white paper for doing your assignment. You could get it in two ways. In the first method, you will purchase about 500 sheets of paper. By the time you complete your assignment, you would have used only 100 sheets! So you are wasting 400 sheets of paper. In the second method, you could get 10 sheets of paper to start with and later on, as and when required, you could demand additional sheets of paper. This way, you will not be wasting money.

You talked about hardware support being required for demand paging. How does this support work?

An extra bit called the **valid-invalid** bit is attached to each entry in the page table. This bit indicates whether the page is in memory or not. If the bit is set to **invalid**, then it means that the page is not in memory. On the other hand, if the bit is set to **valid**, it means that the page is in memory. The following figure illustrates this:



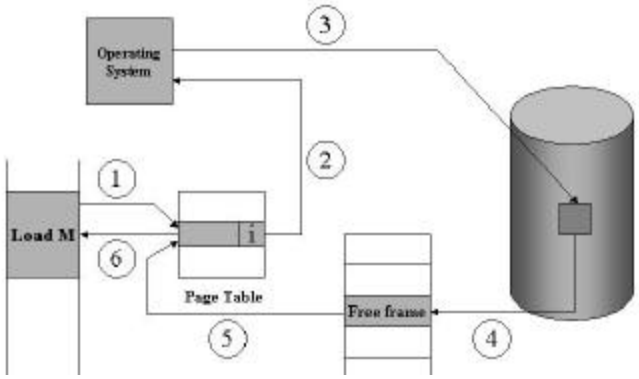
What happens if a process tries to use a page that was not brought into memory?

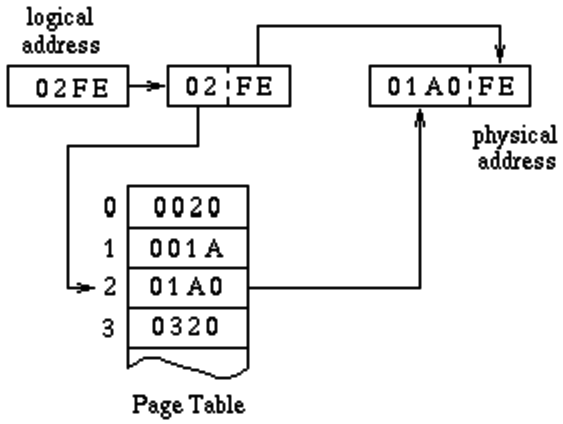
If you try to access a page that is marked invalid (not in memory), then page fault occurs.

How do you handle such page faults?

Upon page fault, the required page brought into memory by executing the following steps:

1. Check an internal table to determine whether the reference was valid or invalid memory access.
2. If invalid, terminate the process. If valid, page in the required page
3. Find a free frame (from the free frame list).
4. Schedule the disk to read the required page into the newly allocated frame
5. Modify the internal table to indicate that the page is in memory
6. Restart the instruction interrupted by page fault





What is the advantage of demand paging?

Demand paging avoids reading into memory pages that will not be used anyway. This decreases the swap time and also the physical memory needed.

We saw that whenever the referenced page is not in memory, it needs to be paged in. To start with, a certain number of frames in main memory are allocated to each process. Pages (through demand paging) are loaded into these frames.

What happens when a new page needs to be loaded into memory and there are no free frames available?

Well, the answer is simple. Replace one of the pages in memory with the new one. This process is called **page replacement**.

So Virtual memory basics

A. Virtual memory is an extension of paging and/or segmentation

The basic implementation of virtual memory is very much like paging or segmentation. In fact, from a hardware standpoint, virtual memory can be thought of as a slight modification to one of these techniques. For the sake of simplicity, we will discuss virtual memory as an extension of paging; but the same concepts would apply if virtual memory were implemented as an extension of segmentation.

B. Page table used to translate logical to physical addresses

Recall that in a paging scheme each process has a page table which serves to map logical addresses generated by the process to actual physical addresses. The address translation process can be described as follows:

1. Break the **logical address** down into a **page number** and an **offset**.
2. Use the **page number** as an index into the **page table** to find the corresponding **frame number**.
3. Using the **frame number** found there, generate a **physical address** by concatenating the frame number and the offset from the original address.

Example: suppose the page table for a process looks like this. Assume that the page size is 256 bytes, that logical addresses are 16 bits long, and that physical addresses are 24 bits long. (All numbers in the table are hexadecimal):

A logical address 02FE would be translated into the physical address 01A0FE.

C. Security in a paging system

In a paging system, one security provision that is needed is a check to be sure that the page number portion of a logical address corresponds to a page that has been allocated to the process. This can be handled either by comparing it against a maximum page number or by storing a validity indication in the page table. This can be done by providing an additional bit in the page table entry in addition to the frame number. In a paging system, an attempt to access an invalid page causes a hardware trap, which passes control to the operating system. The OS in turn aborts the process.

D. Situations that cause traps to the Operating System

In a virtual memory system, we no longer require that all of the pages belonging to a process be physically resident in memory at one time. Thus, there are two reasons why a logical address generated by a process might give rise to a hardware trap:

1. violations

The logical address is outside the range of valid logical addresses for the process. This will lead to aborting the process, as before. (We will call this condition a memory-management **violation**.)

2. page faults

The logical address is in the range of valid addresses, but the corresponding page is not currently present in memory, but rather is stored on disk. The operating system must bring it into memory before the process can continue to execute. (We will call this condition a **page fault**).

E. Need a paging device to store pages not in memory

In a paging system, a program is read into memory from disk all at once. Further, if swapping is used, then the entire process is swapped out or in as a unit. In a virtual memory system, processes are paged in/out in a piece-wise fashion. Thus, the operating system will need a paging device (typically a disk) where it can store those portions of a process which are not currently resident.

1. When a fault for a given page occurs, the operating system will read the page in from the paging device.

2. Further, if a certain page must be moved out of physical memory to make room for another being brought in, then the page being removed may need to be written out to the paging device first. (It need not be written out if it has not been altered since it was brought into memory from the paging device.)
3. When a page is on the paging device rather than in physical memory, the page table entry is used to store a pointer to the page's location on a the paging device.

F. Virtual memory has an impact on CPU scheduling

In a virtual memory system, the hardware can behave in basically the same way as for paging. However, the operating system no longer simply aborts the process when the process accesses an invalid page. Instead, it determines which of the above two reasons caused the trap. If it is the latter, then the operating system must initiate the process of bringing in the appropriate page. The process, of course, must be placed into a wait state until this is completed. So our set of possible process states must be extended from:

RUNNING

READY

WAITING for IO to complete

to:

RUNNING

READY

WAITING for IO to complete

WAITING for a page to be brought in

(Note, though, that a page wait is in reality just another form of IO wait, except that here the reason for the wait is not an explicit IO instruction in the process.)

G. Hardware support beyond that for paging along is required for virtual memory

Though the burden of recognizing and handling page faults falls on the operating system, certain provisions must be present in the hardware that are not needed with simple paging:

1. A page fault could occur while a single instruction is being carried out

The ability to restart an instruction that caused a fault in mid-stream. This can be tricky if the instruction accesses large blocks of memory - e.g. a block move that copies a character string en masse.

2. Page table entry should include a "dirty" bit

Though it is not strictly necessary, it is desirable to include a "written-in" bit in the page table entry, along with the valid bit noted above. This bit is set if any location in the page has been modified since it was brought into physical memory. This bit comes into play when the operating system finds it necessary to take the frame away from a page to make room for a new page being faulted in. If the old page has not been written in, then it need not be written back to disk, since it is the same as the copy on disk that was brought in originally.

3. May want a bit to indicate that a page has been accessed

Some implementations also require a per-page accessed bit that is set whenever any access (read or write) to the page occurs. This can be used to help decide which pages are no longer being actively used and so can be paged out to make room for new pages coming in. Not all memory management strategies require this, however.

Virtual memory design issues

A. Policy for bringing pages into memory

1. When does the OS decide to bring a page in?

We have already noted that, in general, only a portion of the pages belonging to a given process will actually be resident in physical memory at any given time. Under what circumstances is a given page brought in from the paging device?

2. Demand paging

The simplest policy is **demand paging**. Simply stated, under demand paging, a given page is only brought into memory when the process it belongs to attempts to access it. Thus, the number of page faults generated by a process will at least be equal to the number of pages it uses. (The number of faults will be higher if a page that has been used is removed from memory and then is used again.) In particular, when a process starts running a program there will be a period of time when the number of faults generated by the process is very high:

a. Page faults occur one-by-one as program begins running

To start running the program, the CPU PC register is set to the first address in the program. Immediately, a page fault occurs and the first page of the program is brought in. Once control leaves this page (due either to running off the end or to a subroutine call) another fault occurs etc. Further, any access to data will also generate a fault.

b. Startup and post-swapped time can be slow

An implication of pure demand paging is that the initial startup of a new program may take a significant amount of time, since each page needed will require a disk access to get it. Likewise, if a process is ever swapped out of memory due to a long IO wait then when it is brought back in it will be paged in one page at a time.

c. No pages are brought into memory unnecessarily

The chief advantage of demand paging is that no pages are ever brought into memory unnecessarily. For example, if a program contains code for handling a large number of different kinds of input data, only the code needed for the actual data presented to it will ever be brought in.

3. Anticipatory or Pre-paging

Some systems combine demand paging with some form of anticipatory paging or pre-paging. Here, the idea is to bring a page in before it is accessed because it is felt that there is good reason to expect that it will be accessed. This will reduce the number of page faults a process generates, and thus speed up its startup at the expense of possibly wasting physical memory space on unneeded pages. Anticipatory paging becomes increasingly attractive as physical memory costs go down.

a. Pages known to be initially required can all be loaded at once

When initially loading a program, there may be a certain minimum set of pages that have to be accessed for program initialization before branching based on the input data begins to occur. These can all be read in at once.

b. All pages swapped out can later be swapped back in at once

If a process is totally swapped out during a long IO wait, then swap the whole set of pages that were swapped out back in when it is resumed instead of paging it back in a little bit at a time.

c. Structure of page device may make it advantageous to read several pages at once

Another form of anticipatory paging is based on the clustering of the paging device. If several pages reside in the same cluster on the paging device, then it may be advantageous to read all of them in if any one of them is demanded, since the added transfer time is only a small fraction of the total time needed for a disk access. This is especially advantageous if the pages correspond to logically-adjacent memory locations.

B. Page replacement policies: What page do we remove from memory?

Over time, the number of pages physically resident in memory on a system under any significant load will eventually equal the number of available frames. At this point, before any new page can be faulted in a currently resident page must be moved out to make room for it. The question of how to select a page to be replaced is a very important one. In general, there are two kinds of page replacement policies.

1. Global policies

When process X needs to fault in a new page, the set of candidates for replacement includes all pages belonging to all processes on the system. Note that unless a page belonging to X already happens to be chosen, this will result in an increase in the total amount of physical memory allocated to X.

2. Local policies

When process X needs to fault in a new page, the set of candidates for replacement includes only those pages currently belonging to process X. Note that this means that the total amount of physical memory allocated to X will not change.

3. In general, a system will have to incorporate both kinds of policy:

a. At startup, we must use a global policy

When a process is just starting up, a global policy will have to be used since the new process has few pages available as replacement candidates.

b. Local paging may be used to keep a particular process from using too much memory

Eventually, however, a local policy may have to be imposed to keep a given process from consuming too much of the system's resources.

4. The working set of a process

Many of the policies to be discussed below can be applied either locally or globally. The notion of a process's **working set** can be used to help decide whether the process should be allowed to grow by taking pages from other processes or should be required to page against itself.

a. The working set is the set of pages that a process has accessed in the time interval [T - ΔT , T]

The working set for a process is defined in terms of some interval ΔT back from the current time T. Building on the principle of **locality of reference**, it is assumed that this is a good approximation to the set of pages that the process must have physically resident in order to run for an interval ΔT into the future without a page fault. (The interval ΔT is chosen to keep the percentage of memory accesses resulting in a fault to an acceptable level. A time corresponding to around 10,000 memory accesses being a good rule of thumb.)

b. During the life of a process, there are times when the working set changes slowly and other times when it changes rapidly

Studies of the memory access behavior of processes show that typically there are periods of time during which the working set of a given process changes very little. During these periods, if sufficient physical memory is allocated to the process then it can **page locally** against itself with an acceptably low rate of page faults. These periods are separated by bursts of paging activity when the process's working set is changing rapidly. These correspond to major stages in the program execution - e.g. the termination of one top level subroutine and the starting up of another. When this happens performance is improved if the **global paging** is used.

c. Maintaining a working set requires some system overhead

Of course, determining what the actual working set of a process is requires a certain amount of overhead - notably keeping track of what pages have been referenced during a past interval. (This is one of the places that a hardware referenced bit comes in.) One way to keep track of a process's working set involves using a timer that interrupts at the chosen interval ΔT:

- At the start of the interval, turn off all of the referenced bits in the page table for the currently running process.
- When the timer interrupts, include in the working set only those pages whose referenced bit is now on.

d. The working set concept can also be applied without going to all of the effort needed to determine the exact working set:

- If the **page fault rate** for a process **lies within a certain empirically determined range**, then assume that it has sufficient physical memory allocated to it to hold its (slowly evolving) working set and **page it locally**.
- If the page fault rate **increases** above the upper limit, assume its working set is expanding and **page it globally**, allowing its physical memory allocation to grow to keep pace with its presumably growing working set.
- If the page fault rate **drops too low**, then consider reducing its physical memory allocation by **not only paging it against itself** but also **allowing other processes to take page frames from it**. This corresponds to an assumption that the size of

its working set is less than the amount of physical memory currently allocated to it.

5. We defer detailed discussion of page replacement policies until we briefly note one further issue.

C. The degree of memory over allocation.

1. It is unusual in today's multiprogrammed systems for a single process to exceed the limits of the system's physical memory

We have seen that, under a virtual memory system, it is possible for the logical memory allocated to any one process to exceed the amount of physical memory available. In practice, however, this does not often occur, since virtual memory systems are generally multiprogrammed and thus are configured with sufficient physical memory to allow portions of many processes to be resident at once.

2. However, the sum of memory required by all processes on the system often exceeds the amount of physical memory

However, the sum total of the logical address spaces allocated to all the processes on the system will generally be far greater than the total amount of physical memory available. (If this were not so, then virtual memory would be of no benefit.) When memory is over allocated, each page faulted in will result in another page having to be moved out to make room for it. In general:

a. Too little over allocation (or none at all)

This means that the resource of physical memory is not really being used well. Pages that could be moved out to the paging device without harm are being kept in physical memory needlessly.

b. But too much over allocation can lead to a serious performance problem known as thrashing

Thrashing occurs when all of the pages that are memory resident are high-demand pages that will be referenced in the near future. Thus, when a page fault occurs, the page that is removed from memory will soon give rise to a new fault, which in turn removes a page that will soon give rise to a new fault ... In a system that is thrashing, a high percentage of the system's resources is devoted to paging, and overall CPU utilization and throughput drop dramatically.

c. The only way to prevent thrashing is to limit the number of processes that are actively competing for physical memory.

This can be done by using a form of intermediate scheduling, with certain processes being swapped out wholesale as in a non virtual memory system.

Ex: VMS has the concept of the **balance set** - which is the set of processes currently allowed to compete for physical memory. The size of the balance set is determined by the criterion: sum total of the working sets of all processes in the balance set <= available physical memory

Review Questions:

1. What is Virtual Memory? How it is implemented?

_____

_____

_____

_____

_____

_____

_____

_____

2. Explain Demand Paging? What are its advantages and disadvantages?

_____

_____

_____

_____

_____

_____

_____

_____

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems / Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

Notes

Objectives

In the earlier lecture, you learnt about demand paging, how it is different from swapping, its advantages and finally about page replacement concepts. In this lecture you will learn about page replacement algorithms.

We will start our discussion with explaining what a page replacement algorithm is.

In the last lecture, you saw that if there are no free memory frames to accommodate a newly demanded page, then one of the existing pages in memory needs to be swapped out and the new page loaded. The question that arises in our mind is "Which page should I swap out?"

To decide on this, we have page replacement policies or algorithms. The policy:

1. Deals with the selection of page in memory.
2. Determines which page currently in memory is to be replaced.

The objective of all policies is to remove the page least likely to be referenced in the near future.

What are the different page replacement algorithms?

There are a number of page replacement algorithms each with their own advantages and disadvantages. We discuss the following algorithms:

1. FIFO (First-In-First-Out)
2. Optimal (Reference, not practical)
3. LRU (Least Recently Used)
4. Clock

Before I discuss the various algorithms, let me highlight the goals of these page replacement algorithms.

The goals are to

- Minimize the number of page faults and
- Maintain a healthy page-table hit ratio which is defined as:

(Number of page references found in page table) **divided by (**number of page requests)

How can you determine the number of page faults occurring for a given algorithm?

We evaluate an algorithm by running it on a particular string of memory references called the **reference string** and computing the number of page faults.

We can generate the reference strings artificially or by tracing a given system and recording the address of each memory reference. This method may generate a large volume of data. In order to reduce the data, we note two things:

1. For a given page size, we need to consider only the page number and not the entire address.
2. If we have a reference to a page p, then any immediately following reference to page p will never cause a page fault. Page p will be in the memory after the first reference.

For example, if we trace a particular process, we might record the following address sequence:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102,0103, 0104, 0101, 0609, 0102,0105

If the page size is 100 bytes, then we have the following reference string that is generated: 1,4,1,6,1,6,1,6,1,6,1

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. As the number of page frames available increases, the page faults will decrease. To illustrate the page replacement algorithms, we shall use the following reference string for a memory with three frames.

7	0	1	2	0	3	0
4	2	3	0	3	2	1
2	0	1	7	0	1	

Let us start with discussing the FIFO algorithm:

FIFO algorithm:

Idea: Replace the page which is in memory for the longest time. Associated with each page is the time when that page was brought into memory. When a page must be replaced, the oldest one is picked.

Illustration: (Assume the three frames are initially empty)

Reference String

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Frame-1	7	7	7	2	H	2	2	4	4	4	0	H	H	0	0	H	H	7	7	7
Frame-2		0	0	0		3	3	3	2	2	2			1	1			1	0	0
Frame-3			1	1		1	0	0	0	3	3			3	2			2	2	1
	*	*	*	^		^	^	^	^	^	^			^	^			^	^	^

The first three references (7, 0, 1) cause page faults (shown by *) and get loaded into three empty frames. The next reference (2) causes a page fault and page replacement (shown by ^). The page replaced is page 7 (because it was brought in the first). H represents a 'HIT' (referenced page is already in the memory – no page fault).

Performance:

Hit ratio = 5/20 = 25%

Advantages: Easy to understand/implement.

Disadvantages:

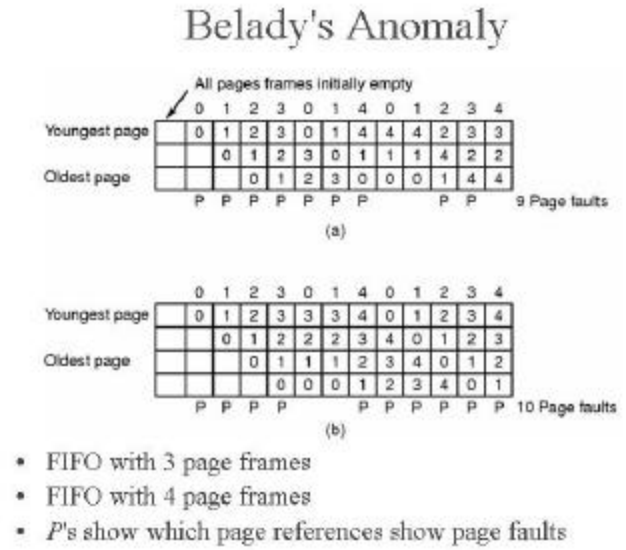
1. It has a poor performance because page usage is ignored. The page replaced may be an initialization module that was used a long time ago and is no longer needed. On the other

hand, it could contain a heavily used variable that was initialized early and is in constant use.

2. Suffers from Belady's Anomaly. Contrary to expectation, allocating more frames can result in worse page-fault behavior.

In the illustration above: (check this out by yourself)

- Hit ratio with 3 frames: 25%
- Hit ratio with 4 frames: 16.7%



Optimal Algorithm Idea: Replace the page that won't be needed for longest time.

Illustration: (Assume the three frames are initially empty)

Reference String

	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Frame-1	7	7	7	2	H	2	H	2	H	H	2	H	H	2	H	H	H	7	H	H
Frame-2		0	0	0		0		4			0			0				0		
Frame-3			1	1		3		3			3			1				1		
	*	*	*	^		^		^			^			^						

The first three references (7, 0, 1) cause page faults (shown by *) and get loaded into three empty frames. The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at reference 5, and page 1 at reference 14.

The reference to page 3 replaces page 1 as it would be the last of the three pages in memory to be referenced again.

Performance:

Hit ratio = 11/20 = 55%

Advantages:

1. Has the lowest page fault rate for a fixed number of frames
2. Never suffers from Belady's anomaly.

Disadvantages:

1. Difficult to implement. Requires future knowledge of reference string.

Used mainly for comparison studies.

LRU Algorithm

Idea: Approximation of optimum algorithm. Replace the page that has not been used for the longest period of time. Associates with each page, the time of that page's last use. When a page has to be replaced, LRU chooses that page that has not been used for the longest period of time.

Illustration: **(Assume the three frames are initially empty)**

Reference String

	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Frame-1	7	7	7	2	H	2	H	4	4	4	0	H	H	1	H	1	H	1	H	H
Frame-2		0	0	0		0		0	0	3	3			3		0		0		
Frame-3			1	1		3		3	2	2	2			2		2		7		
	*	*	*	^		^		^	^	^	^			^		^				

The first seven references yield same result as in Optimal Algorithm. When reference to page 4 occurs, , however, LRU algorithm sees that, of the three frames in memory, page 2 was used least recently. The most recently used page was page 0 and just before that, page 3 was used. Thus LRU replaces page 2 not knowing that page 2 is about to be used again. When page 2 is again referenced, LRU replaces page 3 since it is the least recently used.

Performance

Hit ratio = 8/20 = 40%

Advantages:

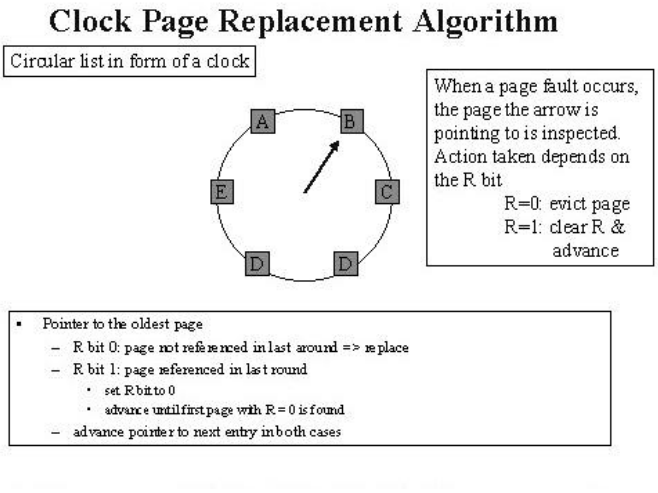
1. Reasonable approximation of Optimal Algorithm
2. Consecutive page references tend to be at pages that are close together.

Disadvantages:

1. Requires substantial hardware assistance.
2. How to implement? (How to determine an order for frames defined by the time of last use?)

Clock Page Replacement Algorithm

This algorithm is an approximation of the LRU algorithm. Hardware keeps a use bit or reference (R) bit per page frame and sets it on every reference to that page. When a page fault occurs, advance clock hand, and check the use bit (R). If it is 1, then clear the use bit and continue. If it is 0, then replace that page. The picture below summarizes the algorithm.



Review Questions

1. Discuss the various implementations of the LRU algorithm.

_____

_____

_____

_____

_____

_____

_____

_____

2. What is Belady's Anomaly?

_____

_____

_____

_____

_____

_____

_____

_____

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems / Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

Objectives

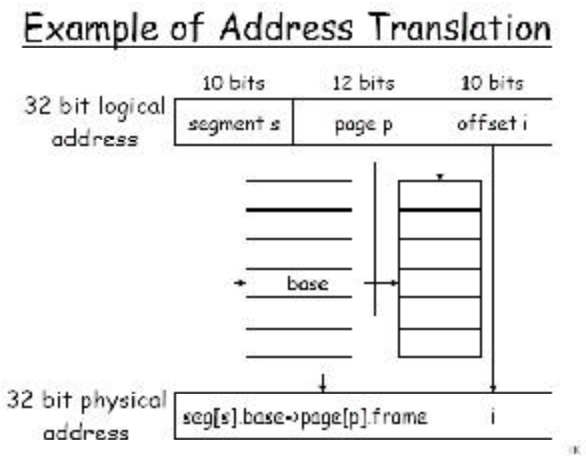
In the last lecture, you learnt about paging and segmentation which are two methods of implementing virtual memory. You also saw the advantages and disadvantages of these two methods. In this lecture, we will study about a method which is the combination of paging and segmentation.

Paged Segmentation

In this model, the logical memory is composed of segments. Each segment is composed of pages. The per process segment table is in memory pointed to by register. Entries map segment number to page table base. The page table is as described in the previous lecture.

How is the mapping from the logical address to physical address done in this combined approach?

The Logical address now consists of segment number, page number, and offset. The segment number is indexed into segment table to get **base** of page table. The page number is then used to index into page table to get the frame number. The frame number is then concatenated with the offset to get the physical address. The figure below gives an example of this mapping.



What are the main advantages of this combined approach?

The advantages stem out from the fact that it combines the individual advantages of paging and segmentation.

- Reduces external fragmentation (due to paging within a segment)
- Multiple address spaces available (for various segments)
- Distinguishes between access violations and page faults
- Swapping can occur incrementally
- Instructions can have smaller address fields
- What are the main disadvantages of this combined approach?
- Two memory accesses per translation. First the SMT and then PMT.
- More tables to manage (SMT and PMT)

How can you minimize the memory access?

This can be done by providing something called as TLB (Translation Look-aside Buffer). The TLB is like a cache. It keeps the most recent translations. Only when there is a miss in the TLB will the memory is accessed.

How does the TLB work?

When a reference to a page is made, the TLB is checked to see if there is an entry. If yes, then the frame number is retrieved from the TLB. If not, there is a TLB miss and the PMT is accessed. If the page is in the PMT, then it is loaded from there into the TLB and the physical address is computed. If the page is not there in the PMT also, then a page fault occurs and the required page is retrieved from the virtual memory and loaded into the PMT and then the TLB.

What is a page fault?

When a page is referenced and it is not in the PMT (and hence in memory), then it needs to be fetched from the virtual memory. This is called as page fault.

Review Questions

1. Explain Paged Segmentation

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems / Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.
Edition 3rd Ed.
Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)
Main Title Introduction To Computer Systems / N.A.B. Gray.
Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.
Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.
Edition 2nd Ed.
Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.
Main Title Operating Systems / William Stallings.
Edition 6th Ed.
Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-
Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.
Edition 2nd Ed.
Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.
Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.
Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.
Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.
Edition 6th Ed.
Publisher Reading, Mass. : Addison Wesley Longman, C1998.

Notes

LESSON-29
SELF- ASSESSMENT INTERACTIVE TOPIC

7.1 List types of resources we might consider in deadlock problems on computers.

Answer: CPU cycles, memory space, files, I / O devices, tape drives, printers.

7.2 Define deadlock.

Answer: A situation where every process is waiting for an event that can be triggered only by another process.

7.3 What are the four necessary conditions needed before deadlock can occur?

Answer:

a. At least one resource must be held in a nonsharable mode.

b. A process holding at least one resource is waiting for more resources held by other processes.

c. Resources cannot be preempted.

d. There must be a circular waiting.

7.4 Give examples of sharable resources.

Answer: Read-only files, shared programs and libraries.

7.5 Give examples of nonsharable resources.

Answer: Printer, magnetic tape drive, update-files, card readers.

7.6 List three overall strategies in handling deadlocks.

Answer:

a. Ensure system will never enter a deadlock state.

b. Allow deadlocks, but devise schemes to recover from them.

c. Pretend deadlocks don't happen.

7.7 Consider a traffic deadlock situation

a. Show that the four necessary conditions for deadlock indeed hold in this example.

b. State a simple rule that will avoid deadlocks in this system.

Answer:

a. Each section of the street is considered a resource.

- **Mutual-exclusion** —only one vehicle on a section of the street.
- **Hold-and-wait** — each vehicle is occupying a section of the street and is waiting to move to the next section.
- **No-preemption** — a section of a street that is occupied by a vehicle cannot be taken away from the vehicle unless the car moves to the next section.
- **Circular-wait** —each vehicle is waiting for the next vehicle in front of it to move.

b. Allow a vehicle to cross an intersection only if it is assured that the vehicle will not have to stop at the intersection.

7.8 List the data structures needed for the banker's algorithm.

Answer:

- available vector *Available*(m)
- demand matrix *Max*(n,m)
- allocation matrix *Allocation*(n,m)
- need matrix *Need*(n,m)

7.9 Summarize the banker's algorithm.

Answer:

a. If request for process *i* exceeds its need, error has occurred.

b. If request of process *i* exceeds available resources, process *i* must wait.

c. The system temporarily allocates the resources process *i* wants; if the state is unsafe, the allocation is postponed.

7.10 Summarize the Safety Algorithm.

Answer:

a. Initialize vector *Work* to *Available* and set vector *Finish* to false.

b. Find a process such that *Finish*(i) = false and *Need*(i) *leq Work*.

c. If found, add *Allocation*(i)to*Work*(i), *Finish*(i)to true,and go to step b.

d. If not found, continue here. If *Finish*(i) = true for all processes then state is safe, else it is unsafe.

7.11 How can we determine whether current state is "safe" in systems with only one instance of each resource type?

Answer: State is unsafe if any cycle exists.

7.12 What conditions must exist before a wait-for graph is useful in detecting deadlocks?

Answer: A cycle.

7.13 What does a cycle in a wait-for graph indicate?

Answer: A deadlock.

7.14 Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock-free.

Answer: Suppose the system is deadlocked. This implies that each process is holding one resource and is waiting for one more. Since there are three processes and four resources, one process must be able to obtain two resources. This process requires no more resources and therefore it will return its resources when done.

7.16 What is starvation?

Answer: System is not deadlocked, but at least one process is indefinitely postponed..

7.17 List three options for breaking an existing deadlock.

Answer

a. Violate mutual exclusion, risking data.

b. Abort a process.

c. Preempt resources of some process.

7.18 What three issues must be considered in the case of preemption?

Answer:

a. Select a victim to be preempted.

b. Determine how far back to rollback the victim.

c. Determine means for preventing that process from being "starved."

Objectives

Today I will be covering basic concepts related to files , basic file structures ,various file operations

Introduction

Whatever the objectives of the applications, it involves the generation and use of information. As you know the input of the application is by means of a file, and in virtually all applications, output is saved in a file for long-term storage.

You should be aware of the objectives such as accessing of files, saving the information and maintaining the integrity of the contents, virtually all computer systems provide file management services. Hence a file management system needs special services from the operating system.

1. Files:

The following are the commonly discussed with respect to files:

- **Field:** Basic element of data. Its length and data type characterizes it. They can be of fixed or variable length.
- **Record:** Collection of related fields. Depending on the design, records may be of fixed or variable length. Ex: In sequential file organization the records are of fixed length where as in Line sequential file organization the records are of variable length.
- **File:** Collection of similar records and is referenced by name. They have unique file names. Restrictions on access control usually apply at the file level. But in some systems, such controls are enforced at the record or even at the field level also.
- **Database:** Collection of related data. The essential aspects of a database are that the relationships that exists among elements of data. The database itself consists of one or more types of files.

Files are managed by the operating system. How they are structured, named, accessed, used, protected and implemented are the major issues in operating system design. As a whole, the part of the operating system deals with files is known as the file system. The linked lists and bitmaps are used to keep track of free storage and how many sectors there are in a logical block are important to the designers of the file system.

1.1 File Naming:

Files are an abstraction mechanism. The main characteristic feature of abstraction mechanism is the way the objects being managed are name. The exact rules for the file naming vary from system to system, but all current operating system allows strings of one to eight letters as legal file names. Many file systems support names as long as 255 characters with a distinguish in upper and lower case. Many operating systems support two-part file names, with the two parts separated by a period. The first part is called primary file name and the second part is called secondary or extension file name.

1.2 File Structure:

File can be structured in any of several ways. Three common possibilities are depicted (a) is an unstructured sequence of bytes (b) record sequence (c) tree structure.

a) Unstructured sequence of bytes: It provide the maximum flexibility. User programs can put anything they want in their files and name them any way that is convenient.

b) Record sequence: In this model, a file is a sequence of fixed length records each with some internal structure. Central idea of a file being a sequence of records is the idea that the read operation returns and the write operation overwrites or appends one record.

c) Tree structure: In this organization, a file consists of a tree of records, not necessarily all the same length, each containing a key field in a fixed position in the record. The tree is sorted on the key field, to allow rapid searching for a particular key.

Let us discuss File System components

- Device Drivers:
  - Communicates directly with peripherals devices (disks, tapes, etc)
  - Responsible for starting physical I/O operations on the device
  - Processes the completion of an I/O request
  - Schedule access to the device in order to optimize performance
- Basic File System:
  - Uses the specific device driver
  - Deals with blocks of data that are exchanged with the physical Device
  - Concerned with the placement of blocks on the disk
  - Concerned with buffering blocks in main memory
- Logical File System:
  - Responsible for providing the previously discussed interface to the user including:
  - File access
  - Directory operations
  - Security and protection.

1.3 File Types:

Many operating systems support several types of files. Unix and Windows, have regular files and directories. Regular files are the ones that contain user information generally in ASCII form. Directories are system files for maintaining the structure of the file system. Character special files are related to input/output and

used to model serial I/O devices such as terminals, printers and networks. Block special files are used to model disks.

1.4 File Access:
Early operating systems provided only one kind of file access: sequential access. In these system, a process could read all the bytes or records in a file in order, starting at the beginning, but could not skip around and read them our of order. Sequential files were convenient when the storage medium was magnetic tape, rather than disk. Files whose bytes or records can be read in any order are called random access files. Two methods are used form specifying where to start reading. In the first one, every read operation gives the position in the file to start reading at. In the second one, a special operation, seek, is provided to set the current position. This allows the system to use different storage techniques for the two classes. Where as in modern operating systems all the files are automatically random access.

1.5 File Attributes:
Every file has a name and its data. In addition all operating systems associate other information with each file such as the date and time the file was created and the file's size. The list of attributes varies considerably from system to system. Attributes such as protection, password, creator and owner tell who may access it and who may not. The flags are bits or short fields that control or enable some specific property. The record length, key, position and key length fields are only present in files whose records can be looked up using a key. The various times keep track of when the file was created, most recently accessed and most recently modified. These are useful for a variety of purpose. The current size tells how big the file is at present.

1.6 File operations:
Files exist to store information and allow it to be retrieved later. Different systems provide different operations to allow storage and retrieval. The few of them of the most common system calls relating to files are:

1.6.1 Create:
The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.

1.6.2 Delete:
When the file is no longer needed, it has to be deleted to free up disk space.

1.6.3 Open:
Before using a file, a process must open, the purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.

1.6.4 Close:
When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space.

1.6.5 Read:
Data re read from file. Usually, the bytes a come from the current position. The caller must specify how much data are needed and must also provide a buffer to put them in.

1.6.6 Write:
Data are written to the file, again, usually at the current position. If the current position is end of the file then the file size gets increased.

7.7.7 Append:

This call is a restricted from of write. It can only add data to the end of the file.

1.6.8 Seek:

For random access files, a method is needed to specify from where to take the data.

1.6.9 Get attributes:
Processes often need to read file attributes to do their work.

1.6.10 Set attributes:
Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible. The protection mode information is an obvious example.

11.11.11Rename:
It frequently happens that a user needs to change the name of an existing file.

Review Exercise:

2. What is the importance of a filename having two parts?

_____
_____
_____
_____
_____

3. What are the rules that govern for naming a file?

_____
_____
_____
_____
_____

4. Discuss to make the file system more useful.

_____
_____
_____
_____
_____

5. What are the components that constitute the file system?

_____
_____
_____
_____
_____

6. What are the various file attributes that are applicable for different types of files. Explain?

Reference Books:

Author Dahmke, Mark.

Main Title Microcomputer Operating Systems / Mark Dahmke.

Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-

Main Title An Introduction To Operating Systems / Harvey M. Deitel.

Edition Rev. 1st Ed.

Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-

Main Title Fundamentals Of Operating Systems / A.M. Lister.

Edition 3rd Ed.

Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)

Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

Notes

Objectives

In this lecture I will be covering all the points given below.

- Concept of directories
- Their organization & structure
- And various operations on directories
- Directory Implementations

Directories

You should know that the operating system keeps the track of files, file systems normally have directories or folders, which, in many systems are themselves, are files.

Directory Organization

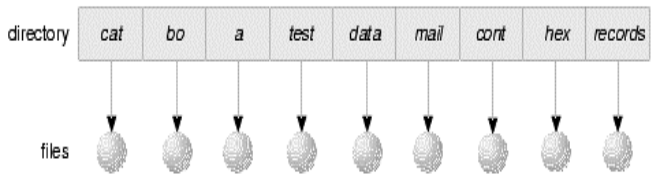
- **Goals:**
  - Efficiency: quickly locating the file.
  - Convenience: naming files in a convenient way to users.
  - Grouping: allowing users to group files based on users classifications.
- **Organization of the file system**
  - Virtual disks (called volumes, or minidisks, or partitions). May span a physical disk, part of a physical disk, or several disks.
  - Virtual disk directory.
- **Directory Structure**

Single-Level Directory System:

The simplest form of directory system is having one directory containing all the files, which is called the root directory. It is commonly used on personal computers, which is used by only one user. But the problem arises when multiple users create their directories, where the name of the directory may coincide resulting to problem.

Important aspects of Single Level Directory

- All files are contained in the same directory.
- Easy to support and understand.
- **Have significant limitations in terms of:**
  - Large number of files (naming).
  - Ability to support different users / topics (grouping).

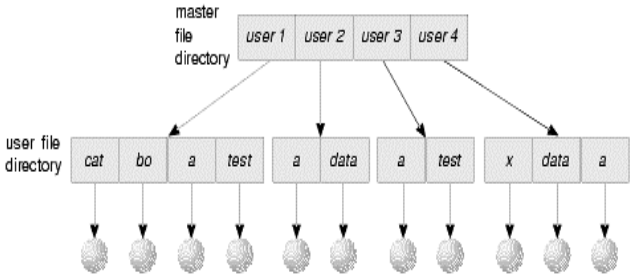


Two-Level Directory Systems:

To avoid conflicts caused by different users choosing the same file name for their own files, the next step up is giving each user a private directory. In that way, names chosen by one user do not interfere with names chosen by a different user and there is no problem caused by the same name occurring in two or more directories. This design used on multi-user computer or on a simple network of personal computers that shared a common file server over a local area network. This level structure uses login procedure, which restricts the unauthorized user access.

These are the Important aspects of Two-Level Directory

- One master file directory.
- Each user has their own user file directory.
- Each entry in the master file directory points to a user file directory.
- Issues:
  - Sharing - accessing other users files.
  - System files.
  - Grouping problem.

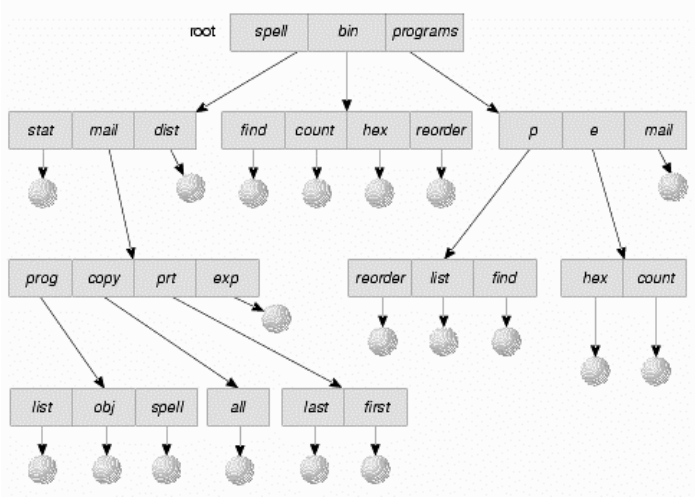


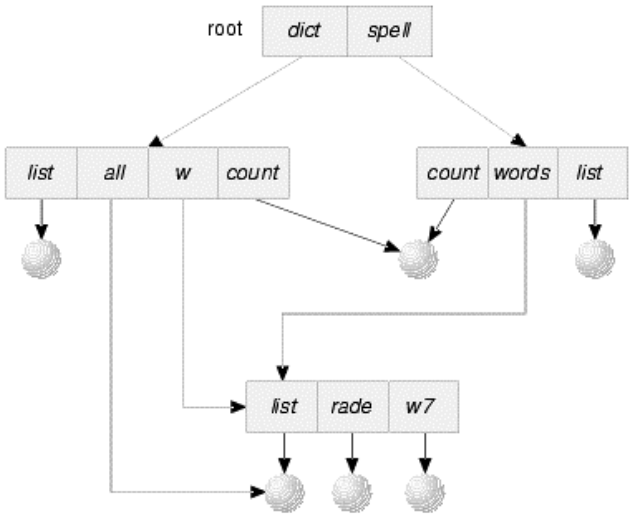
Hierarchical Directory Systems:

The two-level hierarchy eliminates name conflicts among users but is not satisfactory for users with a large number of files. In this approach, each user can have as many directories as are needed so that files can be grouped together in natural ways. The ability for users to create an arbitrary number of subdirectories provides a powerful structuring tool for users to organize their work.

Tree-Structured Directory

- The directory structure is a tree with arbitrary height.
- Users may create their own subdirectories.
- Issues
  - Efficient Searching, grouping.
  - Current directory notion / change directory (absolute/ relative naming).
  - Directory semantics (e.g. deletion).





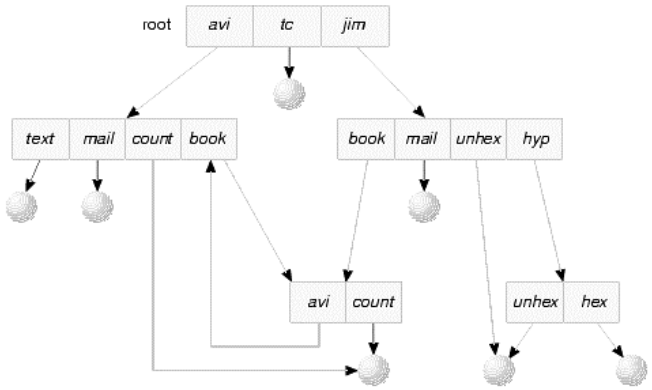
Acyclic-Graph Directory

- Allow sharing of directories and files several times on the tree structure.
- **Issues**
  - Only one actual copy of the file or directory is stored.
  - Can be accessed through one or more paths.
  - May have different names.
- There are several ways to implement shared files and directories.
- **Example - Unix:**
- Symbolic links:
  - A different type of a directory entry (other than a file or a directory).
  - Specifies the name of the file that this link is pointing to.
  - Can be relative or absolute.
  - What happens when one deletes the original file?
  - Duplicate directory entries (also called hard links):
  - The original and copy entries are the same.
  - What happens when one deletes the original or copy entries?

General-Graph Directory

- **A problem with acyclic graphs - how to ensure that there are no cycles**.
  - Can happen only when linking a directory.
  - Every time a link is added to a directory - use a cycle detection algorithm...
- General graph directories allow cycles.
- Issues:

How to avoid traversing a component in a cycle while searching.



How to release orphan cycles (garbage collection).

Path Names:
When the file system is organized as a directory tree, some way is needed for specifying file names. Two different methods are commonly used. In the first method, each file is given an absolute path name consisting of the path from the root directory to the file. The other kind of name is the relative path name in conjunction with the concept of the working directory.

Directory Operations:
The allowed system calls for managing directories exhibit more variation from system to system than system calls for files.

Create: A directory is created. It is empty except for dot and dotdot which are put there automatically be the system.

Delete:
Only an empty except directory can be deleted.

Opendir:
To list all the files in a directory, a listing program opens the directory to read out the names of all the files it contains. Before a directory can be read, it must be opened, analogous to opening and reading a file.

Closedir:
When a directory has been read, it should be closed to free up internal table space.

Readdir:
This call returns the next entry in an open directory. It always returns one entry in a standard format, no matter which of the possible directory structures is being used.

Rename:
They can be renamed the same way file can be.

Link:
Linking a technique that allows a file to appear in more than one directory. This system call specifies an existing file and a path name, and creates a link from the existing file to the name specified by the path.

Unlink:
A directory entry is removed. If the file being unlinked is only present in one directory, it is removed from the file system. If it is present in multiple directories, only the path name specified is removed.

Directory Implementations

We can implement the directory in several ways.

Linear List

The simplest approach to implementing a directory is simply maintaining a list (or array) of file names (and other info).

This approach has some problems like: searching for a file means we must do a linear search, and inserting and deleting from a fixed list is not simple.

We can also maintain a sorted list, but that would mean maintaining the sorted structure.

Hash Table

We can also implement the directory as a hash table. This could be in addition to the linear list. A hash table allows us to quickly find any individual file.

Hash tables aren't the perfect solution either. One problem is hash clashes. These must be resolved, and ways to resolve them aren't exactly better than maintaining a linear list.

Let me brief you the description of Directory Operations:

- Create a file.
- Delete a file.
- Search for a file.
- List a directory.
- Rename a file.
- Traverse the file system.

Review Exercise:

1. What are the similarities and dissimilarities between files and directories?

_____
_____
_____
_____
_____

2. What is the advantage of directories over files?

_____
_____
_____
_____
_____

3. Describe the directory structure?

_____
_____
_____
_____
_____

4. Explain various operations on a directory?

_____
_____
_____
_____
_____

Reference Books:

Author Dahmke, Mark.
Main Title Microcomputer Operating Systems / Mark Dahmke.
Publisher Peterborough, N.H : Mcgraw-Hill/Byte Books, C1982.

Author Deitel, Harvey M., 1945-
Main Title An Introduction To Operating Systems / Harvey M. Deitel.
Edition Rev. 1st Ed.
Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-
Main Title Fundamentals Of Operating Systems / A.M. Lister.
Edition 3rd Ed.
Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)
Main Title Introduction To Computer Systems / N.A.B. Gray.

Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.

Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.

Edition 2nd Ed.

Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.

Main Title Operating Systems / William Stallings.

Edition 6th Ed.

Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-

Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.

Edition 2nd Ed.

Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.

Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.

Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.

Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.

Edition 6th Ed.

Publisher Reading, Mass. : Addison Wesley Longman, C1998.

Notes

10.1 What is a file?

Answer: A named collection of related data defined by the creator, recorded on secondary storage.

10.2 What does OPEN do?

Answer: Creates memory buffers, creates data control blocks, and creates other data structures needed for the I/O. If file is new, it also allocates space, and enters name in directory.

10.3 What does CLOSE do?

Answer: Outputs last buffer of information. Deletes buffers, data control blocks, and other data structures.

10.4 List advantages of operating system "knowing" and supporting many file types.

Answer: Can prevent user from making ridiculous mistakes. Can make system convenient to use by automatically doing various jobs after one command.

10.5 List the disadvantages of operating system "knowing" and supporting many file types.

Answer: Size of operating system becomes large. Every file type allowed must be de-fined, thus hinders in creating new file types.

10.6 What is a sequential file?

Answer: A file that is read one record or block or parameter at a time in order, based on a tape model of a file.

10.7 What is direct access?

Answer: A file in which any record or block can be read next. Usually the blocks are fixed length.

10.8 How does user specify block to be fetched in direct access?

Answer: By specifying the relative block number, relative to first block in file, which is block 0.

10.9 Can a direct access file be read sequentially? Explain.

Answer: Yes. Keep a counter, *c*p, initially set to 0. After reading record *c*p, increment *c*p.

10.10 How can an index file be used to speed up the access in direct-access files?

Answer: Have an index in memory; the index gives the key and the disk location of its corresponding record. Scan the index to find the record you want, and then access it directly.

10.11 Explain what ISAM is.

Answer: Indexed sequential access method. The file is stored in sorted order. ISAM has a master index file, indicating in what part of another index file the key you want is; the secondary index points to the file records. In both cases, a binary search is used to locate a record.

10.12 List two types of system directories

Answer:

a. Device directory, describing physical properties of files.

b. File directory, giving logical properties of the files.

10.13 List operations to be performed on directories.

Answer: Search for a file, create a file, delete a file, list a directory, rename a file, traverse the file system.

10.14 List disadvantages of using a single directory.

Answer: Users have no privacy. Users must be careful in choosing file names, to avoid names used by others. Users may destroy each other's work.

10.15 What is the MFD? UFD? How are they related?

Answer: MFD is master-file directory, which points to the UFDs. UFD is user-file directory, which points to each of user's files.

10.16 What advantages are there to this two-level directory?

Answer: Users are isolated from each other. Users have more freedom in choosing file names.

10.17 What disadvantages are there to this two-level directory?

Answer: Without other provisions, two users who want to cooperate with each other are hampered in reaching each other's files, and system files are inaccessible.

10.18 How do we overcome the disadvantages of the two-level directory?

Answer: Provide links from one user directory to another, creating path names; system files become available by letting the command interpreter search your directory first, and then the system directory if file needed is not in first directory.

10.19 What is a file path name?

Answer: A list of the directories, subdirectories, and files we must traverse to reach a file from the root directory.

10.20 If we use the two-level directory, how do we access common files and programs, like FORTRAN compiler? Show two or more ways.

Answer:

a. Keep copy of each common file in each user account.

b. Keep common files in a special account of system files, and translate the commands to path names to those files.

c. Permit path names from one directory to another.

10.21 Why would we want a subdirectory in our account?

Answer: To group files into collections of similar nature, and to protect certain groups of files from other users.

10.22 List steps you need to follow to delete a subdirectory in your account.

Answer: Delete all files in subdirectory. Change protection code to allow deletion, and then delete the subdirectory. This procedure must be followed, starting with the deepest subdirectory.

10.23 What is an acyclic graph?

Answer: A tree that has been corrupted by links to other branches, but does not have any cyclic paths in it.

10.24 List ways to share files between directories in operating systems.

Answer:

a. Copy file from one account into another.

b. Link directory entry of "copied" file to directory entry of original file.

c. Copy directory entry of file into account file is "copied" into.

10.25 What problems might arise on deletion if a file is shared?

Answer: Copier of file might delete the original shared file, depriving rest of users. They have a pointer to a deleted directory entry pointing to the original file or one overwritten by other users of the system, or a new entry pointing to a new file created by the original user.

10.26 How can we solve this problem?

Answer: Keep a count of the number of links to a file in original directory. As each person deletes a file, the count decreases by 1.

10.27 What is a general graph?

Answer: A tree structure where links can go from one branch to a node earlier in the same branch or other branch, allowing cycles.

10.28 What problems arise if the directory structure is a general graph?

Answer: Searching for a particular file may result in searching the same directory many times. Deletion of the file may result in the reference count to be nonzero even when no directories point to that file.

10.29 What is garbage collection?

Answer: Determining what file space is available, and making it available for users.

(Note: garbage collection is also done in BASIC, to reclaim space used by deleted strings.)

10.30 How can we protect files on a single-user system?

Answer:

a. Hide the disks.

b. Use file names that can't be read.

c. Backup disks.

d. On floppies, place a write-disable-tab on.

10.31 What might damage files?

Answer: Hardware errors, power surges, power failures, disk-head crashes (read/write head scraping magnetic material off disk), dirt, temperature, humidity, software bugs, fingerprints on magnetic material, bent disk or cover, vandalism by other users, storing diskettes near strong magnets which are found in CRTs, radio speakers, and so on.

10.32 List kinds of access we might want to limit on a multiuser system.

Answer: Reading files in given account; creating, writing, or modifying files in given account; executing files in given account; deleting files in given account.

10.33 List four ways systems might provide for users to protect their files against other users.

Answer:

a. Allowing user to use unprintable characters in naming files so other users can't determine the complete name.

b. Assigning password(s) to each file that must be given before access is allowed.

c. Assigning an access list, listing everyone who is allowed to use each file.

d. Assigning protection codes to each file, classifying users as system, owner, group, and world (everyone else)..

Self-assessment interactive topic

11.1 List three ways of allocating storage, and give advantages of each.

Answer:

a. Contiguous allocation. Fastest, if no changes are to be made. Also easiest for random-access files.

b. Linked allocation. No external fragmentation. File can grow without complications.

c. Indexed allocation. Supports direct access without external fragmentation.

11.2 What is contiguous allocation?

Answer: Allocation of a group of consecutive sectors for a single file.

11.3 What main difficulty occurs with contiguous allocation?

Answer: Finding space for a new file.

11.4 What is a "hole" in contiguous allocation method?

Answer: An unallocated segment of blocks.

11.5 Explain first-fit, best-fit, and worst-fit methods of allocating space for contiguous files.

Answer:

- First-fit: Scan available blocks of disk for successive free sectors; use the first area found that has sufficient space; do not scan beyond that point.
- Best-fit: Search for smallest area large enough to place the file.
- Worst-fit: Search for largest area in which to place the file.

11.6 What is external fragmentation in a system with contiguous files?

Answer: The disk has files scattered all over; fragmentation occurs when there is enough empty space collectively for the next file, but there is no single gap large enough for the entire file to fit in.

11.7 How can we overcome fragmentation?

Answer: We can use an allocation technique that does not result in fragmentation; or we can move the files around on disk, putting them closer together, to leave us larger blocks of available sectors.

11.8 What is preallocation? Why do it?

Answer: Allocating space for a file before creating the file to allow for expansion. This reserves space for a particular file so that other files can't grab it. The new file may initially use only a small portion of this space.

11.9 What is linked allocation, as detailed in text?

Answer: Directory contains pointers to first and last blocks of file. Each block of file (except last) has pointer to the next block.

11.10 Can linked allocation have external fragmentation? Internal fragmentation?

Answer: External - no. Internal - Yes.

11.11 Can linked allocation be used for direct-access files?

Answer: Not in the form suggested in the book. RSTS on the PDP-11 stores the sector numbers in the directory, with each group of seven addresses linked to the next group of seven. Direct access using this modified linked allocation is possible. (This approach is really a hybrid of linked and indexed allocations.)

11.12 What is indexed allocation?

Answer: Each file has its own block of pointers to the sectors of the file.

11.13 Rank the allocation methods on speed.

Answer: Contiguous is fastest. Linked is slower, because the disk head may have to move between accesses of file. Indexed is slowest, unless the entire index can be kept in memory at all times. If not, then extra time must be used to access next block of file indexes.

11.14 List four ways a system could use to determine which sectors are free. Give advantages of each way.

Answer:

a. Free-space list. Each section indicates a sector that is available. Not encumbered by a used-sector list.

b. Bit vector is a compact version. Has no links that can be broken.

c. Link all free sectors together in an available list. Takes no usable space. But links could break.

d. List giving start of each block of free sectors, and a count of number of sectors in this block. This is fast for use in contiguous storage search.

11.15 What data structures can be used for directory information?

Answer:

a. Linear list

b. Linked list

c. Sorted list

d. Linked binary tree

e. Hash table

11.16 What problems might arise with above data structures?

Answer:

a. Linear list is slow to access particular file. Also must decide how to take care of deletions (mark, copy last entry to it, ...).

b. Linked list requires storage overhead for pointers; also, if link goes bad, rest of files are lost.

c. Sorted list requires list always to be sorted, which means extra work on creating and deleting files.

d. Binary tree suffers like linked list.

e. Hash tables are set up for a maximum number of files; also there is a problem with collisions.

11.17 Give advantages of each directory structure above.

Answer:

Linear list Simple to program search.

Linked list Easier to process deletes.

Sorted list Fast access.

Linked binary tree Faster access.

Hash table Fastest access...

Notes

ANSI
American National Standardisation Institute

ASCII
American Standard Code for Information Interchange - a table converting nummeric values into human readable characters.

API
Application Programming Interface - the set of routines / functions made available to a program developer.

ATA
AT Attachment - also known as IDE.

ATAPI
ATA Packet Interface - minor extension to IDE to control additional device types.

BASIC
Beginners All-purpose Symbolic Instruction Code - a high-level interpreted programming language which is very easy to learn.

Binary
A base-2 system written as either 1b or 1_2, unlike our normal base-10. Each place is multiplied with 2 as you move left: 00000001b = $1*2^0 = 1$, 00000010b = $1*2^1+0*2^0$, 00000011b = $1*2^1+1*2^0$, 00000100b = $1*2^2+0*2^1+0*2^0 = 4$, etc.

Bit
Binary Digit - the smallets possible piece of information: set or cleared, 1 or 0, true or false.

Byte
A group of 8 bits making up a base-2 number.

CD-ROM
Compact Disc Read Only Memory - Optical storage media with a capacity of 74 minutes of digital music or 650-740 MB of read only storage. Serveral different formats exists, known by their book-color (red book, yellow book etc.).

CDi
CD Interactive - Philips extended CD-ROM architecture for video storage.

Char
Character - usually a 1 byte data size representing one character to the user. In the case of Unicode of DBCS a char may take up two bytes of memory.

CP/M
Control Program for Microcomputers - A simple single tasking single user OS used on home PC in the late 1970s. Claimed to be the blueprint for QDOS.

CPU
Central Processing Unit - the chip that executes a stream of instructions, like add 2 and 4, read from memory, etc. It control everything in your computer. Often supplied by Intel and known as 486, Pentium or Pentium II. Other CPUs exist, like Digitals Alpha, Suns UltraSPARC, Mips, Hewlett-Packard.

DASD
Direct Access Storage Device

DBCS
Dual Byte Character Set - an expansion of ASCII to support foregin languages better. Special char values are reserved and results in the char size increasing to two bytes (eg. 4Ah is normal one byte where F8h,B3h is a double byte character). A very efficient method to store all possible characters in limited memory space.

Disk
A circular storage media, on which data can be recorded, usually in sectors. Often magnetically or optically coded.

DLL
Dynamic Link Libary - a set of routines stored in a file. Addresses to the routines are determined by the OS on load or at run-time.

DOS
Disk Operating System - a simple single user single tasking OS bought by Bill Gates as QDOS and used on the first IBM PC's. A clone, DR-DOS is available.

DOS 8.3
Refers to the naming convention on DOSs FATFS volumes, where a filename had to be 1-8 chars long and have an extension of 0-3 chars, seperated by a period.

EMS
Expanded Memory Specification - a system to access additional memory in DOS. It is specified in the so-called LIM (Lotus-Intel-Microsoft) EMS specification. It uses 16K pages which can be swapped in and out of real-mode addressing space.

GID
Group ID - a number or a string that represents the group to which a user belongs uniquely within the OS.

GPF
General Protection Fault - a process violated its assigned resources and tried to access a resource which it was not granted (often unavailable memory due to pointer errors).

GUI
Graphical User Interface - a representation using squares (windows) to represent a programs output. Uses buttons and icons to implement a more user-friendly interface with the computer. Today, most OS are delivered with a GUI. The GUI was developed at RankXerox PARC, introduced to the public with Apple Macintosh and made common by MS Windows 3.0.

Handle
Usually a number used by the operating system to identify an object, like an open file or a window.

Hex
Hexadecimal notation - a base-16 number system, where a=10, b=11, c=12, d=13, e=14, f=15. Written as 00h, $00 or 00_{16}. Eg. 10h = 16, 2Fh=47 (2*16+15), 123h=291 ($1*16^2+2*16+3$).

HMA
High Memory Area - a 64 KB block of memory accessible by the i80286 and later above 1 MB when running in real mode.

IDE
Integrated Drive Electronics - a system for connecting harddisks to your computer. Used in PC. Supports upto 2 units. Replaced by EIDE.

ISO
International Standard Organization

ISO 9660
A specification for a filesystem on the CD-ROM.

Kb
Kilobit - 1024 bit (2^{10}) - 128 bytes (2^7).

KB
Kilobyte - 1024 bytes (2^{10}).

LFN
Long File Name - a filename that is longer than the DOS 8.3 specification.

Mb
Megabit - 1048576 bit (2^{17}) = 131072 bytes (2^{17}).

MB
Megabyte - 1048576 bytes (2^{20}).

OS
Operating System(s) - I honestly do not care to write operating system each time since it occurs quite often in these texts ;).

Paging
The process of realizing virtual memory in physical memory by moving blocks of physical memory to and from a slower storage media (usually a disk).

Partition
Sub-division of a disk into smaller logical disks.

PhotoCD
Kodak format for storage of images on CD-ROM

Process
A collection of threads that share resources.

Protected Mode
Refers to the Intel 80286 Protected Mode architecture. When the CPU runs in this mode, it supports virtual memory, memory access restrictions and task-switching in hardware. This is the prefered mode for all new operating system and the only mode where you can access memory above 1 MB.

Real Mode
Refers to the Intel 80286 Protected Mode architecture. The mode in which the 8086 ran. It supports upto 1 MB of address space and has no access restrictions whatsoever.

Ring 0
Refers to the Intel 80286 Protected Mode architecture. The least restricted level with access to all system resources. Should only be used by the OS and its drivers.

Ring 3
Refers to the Intel 80286 Protected Mode architecture. The most restricted level of protection. Should be used by all user applications.

SCSI
Small Computer Systems Interface - a general bus for connecting additional devices to a computer. Modified and exists in serveral variations today, known as SCSI-2, SCSI-3, Ultra-SCSI 1+2, Fast SCSI 1+2, Wide SCSI and serveral other combinations. Basic SCSI uses a 8-bit parallel bus running at 5 MHz giving it a transfer rate of 5MB/s.

Sector
Smallets unit data storage on a disk. Often blocks with a size of 512, 1024, 2048 or 4096 bytes (CD-ROM can use 2352).

Storage Media
A media capable of storing data permanently. Often the shape of a disc or as a tape.

Thread
A thread of execution is the series of machine instructions that the CPU executes.

UID
User ID - a number or a string that represents the user uniquely within the OS.

UMB
Upper Memory Block - in DOS 5 or later, unused blocks of memory in the 640 KB - 1 MB memory range can be used to store device drivers.

Unicode
A newer way of supporting all foregin languages special characters. Used instead of DBCS. Uses two bytes to represent one character, supports upto 65536 different characters. Used primarily in WinNT and Win95/98.

V86 Mode
Refers to the Intel 80386 Protected Mode architecture. A mode where the CPU emulates the 8086 real mode addressing but maintains support for paging and certain access restrictions. Often used by OS's to implement virtual DOS machines and can be used to implement a EMS memory manager (like QEMM386 or EMM386).

Virtual Memory
A CPU-addressible memory area, which does not exist in real memory, but is created on a slower storage media. The process of swapping virtual memory in and out of real memory is called paging.

Volume
An area of a disk containing a filesystem of some kind.

XMS
eXtended Memory Specification - a system to access additional memory in DOS. It uses a copying mechanism to copy to/from conventional memory to/from extended memory.

REFERENCE BOOKS:

Author Dahmke, Mark.
Main Title Microcomputer Operating Systems / Mark Dahmke.
Publisher Peterborough, N.H : Mcgraw-Hill / Byte Books, C1982.

Author Deitel, Harvey M., 1945-
Main Title An Introduction To Operating Systems / Harvey M. Deitel.
Edition Rev. 1st Ed.
Publisher Reading, Mass : Addison-Wesley Pub. Co., C1984.

Author Lister, A. (Andrew), 1945-
Main Title Fundamentals Of Operating Systems / A.M. Lister.
Edition 3rd Ed.
Publisher London : Macmillan, 1984.

Author Gray, N. A. B. (Neil A. B.)
Main Title Introduction To Computer Systems / N.A.B. Gray.
Publisher Englewood Cliffs, New Jersey ; Sydney : Prentice-Hall, 1987.

Author Peterson, James L.
Main Title Operating System Concepts / James L. Peterson, Abraham Silberschatz.
Edition 2nd Ed.
Publisher Reading, Mass. : Addison-Wesley, 1985.

Author Stallings, William.
Main Title Operating Systems / William Stallings.
Edition 6th Ed.
Publisher Englewood Cliffs, Nj : Prentice Hall, C1995.

Author Tanenbaum, Andrew S., 1944-
Main Title Operating Systems : Design And Implementation / Andrew S. Tanenbaum, Albert S. Woodhull.
Edition 2nd Ed.
Publisher Upper Saddle River, Nj : Prentice Hall, C1997.

Author Nutt, Gary J.
Main Title Operating Systems : A Modern Perspective / Gary J. Nutt.
Publisher Reading, Mass. : Addison-Wesley, C1997.

Author Silberschatz, Abraham.
Main Title Operating System Concepts / Abraham Silberschatz, Peter Baer Galvin.
Edition 6th Ed.
Publisher Reading, Mass. : Addison Wesley Longman, C1998.

Useful Web Sites

- http://www.d.umn.edu/~tpederse/Courses/CS3221-SPR04/class.html
- http://www.cse.msu.edu/~cse410/
- http://cs.wisc.edu_~solomon_cs537_notes.html/~solomon_cs537_intro.html
- http:

  ww2.cs.uregina.ca_~hamilton_courses_330_notes_index.ht~hamilton_courses_330_notes_index.html
- http://cs.nyu.edu_~gottlieb_courses_2000-01-spring_os_lectures/~gottlieb_courses_2000-01-spring_os_lectures_lectures.html
- http://cs.gordon.edu_courses_cs322_lectures_index.html/courses_cs322_lectures_index.html