1.2 OPERATING SYSTEM

1.2.1 Definition of Operating System:

- An Operating system is a program that controls the execution of application programs and acts as an interface between the user of a computer and the computer hardware.
- A more common definition is that the operating system is the one program running at all times on the computer (usually called the kernel), with all else being applications programs.
- An Operating system is concerned with the allocation of resources and services, such as memory, processors, devices and information. The Operating System correspondingly includes programs to manage these resources, such as a traffic controller, a scheduler, memory management module, I/O programs, and a file system.

1.2.2 Functions of Operating System

Operating system performs three functions:

1. **Convenience:** An OS makes a computer more convenient to use.
2. **Efficiency**: An OS allows the computer system resources to be used in an efficient manner.
3. **Ability to Evolve:** An OS should be constructed in such a way as to permit the effective development, testing and introduction of new system functions without at the same time interfering with service.

1.2.3 Operating System as User Interface

- Every general purpose computer consists of the hardware, operating system, system programs, application programs. The hardware consists of memory, CPU, ALU, I/O devices, peripheral device and storage device. System program consists of compilers, loaders, editors, OS etc. The application program consists of business program, database program.
- The fig. 1.1 shows the conceptual view of a computer system

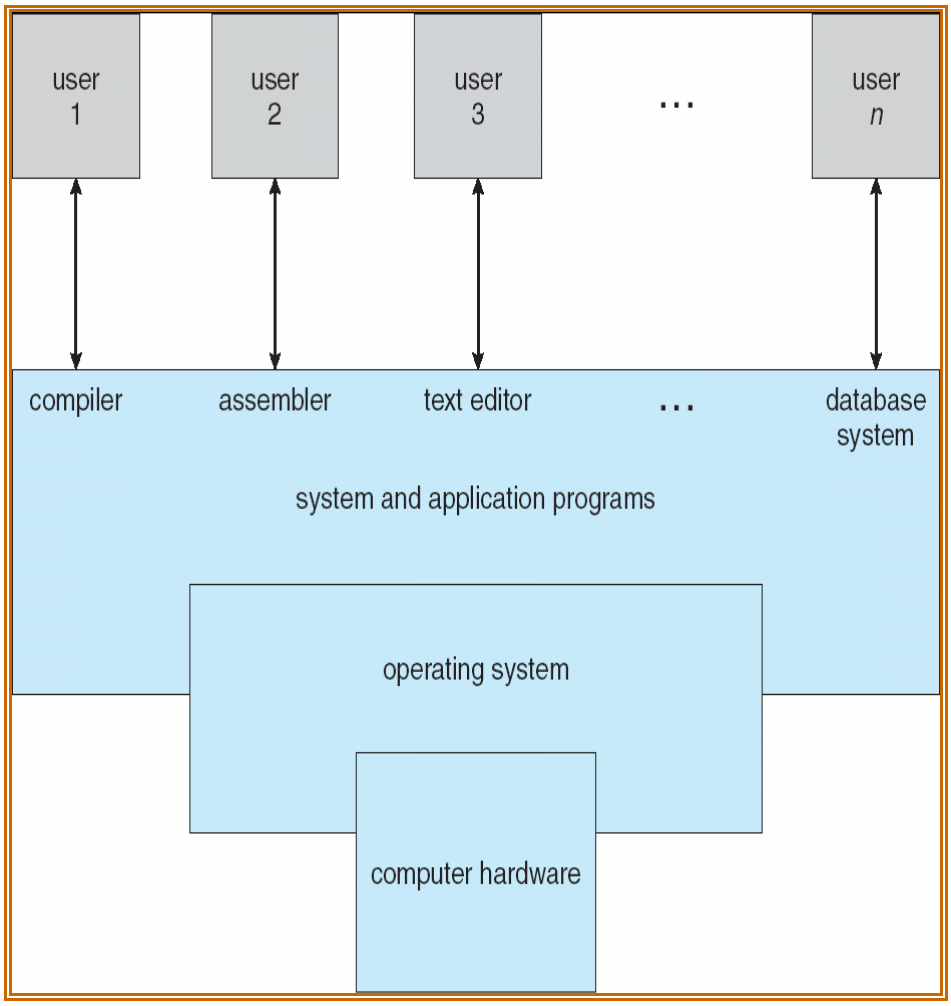


Fig 1.1 Conceptual view of a computer system

- Every computer must have an operating system to run other programs. The operating system and coordinates the use of the hardware among the various system programs and application program for a various users. It simply provides an environment within which other programs can do useful work.
- The operating system is a set of special programs that run on a computer system that allow it to work properly. It performs basic tasks such as recognizing input from the keyboard, keeping track of files and directories on the disk, sending output to the display screen and controlling a peripheral devices.

- OS is designed to serve two basic purposes :

1. It controls the allocation and use of the computing system's resources among the various user and tasks.

2. It provides an interface between the computer hardware and the programmer that simplifies and makes feasible for coding, creation, debugging of application programs.

- The operating system must support the following tasks. The tasks are :

1. Provides the facilities to create, modification of program and data files using and editor.
2. Access to the compiler for translating the user program from high level language to machine language.
3. Provide a loader program to move the compiled program code to the computer's memory for execution.
4. Provide routines that handle the details of I/O programming.

1.3 I/O SYSTEM MANAGEMENT

I/O System Management

- The module that keeps track of the status of devices is called the I/O traffic controller. Each I/O device has a device handler that resides in a separate process associated with that device.

- The I/O subsystem consists of

1. A memory management component that includes buffering, caching and spooling.
2. A general device driver interface.

Drivers for specific hardware devices.

1.4 ASSEMBLER

Input to an assembler is an assembly language program. Output is an object program plus information that enables the loader to prepare the object program for execution. At one time, the computer programmer had at his disposal a basic machine that interpreted, through hardware, certain fundamental instructions. He would program this computer by writing a series of ones and zeros(machine language), place them into the memory of the machine.

1.5 COMPILER

The high level languages – examples are FORTRAN, COBOL, ALGOL and PL/I – are processed by compilers and

interpreters. A compilers is a program that accepts a source program in a “high-level language” and produces a corresponding object program. An interpreter is a program that appears to execute a source program as if it was machine language. The same name (FORTRAN, COBOL etc) is often used to designate both a compiler and its associated language.

1.6 LOADER

A loader is a routine that loads an object program and prepares it for execution. There are various loading schemes: absolute, relocating and direct-linking. In general, the loader must load, relocate, and link the object program. Loader is a program that places programs into memory and prepares them for execution. In a simple loading scheme, the assembler outputs the machine language translation of a program on a secondary device and a loader is placed in core. The loader places into memory the machine language version of the user’s program and transfers control to it. Since the loader program is much smaller than the assembler, thos makes more core available to user’s program.

1.7 HISTORY OF OPERATING SYSTEM

- Operating systems have been evolving through the years. Following table shows the history of OS.

Generation	Year	Electronic devices used	Types of OS and devices
First	1945 – 55	Vacuum tubes	Plug boards
Second	1955 – 1965	Transistors	Batch system
Third	1965 – 1980	Integrated Circuit (IC)	Multiprogramming
Fourth	Since 1980	Large scale integration	PC

1.8 SUMMARY

Operating Systems:

An Operating system is concerned with the allocation of resources and services, such as memory, processors, devices and information. The Operating System correspondingly includes programs to manage these resources, such as a traffic controller, a

2

FUNDAMENTAL OF OPERATING SYSTEM

Unit Structure

2.0 OBJECTIVES

After going through this unit, you will be able to:

- To describe the services an operating system provides to users, processes, and other systems
- Describe operating system services and its components.
- Define multitasking and multiprogramming.
- Describe timesharing, buffering & spooling.

2.1 INTRODUCTION

An operating system provides the environment within which programs are executed. Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. The design of a new operating system is a major task. It is important that the goals of the system be well defined before the design begins.

We can view an operating system from several vantage points. One view focuses on the services that the system provides, another, on the interface that it makes available to users and programmers; a third, on its components and their interconnections.

2.2 OPERATING SYSTEM SERVICES

- An operating system provides services to programs and to the users of those programs. It provided by one environment for the execution of programs. The services provided by one operating system is difficult than other operating system. Operating system makes the programming task easier.

- The common service provided by the operating system is listed below.

1. Program execution
2. I/O operation
3. File system manipulation
4. Communications
5. Error detection

1. **Program execution**: Operating system loads a program into memory and executes the program. The program must be able to end its execution, either normally or abnormally.

2. **I/O Operation** : I/O means any file or any specific I/O device. Program may require any I/O device while running. So operating system must provide the required I/O.

3. **File system manipulation** : Program needs to read a file or write a file. The operating system gives the permission to the program for operation on file.

4. **Communication** : Data transfer between two processes is required for some time. The both processes are on the one computer or on different computer but connected through computer network. Communication may be implemented by two methods:
   a. Shared memory
   b. Message passing.

5. **Error detection** : error may occur in CPU, in I/O devices or in the memory hardware. The operating system constantly needs to be aware of possible errors. It should take the appropriate action to ensure correct and consistent computing.

- Operating system with multiple users provides following services.

1. Resource Allocation
2. Accounting
3. Protection

A) Resource Allocation :

- If there are more than one user or jobs running at the same time, then resources must be allocated to each of them. Operating system manages different types of resources require special allocation code, i.e. main memory, CPU cycles and file storage.

- There are some resources which require only general request and release code. For allocating CPU, CPU scheduling algorithms are used for better utilization of CPU. CPU scheduling algorithms are used for better utilization of CPU. CPU scheduling routines consider the speed of the CPU, number of available registers and other required factors.

B) Accounting :

- Logs of each user must be kept. It is also necessary to keep record of which user how much and what kinds of computer resources. This log is used for accounting purposes.

- The accounting data may be used for statistics or for the billing. It also used to improve system efficiency.

C) Protection :

- Protection involves ensuring that all access to system resources is controlled. Security starts with each user having to authenticate to the system, usually by means of a password. External I/O devices must be also protected from invalid access attempts.

- In protection, all the access to the resources is controlled. In multiprocess environment, it is possible that, one process to interface with the other, or with the operating system, so protection is required.

2.3 OPERATING SYSTEM COMPONENTS

- Modern operating systems share the goal of supporting the system components. The system components are :

1. Process Management
2. Main Memory Management
3. File Management
4. Secondary Storage Management
5. I/O System Management
6. Networking
7. Protection System
8. Command Interpreter System

2.4 BATCH SYSTEM

- Some computer systems only did one thing at a time. They had a list of the computer system may be dedicated to a single program until its completion, or they may be dynamically reassigned among a collection of active programs in different stages of execution.

- Batch operating system is one where programs and data are collected together in a batch before processing starts. A job is predefined sequence of commands, programs and data that are combined in to a single unit called job.

- Fig. 2.1 shows the memory layout for a simple batch system. Memory management in batch system is very simple. Memory is usually divided into two areas : Operating system and user program area.

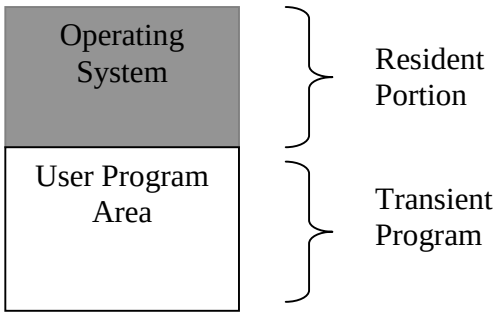


Fig 2.1 Memory Layout for a Simple Batch System

- Scheduling is also simple in batch system. Jobs are processed in the order of submission i.e first come first served fashion.

- When job completed execution, its memory is releases and the output for the job gets copied into an output **spool** for later printing.

- Batch system often provides simple forms of file management. Access to file is serial. Batch systems do not require any time critical device management.

- Batch systems are inconvenient for users because users can not interact with their jobs to fix problems. There may also be long turn around times. Example of this system id generating monthly bank statement.

Advantages o Batch System

- Move much of the work of the operator to the computer.
- Increased performance since it was possible for job to start as soon as the previous job finished.

Disadvantages of Batch System

- Turn around time can be large from user standpoint.
- Difficult to debug program.
- A job could enter an infinite loop.
- A job could corrupt the monitor, thus affecting pending jobs.
- Due to lack of protection scheme, one batch job can affect pending jobs.

2.5 TIME SHARING SYSTEMS

- Multi-programmed batched systems provide an environment where the various system resources (for example, CPU, memory, peripheral devices) are utilized effectively.

- Time sharing, or multitasking, is a logical extension of multiprogramming. Multiple jobs are executed by the CPU switching between them, but the switches occur so frequently that the users may interact with each program while it is running.

- An interactive, or hands-on, computer system provides on-line communication between the user and the system. The

user gives instructions to the operating system or to a program directly, and receives an immediate response. Usually, a keyboard is used to provide input, and a display screen (such as a cathode-ray tube (CRT) or monitor) is used to provide output.

- If users are to be able to access both data and code conveniently, an on-line file system must be available. A file is a collection of related information defined by its creator. Batch systems are appropriate for executing large jobs that need little interaction.

- Time-sharing systems were developed to provide interactive use of a computer system at a reasonable cost. A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory. A program that is loaded into memory and is executing is commonly referred to as a process. When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. I/O may be interactive; that is, output is to a display for the user and input is from a user keyboard. Since interactive I/O typically runs at people speeds, it may take a long time to completed.

- A time-shared operating system allows the many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that she has her own computer, whereas actually one computer is being shared among many users.

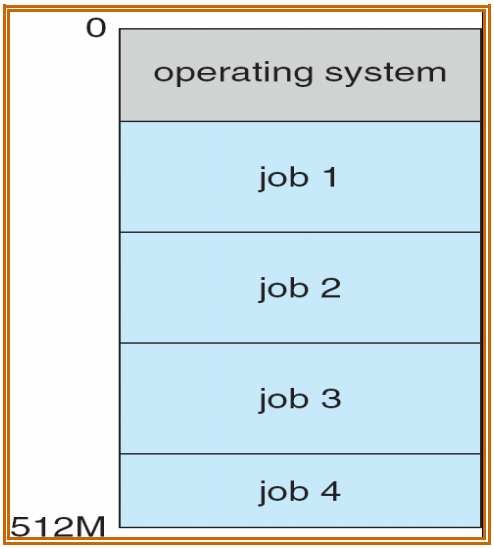
- Time-sharing operating systems are even more complex than are multi-programmed operating systems. As in multiprogramming, several jobs must be kept simultaneously in memory, which requires some form of memory management and protection.

2.6 MULTIPROGRAMMING

- When two or more programs are in memory at the same time, sharing the processor is referred to the multiprogramming operating system. Multiprogramming assumes a single processor that is being shared. It

increases CPU utilization by organizing jobs so that the CPU always has one to execute.

- **Fig. 2.2** shows the memory layout for a multiprogramming system.



- The operating system keeps several jobs in memory at a time. This set of jobs is a subset of the jobs kept in the job pool. The operating system picks and begins to execute one of the job in the memory.

- Multiprogrammed system provide an environment in which the various system resources are utilized effectively, but they do not provide for user interaction with the computer system.

- Jobs entering into the system are kept into the memory. Operating system picks the job and begins to execute one of the job in the memory. Having several programs in memory at the same time requires some form of memory management.

- Multiprogramming operating system monitors the state of all active programs and system resources. This ensures that the CPU is never idle unless there are no jobs.

Advantages

1. High CPU utilization.

2. It appears that many programs are allotted CPU almost simultaneously.

Disadvantages

1. CPU scheduling is requires.
2. To accommodate many jobs in memory, memory management is required.

2.7 SPOOLING

- Acronym for simultaneous peripheral operations on line. Spooling refers to putting jobs in a buffer, a special area in memory or on a disk where a device can access them when it is ready.

- Spooling is useful because device access data that different rates. The buffer provides a waiting station where data can rest while the slower device catches up. Fig 2.3 shows the spooling.

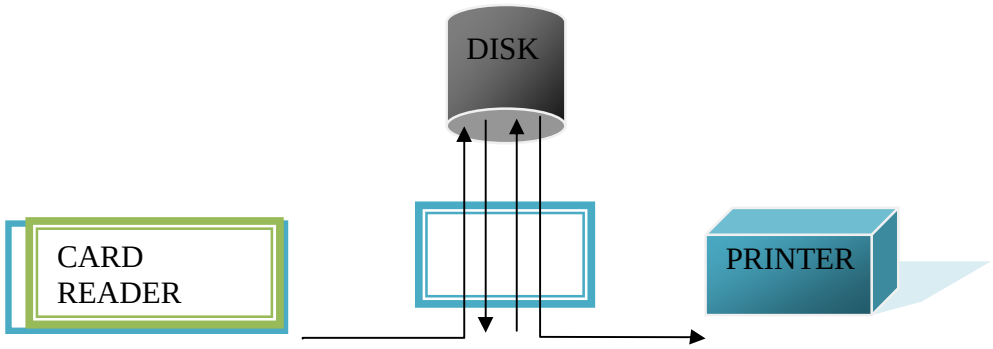


Fig 2.3 Spooling Process

- Computer can perform I/O in parallel with computation, it becomes possible to have the computer read a deck of cards to a tape, drum or disk and to write out to a tape printer while it was computing. This process is called spooling.

- The most common spooling application is print spooling. In print spooling, documents are loaded into a buffer and then the printer pulls them off the buffer at its own rate.

- Spooling is also used for processing data at remote sites. The CPU sends the data via communications path to a remote printer. Spooling overlaps the I/O of one job with the computation of other jobs.

- One difficulty with simple batch systems is that the computer still needs to read the decks of cards before it can begin to

execute the job. This means that the CPU is idle during these relatively slow operations.

- Spooling batch systems were the first and are the simplest of the multiprogramming systems.

Advantage of Spooling

1. The spooling operation uses a disk as a very large buffer.

2. Spooling is however capable of overlapping I/O operation for one job with processor operations for another job.

2.8 ESSENTIAL PROPERTIES OF THE OPERATING SYSTEM

1. Batch : Jobs with similar needs are batched together and run through the computer as a group by an operator or automatic job sequencer. Performance is increased by attempting to keep CPU and I/O devices busy at all times through buffering , off line operation, spooling and multiprogramming. A Batch system is good for executing large jobs that need little interaction, it can be submitted and picked up latter.

2. Time sharing : Uses CPU s scheduling and multiprogramming to provide economical interactive use of a system. The CPU switches rapidly from one user to another i.e. the CPU is shared between a number of interactive users. Instead of having a job defined by spooled card images, each program reads its next control instructions from the terminal and output is normally printed immediately on the screen.

3. Interactive : User is on line with computer system and interacts with it via an interface. It is typically composed of many short transactions where the result of the next transaction may be unpredictable. Response time needs to be short since the user submits and waits for the result.

4. Real time system : Real time systems are usually dedicated, embedded systems. They typically read from and react to sensor data. The system must guarantee response to events within fixed periods of time to ensure correct performance.

5. Distributed : Distributes computation among several physical processors. The processors do not share memory or a clock.

3

PROCESS MANAGEMENT

Unit Structure

3.0 OBJECTIVES

After going through this unit, you will be able to:

- To introduce the notion of a process – a program in execution, which forms the basis of all computation

- To describe the various features of processes, including scheduling, creation and termination, and communication.

3.1 CONCEPT OF PROCESS

- A process is sequential program in execution. A process defines the fundamental unit of computation for the computer. Components of process are :

1. Object Program
2. Data
3. Resources
4. Status of the process execution.

- Object program i.e. code to be executed. Data is used for executing the program. While executing the program, it may require some resources. Last component is used for verifying the status of the process execution. A process can run to completion only when all requested resources have been allocated to the process. Two or more processes could be executing the same program, each using their own data and resources.

3.1.1 Processes and Programs

- Process is a dynamic entity, that is a program in execution. A process is a sequence of information executions. Process exists in a limited span of time. Two or more processes could be executing the same program, each using their own data and resources.

- Program is a static entity made up of program statement. Program contains the instructions. A program exists at single place in space and continues to exist. A program does not perform the action by itself.

3.2 PROCESS STATE

- When process executes, it changes state. Process state is defined as the current activity of the process. Fig. 3.1 shows the general form of the process state transition diagram. Process state contains five states. Each process is in one of the states. The states are listed below.

1. New
2. Ready
3. Running
4. Waiting
5. Terminated(exist)

1. **New :** A process that just been created.

2. **Ready :** Ready processes are waiting to have the processor allocated to them by the operating system so that they can run.

3. **Running :** The process that is currently being executed. A running process possesses all the resources needed for its execution, including the processor.

4. **Waiting :** A process that can not execute until some event occurs such as the completion of an I/O operation. The

running process may become suspended by invoking an I/O module.

5. **Terminated** : A process that has been released from the pool of executable processes by the operating system.

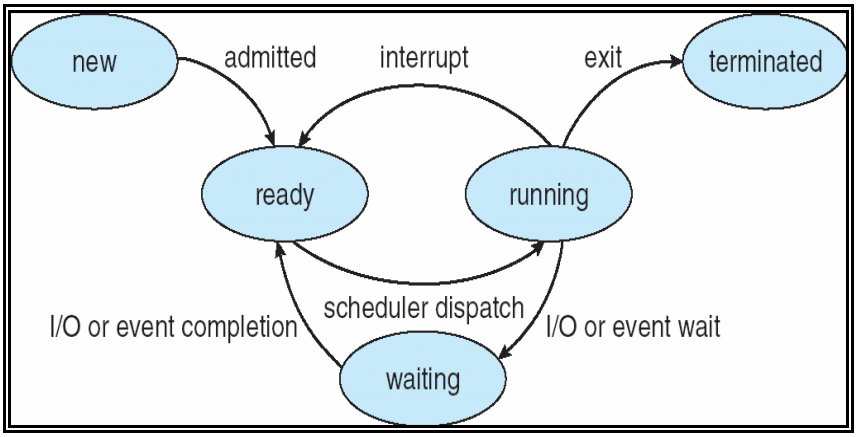


Fig 3.1 Diagram for Process State

- Whenever processes changes state, the operating system reacts by placing the process PCB in the list that corresponds to its new state. Only one process can be running on any processor at any instant and many processes may be ready and waiting state.

3.2.1 Suspended Processes

Characteristics of suspend process

1. Suspended process is not immediately available for execution.
2. The process may or may not be waiting on an event.
3. For preventing the execution, process is suspend by OS, parent process, process itself and an agent.
4. Process may not be removed from the suspended state until the agent orders the removal.

- Swapping is used to move all of a process from main memory to disk. When all the process by putting it in the suspended state and transferring it to disk.

Reasons for process suspension

1. Swapping
2. Timing
3. Interactive user request
4. Parent process request

Swapping: OS needs to release required main memory to bring in a process that is ready to execute.

Timing: Process may be suspended while waiting for the next time interval.
Interactive user request: Process may be suspended for debugging purpose by user.

Parent process request: To modify the suspended process or to coordinate the activity of various descendants.

3.2.2 Process Control Block (PCB)

- Each process contains the process control block (PCB). PCB is the data structure used by the operating system. Operating system groups all information that needs about particular process. Fig. 3.2 shows the process control block.

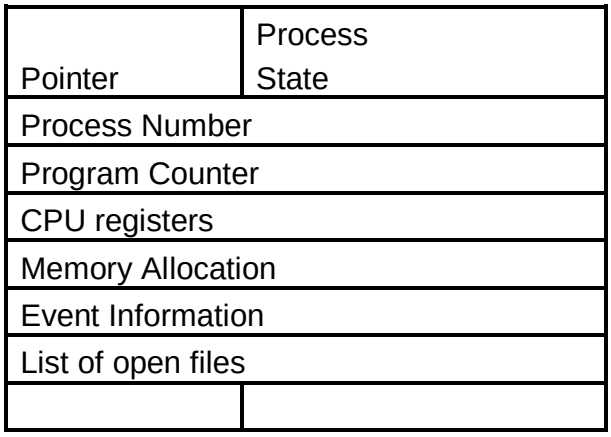


Fig. 3.2 Process Control Block

1. **Pointer :** Pointer points to another process control block. Pointer is used for maintaining the scheduling list.

2. **Process State :** Process state may be new, ready, running, waiting and so on.

3. **Program Counter :** It indicates the address of the next instruction to be executed for this process.

4. **Event information :** For a process in the blocked state this field contains information concerning the event for which the process is waiting.

5. **CPU register :** It indicates general purpose register, stack pointers, index registers and accumulators etc. number of register and type of register totally depends upon the computer architecture.

6. **Memory Management Information :** This information may include the value of base and limit register. This information is useful for deallocating the memory when the process terminates.

7. **Accounting Information :** This information includes the amount of CPU and real time used, time limits, job or process numbers, account numbers etc.

- Process control block also includes the information about CPU scheduling, I/O resource management, file management information, priority and so on. The PCB simply serves as the repository for any information that may vary from process to process.

- When a process is created, hardware registers and flags are set to the values provided by the loader or linker. Whenever that process is suspended, the contents of the processor register are usually saved on the stack and the pointer to the related stack frame is stored in the PCB. In this way, the hardware state can be restored when the process is scheduled to run again.

3.3 PROCESS MANAGEMENT / PROCESS SCHEDULING

- Multiprogramming operating system allows more than one process to be loaded into the executable memory at a time and for the loaded process to share the CPU using time multiplexing.

- The scheduling mechanism is the part of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of particular strategy.

3.3.1 Scheduling Queues

- When the process enters into the system, they are put into a job queue. This queue consists of all processes in the system. The operating system also has other queues.

- Device queue is a queue for which a list of processes waiting for a particular I/O device. Each device has its own device queue. Fig. 3.3 shows the queuing diagram of process scheduling. In the fig 3.3, queue is represented by rectangular box. The circles represent the resources that serve the queues. The arrows indicate the flow of processes in the system.

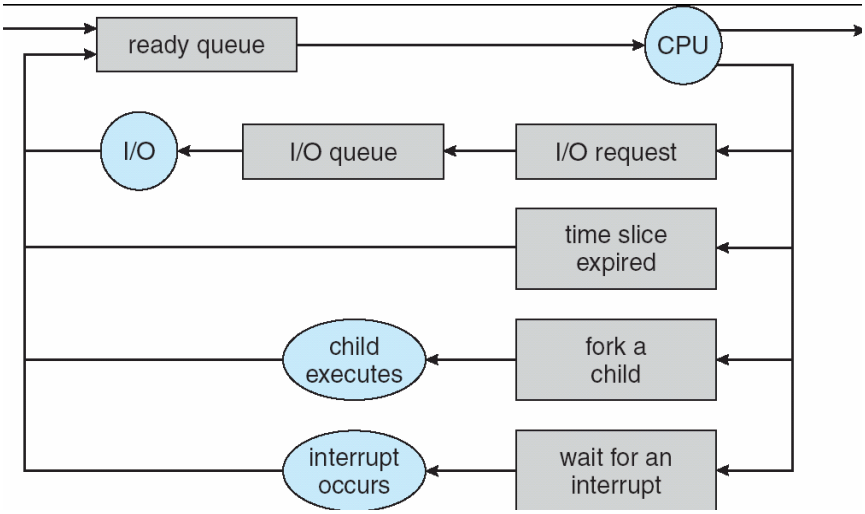


Fig. 3.3 Queuing Diagram

- Queues are of two types : ready queue and set of device queues. A newly arrived process is put in the ready queue. Processes are waiting in ready queue for allocating the CPU. Once the CPU is assigned to the process, then process will execute. While executing the process, one of the several events could occur.

    1. The process could issue an I/O request and then place in an I/O queue.
    2. The process could create new sub process and waits for its termination.
    3. The process could be removed forcibly from the CPU, as a result of interrupt and put back in the ready queue.

3.3.1.1 Two State Process Model

- Process may be in one of two states :
    a) Running
    b) Not Running

- When new process is created by OS, that process enters into the system in the running state.

    - Processes that are not running are kept in queue, waiting their turn to execute. Each entry in the queue is a printer to a particular process. Queue is

implemented by using linked list. Use of dispatcher is as follows. When a process interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then select a process from the queue to execute.

3.3.2 Schedules

- Schedulers are of three types.

1. Long Term Scheduler
2. Short Term Scheduler
3. Medium Term Scheduler

3.3.2.1 Long Term Scheduler

- It is also called job scheduler. Long term scheduler determines which programs are admitted to the system for processing. Job scheduler selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduler. The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

- On same systems, the long term scheduler may be absent or minimal. Time-sharing operating systems have no long term scheduler. When process changes the state from new to ready, then there is a long term scheduler.

3.3.2.2 Short Term Scheduler

- It is also called CPU scheduler. Main objective is increasing system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects from among the processes that are ready to execute and allocates the CPU to one of them.

- Short term scheduler also known as dispatcher, execute most frequently and makes the fine grained decision of which process to execute next. Short term scheduler is faster than long tern scheduler.

3.3.2.3 Medium Term Scheduler

- Medium term scheduling is part of the swapping function. It removes the processes from the memory. It reduces the

degree of multiprogramming. The medium term scheduler is in charge of handling the swapped out-processes.

Medium term scheduler is shown in the **Fig. 3.4**

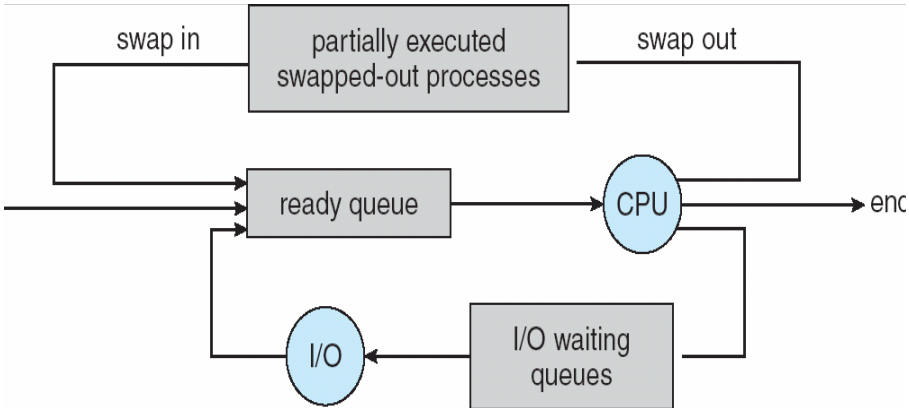


Fig 3.4 Queueing diagram with medium term scheduler

Running process may become suspended by making an I/O request. Suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other process. Suspended process is move to the secondary storage is called swapping, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

3.3.2.4 Comparison between Scheduler

Sr. No.	Long Term	Short Term	Medium Term
1	It is job scheduler	It is CPU Scheduler	It is swapping
2	Speed is less than short term scheduler	Speed is very fast	Speed is in between both
3	It controls degree of multiprogramming	Less control over degree of multiprogramming	Reduce the degree of multiprogramming.
4	Absent or minimal in time sharing system.	Minimal in time sharing system.	Time sharing system use medium term scheduler.

5	It select processes from pool and load them into memory for execution.	It select from among the processes that are ready to execute.	Process can be reintroduced into memory and its execution can be continued.
6	Process state is (New to Ready)	Process state is (Ready to Running)	-
7	Select a good process, mix of I/O bound and CPU bound.	Select a new process for a CPU quite frequently.	-

3.4 CONTEXT SWITCH

- When the scheduler switches the CPU from executing one process to executing another, the context switcher saves the content of all processor registers for the process being removed from the CPU in its process being removed from the CPU in its process descriptor. The context of a process is represented in the process control block of a process. Context switch time is pure overhead. Context switching can significantly affect performance, since modern computers have a lot of general and status registers to be saved.

- Content switch times are highly dependent on hardware support. Context switch requires (n + m) bXK time units to save the state of the processor with n general registers, assuming b store operations are required to save register and each store instruction requires K time units. Some hardware systems employ two or more sets of processor registers to reduce the amount of context switching time.

- When the process is switched the information stored is :
  1. Program Counter
  2. Scheduling Information
  3. Base and limit register value
  4. Currently used register
  5. Changed State
  6. I/O State
  7. Accounting

4

THREAD MANAGEMENT

Unit Structure

4.0 OBJECTIVES

After going through this unit, you will be able to:

- To introduce Thread & its types, Multithreading Models and Threading issues.

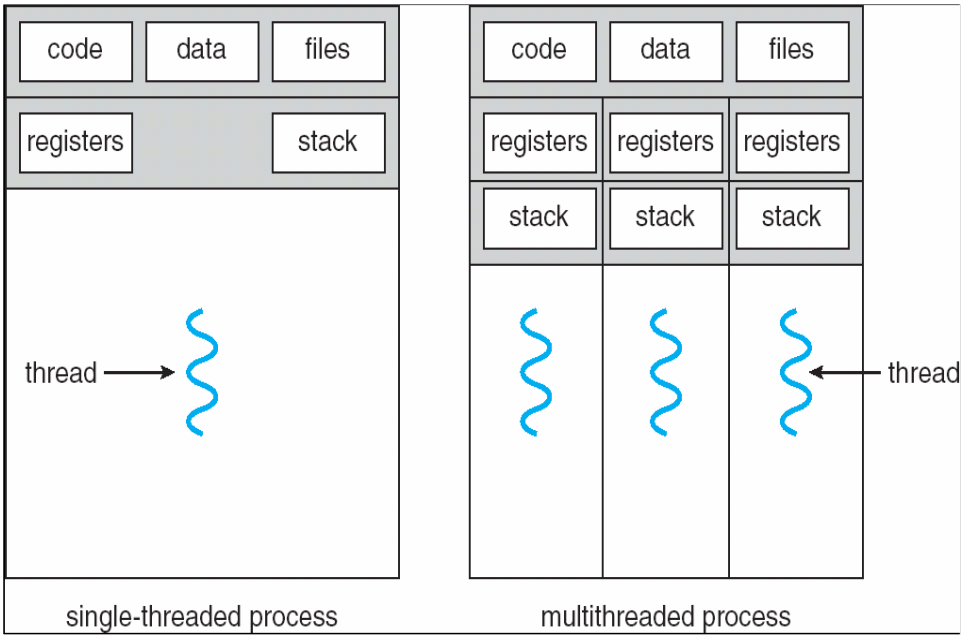
4.1 INTRODUCTION OF THREAD

- A thread is a flow of execution through the process code, with its own program counter, system registers and stack. Threads are a popular way to improve application performance through parallelism. A thread is sometimes called **a light weight process.**

- Threads represent a software approach to improving performance of operating system by reducing the over head

thread is equivalent to a classical process. Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control.

- Fig. 4.1shows the single and multithreaded process.



- Threads have been successfully used in implementing network servers. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors.

4.2 TYPES OF THREAD

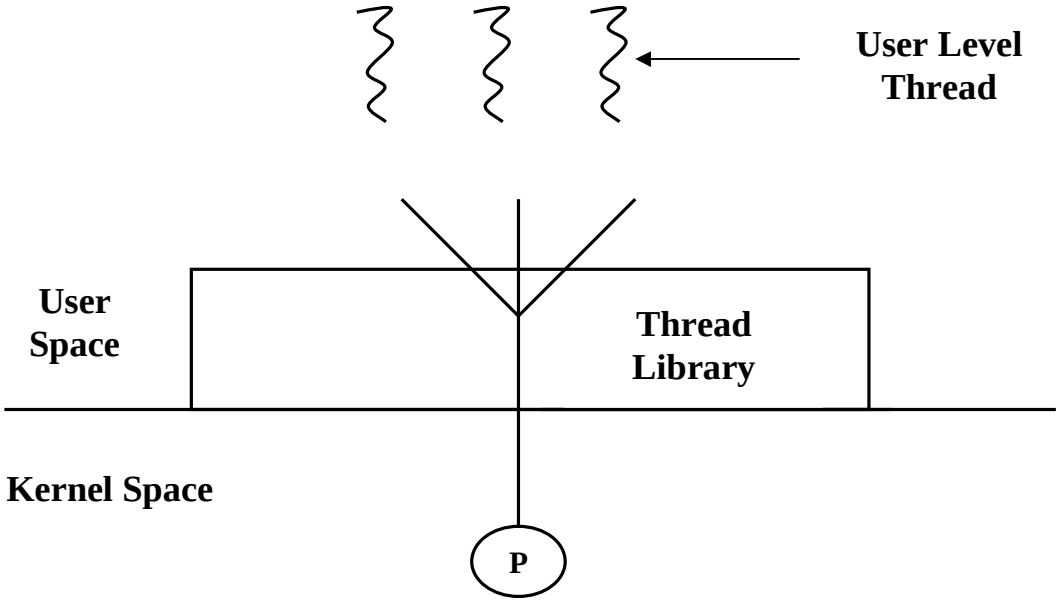
Threads is implemented in two ways :

1. User Level
2. Kernel Level

4.2.1 User Level Thread

- In a user thread, all of the work of thread management is done by the application and the kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application begins with a single thread and begins running in that thread.

- Fig. 4.2 shows the user level thread.



User level threads are generally fast to create and manage.

Advantage of user level thread over Kernel level thread :

1. Thread switching does not require Kernel mode privileges.
2. User level thread can run on any operating system.
3. Scheduling can be application specific.
4. User level threads are fast to create and manage.

Disadvantages of user level thread :

1. In a typical operating system, most system calls are blocking.
2. Multithreaded application cannot take advantage of multiprocessing.

4.2.2 Kernel Level Threads

- In Kernel level thread, thread management done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process. The Kernel maintains context information for the process as a whole and for individuals threads within the process.

- Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Advantages of Kernel level thread:

1. Kernel can simultaneously schedule multiple threads from the same process on multiple process.
2. If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
3. Kernel routines themselves can multithreaded.

Disadvantages:

1. Kernel threads are generally slower to create and manage than the user threads.
2. Transfer of control from one thread to another within same process requires a mode switch to the Kernel.

4.2.3 Advantages of Thread

1. Thread minimize context switching time.
2. Use of threads provides concurrency within a process.
3. Efficient communication.
4. Economy- It is more economical to create and context switch threads.
5. Utilization of multiprocessor architectures –

The benefits of multithreading can be greatly increased in a multiprocessor architecture.

4.3 MULTITHREADING MODELS

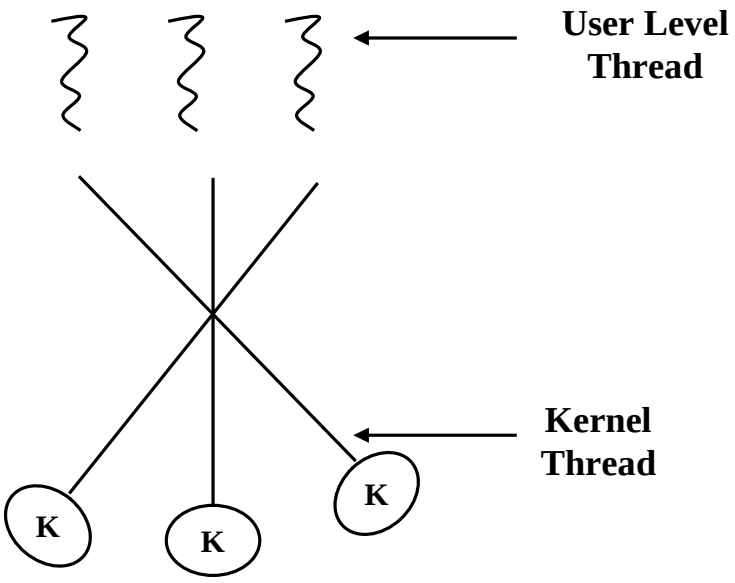
- Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process.
- Multithreading models are three types:

1. Many to many relationship.
2. Many to one relationship.
3. One to one relationship.

4.3.1 Many to Many Model

- In this model, many user level threads multiplexes to the Kernel thread of smaller or equal numbers. The number of Kernel threads may be specific to either a particular application or a particular machine.

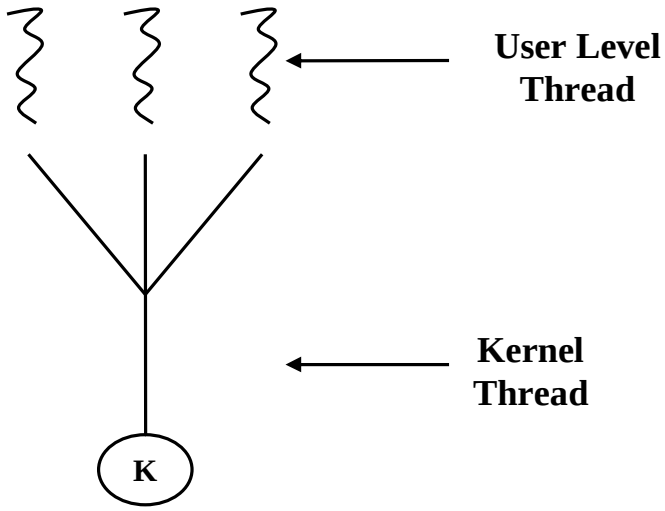
- Fig. 4.3 shows the many to many model. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallels on a multiprocessor.



4.3.2 Many to One Model

- Many to one model maps many user level threads to one Kernel level thread. Thread management is done in user space. When thread makes a blocking system call, the entire process will be blocks. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

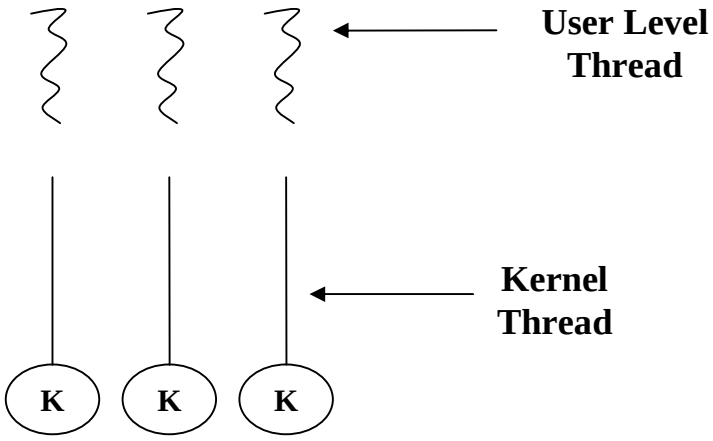
- **Fig.4.4** shows the many to one model.



- If the user level thread libraries are implemented in the operating system, that system does not support Kernel threads use the many to one relationship modes.

4.3.3 One to One Model

- There is one to one relationship of user level thread to the kernel level thread. Fig. 4.5 shows one to one relationship model. This model provides more concurrency than the many to one model.



- It also another thread to run when a thread makes a blocking system call. It support multiple thread to execute in parallel on microprocessors. Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.

4.4 DIFFERENCE BETWEEN USER LEVEL & KERNEL LEVEL THREAD

Sr. No	User Level Threads	Kernel Level Thread
1	User level thread are faster to create and manage.	Kernel level thread are slower to create and manage.
2	Implemented by a thread library at the user level.	Operating system support directly to Kernel threads.
3	User level thread can run on any operating system.	Kernel level threads are specific to the operating system.
4	Support provided at the user level called user level thread.	Support may be provided by kernel is called Kernel level threads.
5	Multithread application cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

4.5 DIFFERENCE BETWEEN PROCESS AND THREAD

Sr. No	Process	Thread
1	Process is called heavy weight process.	Thread is called light weight process.
2	Process switching needs interface with operating system.	Thread switching does not need to call a operating system and cause an interrupt to the Kernel.
3	In multiple process implementation each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one server process is blocked no other server process can execute until the first process unblocked.	While one server thread is blocked and waiting, second thread in the same task could run.

5	Multiple redundant process uses more resources than multiple threaded.	Multiple threaded process uses fewer resources than multiple redundant process.
6	In multiple process each process operates independently of the others.	One thread can read, write or even completely wipe out another threads stack.

4.6 THREADING ISSUES

- System calls fork and exec is discussed here. In a multithreaded program environment, fork and exec system calls is changed. Unix system have two version of fork system calls. One call duplicates all threads and another that duplicates only the thread that invoke the fork system call. Whether to use one or two version of fork system call totally depends upon the application. Duplicating all threads is unnecessary, if exec is called immediately after fork system call.

- Thread cancellation is a process of thread terminates before its completion of task. For example, in multiple thread environment, thread concurrently searching through a database. If any one thread returns the result, the remaining thread might be cancelled.

- Thread cancellation is of two types.

1. Asynchronous cancellation
2. Synchronous cancellation

- In asynchronous cancellation, one thread immediately terminates the target thread. Deferred cancellation periodically check for terminate by target thread. It also allow the target thread to terminate itself in an orderly fashion. Some resources are allocated to the thread. If we cancel the thread, which update the data with other thread. This problem may face by asynchronous cancellation system wide resource are not free if threads cancelled asynchronously. Most of the operating system allow a process or thread to be cancelled asynchronously.

4.7 SUMMARY

A thread is a flow of control within a process. A multithreaded process contains several different flows of control

4

THREAD MANAGEMENT

Unit Structure

4.0 OBJECTIVES

After going through this unit, you will be able to:

- To introduce Thread & its types, Multithreading Models and Threading issues.

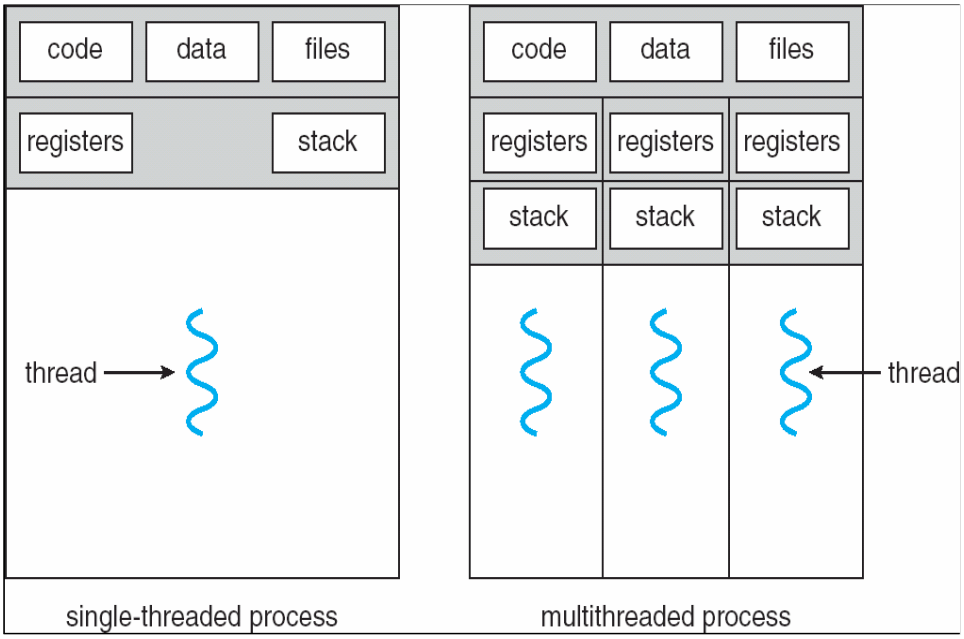
4.1 INTRODUCTION OF THREAD

- A thread is a flow of execution through the process code, with its own program counter, system registers and stack. Threads are a popular way to improve application performance through parallelism. A thread is sometimes called **a light weight process.**

- Threads represent a software approach to improving performance of operating system by reducing the over head

thread is equivalent to a classical process. Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control.

- Fig. 4.1shows the single and multithreaded process.



- Threads have been successfully used in implementing network servers. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors.

4.2 TYPES OF THREAD

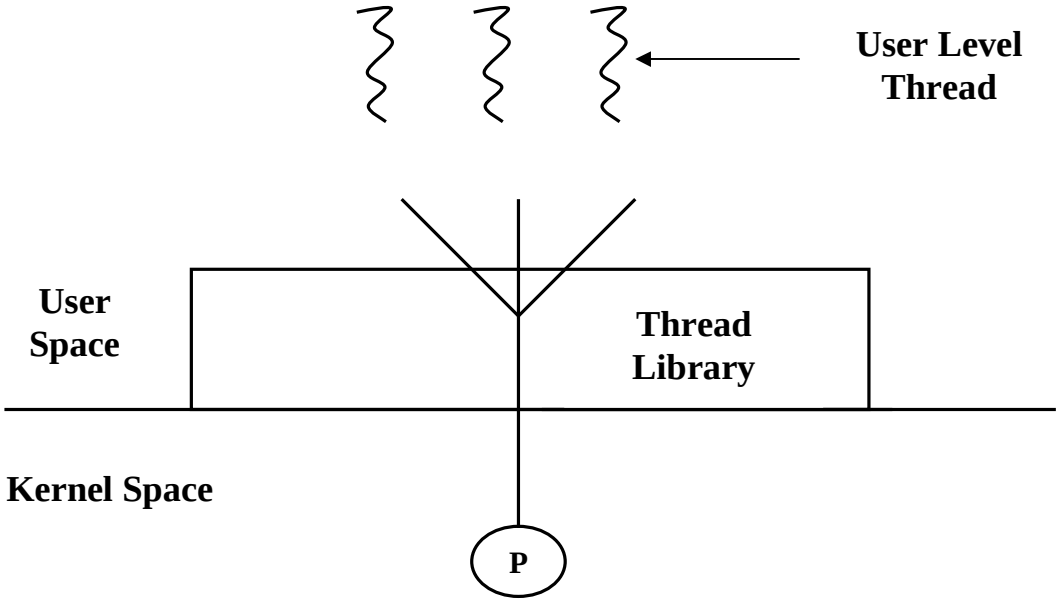
Threads is implemented in two ways :

1. User Level
2. Kernel Level

4.2.1 User Level Thread

- In a user thread, all of the work of thread management is done by the application and the kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application begins with a single thread and begins running in that thread.

- Fig. 4.2 shows the user level thread.



User level threads are generally fast to create and manage.

Advantage of user level thread over Kernel level thread :

1. Thread switching does not require Kernel mode privileges.
2. User level thread can run on any operating system.
3. Scheduling can be application specific.
4. User level threads are fast to create and manage.

Disadvantages of user level thread :

1. In a typical operating system, most system calls are blocking.
2. Multithreaded application cannot take advantage of multiprocessing.

4.2.2 Kernel Level Threads

- In Kernel level thread, thread management done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process. The Kernel maintains context information for the process as a whole and for individuals threads within the process.

- Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Advantages of Kernel level thread:

1. Kernel can simultaneously schedule multiple threads from the same process on multiple process.
2. If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
3. Kernel routines themselves can multithreaded.

Disadvantages:

1. Kernel threads are generally slower to create and manage than the user threads.
2. Transfer of control from one thread to another within same process requires a mode switch to the Kernel.

4.2.3 Advantages of Thread

1. Thread minimize context switching time.
2. Use of threads provides concurrency within a process.
3. Efficient communication.
4. Economy- It is more economical to create and context switch threads.
5. Utilization of multiprocessor architectures –

The benefits of multithreading can be greatly increased in a multiprocessor architecture.

4.3 MULTITHREADING MODELS

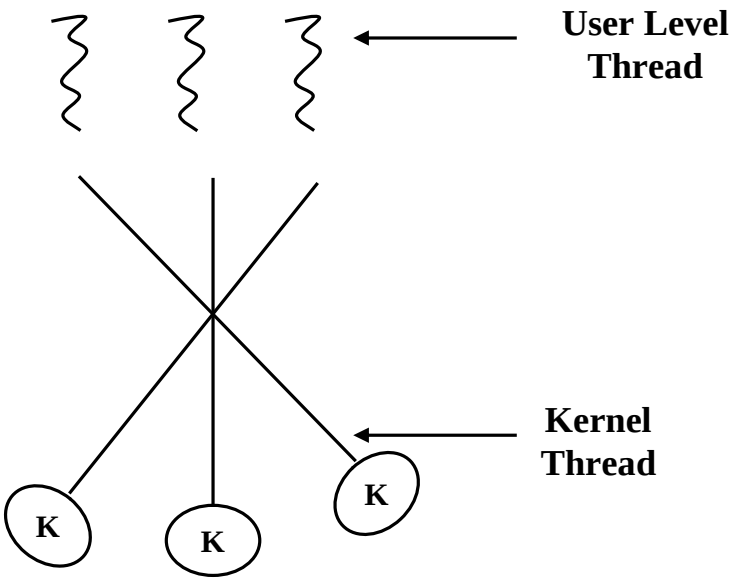
- Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process.
- Multithreading models are three types:

1. Many to many relationship.
2. Many to one relationship.
3. One to one relationship.

4.3.1 Many to Many Model

- In this model, many user level threads multiplexes to the Kernel thread of smaller or equal numbers. The number of Kernel threads may be specific to either a particular application or a particular machine.

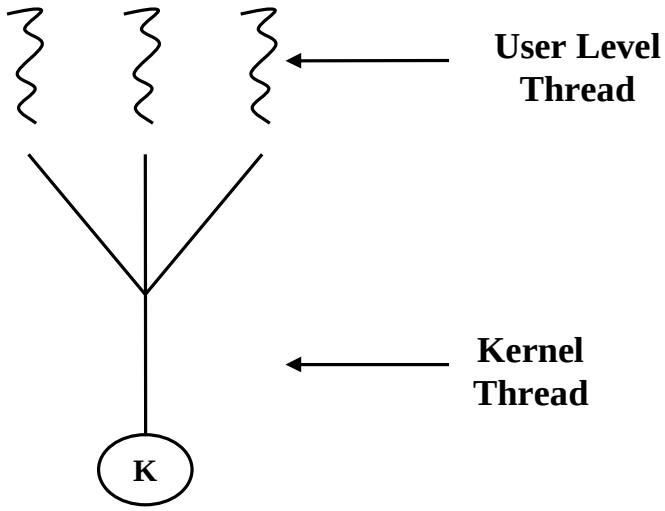
- Fig. 4.3 shows the many to many model. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallels on a multiprocessor.



4.3.2 Many to One Model

- Many to one model maps many user level threads to one Kernel level thread. Thread management is done in user space. When thread makes a blocking system call, the entire process will be blocks. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

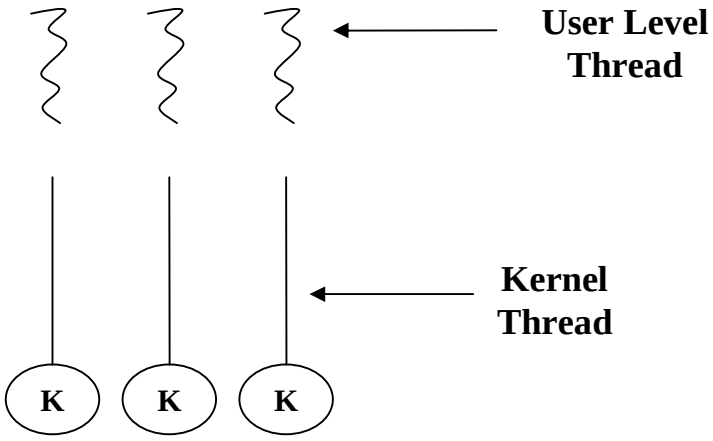
- **Fig.4.4** shows the many to one model.



- If the user level thread libraries are implemented in the operating system, that system does not support Kernel threads use the many to one relationship modes.

4.3.3 One to One Model

- There is one to one relationship of user level thread to the kernel level thread. Fig. 4.5 shows one to one relationship model. This model provides more concurrency than the many to one model.



- It also another thread to run when a thread makes a blocking system call. It support multiple thread to execute in parallel on microprocessors. Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.

4.4 DIFFERENCE BETWEEN USER LEVEL & KERNEL LEVEL THREAD

Sr. No	User Level Threads	Kernel Level Thread
1	User level thread are faster to create and manage.	Kernel level thread are slower to create and manage.
2	Implemented by a thread library at the user level.	Operating system support directly to Kernel threads.
3	User level thread can run on any operating system.	Kernel level threads are specific to the operating system.
4	Support provided at the user level called user level thread.	Support may be provided by kernel is called Kernel level threads.
5	Multithread application cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

4.5 DIFFERENCE BETWEEN PROCESS AND THREAD

Sr. No	Process	Thread
1	Process is called heavy weight process.	Thread is called light weight process.
2	Process switching needs interface with operating system.	Thread switching does not need to call a operating system and cause an interrupt to the Kernel.
3	In multiple process implementation each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one server process is blocked no other server process can execute until the first process unblocked.	While one server thread is blocked and waiting, second thread in the same task could run.

5	Multiple redundant process uses more resources than multiple threaded.	Multiple threaded process uses fewer resources than multiple redundant process.
6	In multiple process each process operates independently of the others.	One thread can read, write or even completely wipe out another threads stack.

4.6 THREADING ISSUES

- System calls fork and exec is discussed here. In a multithreaded program environment, fork and exec system calls is changed. Unix system have two version of fork system calls. One call duplicates all threads and another that duplicates only the thread that invoke the fork system call. Whether to use one or two version of fork system call totally depends upon the application. Duplicating all threads is unnecessary, if exec is called immediately after fork system call.

- Thread cancellation is a process of thread terminates before its completion of task. For example, in multiple thread environment, thread concurrently searching through a database. If any one thread returns the result, the remaining thread might be cancelled.

- Thread cancellation is of two types.

1. Asynchronous cancellation
2. Synchronous cancellation

- In asynchronous cancellation, one thread immediately terminates the target thread. Deferred cancellation periodically check for terminate by target thread. It also allow the target thread to terminate itself in an orderly fashion. Some resources are allocated to the thread. If we cancel the thread, which update the data with other thread. This problem may face by asynchronous cancellation system wide resource are not free if threads cancelled asynchronously. Most of the operating system allow a process or thread to be cancelled asynchronously.

4.7 SUMMARY

A thread is a flow of control within a process. A multithreaded process contains several different flows of control

6

DEADLOCK

Unit Structure

6.0 OBJECTIVES

After going through this unit, you will be able to:

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks.
- To present number of different methods for preventing or avoiding deadlocks in a computer system.

6.1 INTRODUCTION

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. It may happen that waiting processes will never again change state, because the resources they have requested are held by other waiting processes. This situation is called deadlock.

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

A process must request a resource before using it, and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. **Request**: If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can operate on the resource.
3. **Release**: The process releases the resource

6.2 DEADLOCK CHARACTERIZATION

In deadlock, processes never finish executing and system resources are tied up, preventing other jobs from ever starting.

Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. **Mutual exclusion:** At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. **Hold and wait :** There must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.

3. **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.

4. **Circular wait:** There must exist a set {P0, P1, ..., Pn } of waiting processes 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.

6.2.1 Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph. The set

of vertices V is partitioned into two different types of nodes P = {P1, P2, … Pn} the set consisting of all the active processes in the system; and R = {R1, R2, …, R1}, the set consisting of all resource types in the system.

A directed edge from process Pi to resource type Rj is denoted by Pi → Rj, it signifies that process Pi requested an instance of resource type Rj and is currently waiting for that resource. A directed edge from resource type Rj toprocess Pi is denoted by Rj_ Pi it signifies that an instance of resource type Rj has been allocated to process Pi. A directed edge Pi_ Rj is called a request edge; a directed edge Rj _ Pi is called an assignment edge.

When process Pi requests an instance of resource type Rj, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge. When the process no longer needs access to the, resource it releases the resource, and as a result the assignment edge is deleted.

Definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If, on the other hand, the graph contains the cycle, then a deadlock must exist.

If each resource type has several instances, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resources types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

A set of vertices *V* and a set of edges *E*.

- V is partitioned into two types:
    - *P* = {*P*1, *P*2, …, *Pn*}, the set consisting of all the processes in the system.
    - *R* = {*R*1, *R*2, …, *Rm*}, the set consisting of all resource types in the system.
- request edge – directed edge *P*1 → *Rj*
- assignment edge – directed edge *Rj* → *Pi*

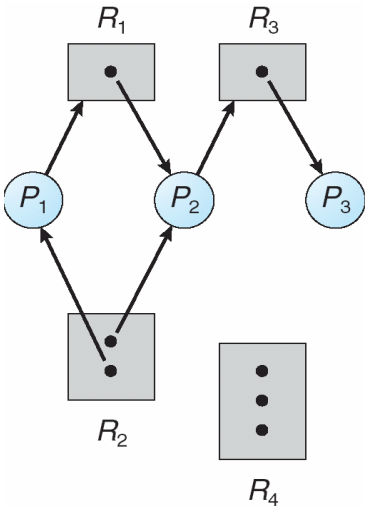


Fig. Resource Allocation Graph

If each resource type has several instance, then a cycle does not necessarily imply that a deadlock incurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

Suppose that process P3requests an instance of resource type R2 Since no resource instance is currently available, a request edge P3 → R2 is added to the graph. At this point, two minimal cycles exist in the system:

P1 → R1 → P2 → R3 → P3 → R2 → P1
P2 → R3 → P3 → R2 → P2

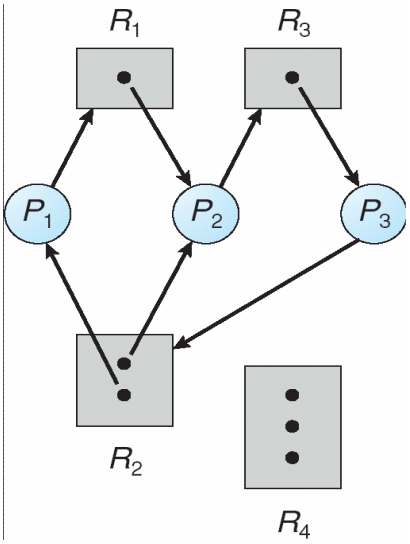


Fig. Resource Allocation Graph with Deadlock

Processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3, on the other hand, is waiting for either process P1 or process P2 to release resource R2. In addition, process PI is waiting for process P2 to release resource R1.

6.3 METHOD FOR HANDLING DEADLOCK /DETECTION

There are are three different methods for dealing with the deadlock problem:

- We can use a protocol to ensure that the system will never enter a deadlock state.
- We can allow the system to enter a deadlock state and then recover.
- We can ignore the problem all together, and pretend that deadlocks never occur in the system. This solution is the one used by most operating systems, including UNIX.

Deadlock avoidance, on the other hand, requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, we can decide for each request whether or not the process should wait. Each request requires that the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must be delayed.

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur If a system does not ensure that a deadlock will never occur, and also does not provide a mechanism for deadlock detection and recovery, then we may arrive at a situation where the system is in a deadlock state yet has no way of recognizing what has happened.

6.4 DEADLOCK PREVENTION

For a deadlock to occur, each of the four necessary-conditions must hold. By ensuring that at least on one these conditions cannot hold, we can prevent the occurrence of a deadlock.

6.4.1 Mutual Exclusion

The mutual-exclusion condition must hold for non-sharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, on the other hand, do not require mutually exclusive access, and thus cannot be involved in a deadlock.

6.4.2 Hold and Wait

1. When whenever a process requests a resource, it does not hold any other resources. One protocol that be used requires each process to request and be allocated all its resources before it begins execution.

2. An alternative protocol allows a process to request resources only when the process has none. A process may request some resources and use them. Before it can request any additional resources, however it must release all the resources that it is currently allocated here are two main disadvantages to these protocols. First, resource utilization may be low, since many of the resources may be allocated but unused for a long period. In the example given, for instance, we can release the tape drive and disk file, and then again request the disk file and printer, only if we can be sure that our data will remain on the disk file. If we cannot be assured that they will, then we must request all resources at the beginning for both protocols. Second, starvation is possible.

6.4.3 No Preemption

If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are preempted. That is this resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

6.4.4 Circular Wait

Circular-wait condition never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration.

Let R = {R1, R2, ..., Rn} be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function F: R _ N, where N is the set of natural numbers.

6.5 DEADLOCK AVOIDANCE

Prevent deadlocks requests can be made. The restraints ensure that at least one of the necessary conditions for deadlock cannot occur, and, hence, that deadlocks cannot hold. Possible side effects of preventing deadlocks by this, melted, however, are Tow device utilization and reduced system throughput.

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. For example, in a system with one tape drive and one printer, we might be told that process P will request first the tape drive, and later the printer, before releasing both resources. Process Q on the other hand, will request first the printer, and then the tape drive. With this knowledge of the complete sequence of requests and releases for each process we can decide for each request whether or not the process should wait.

A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular wait condition. The resource allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

6.5.1 Safe State

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. A sequence of processes <P1, P2, .. Pn> is a safe sequence for the current allocation state if, for each Pi the resources that Pj can still request can be satisfied by the currently available resources plus the resources held by all the Pj, with j < i. In this situation, if the resources that process Pi needs are not immediately available, then Pi can wait until all Pj have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task return its allocated resources, and terminate. When Pi terminates, Pi + 1 can obtain its needed resources, and so on.

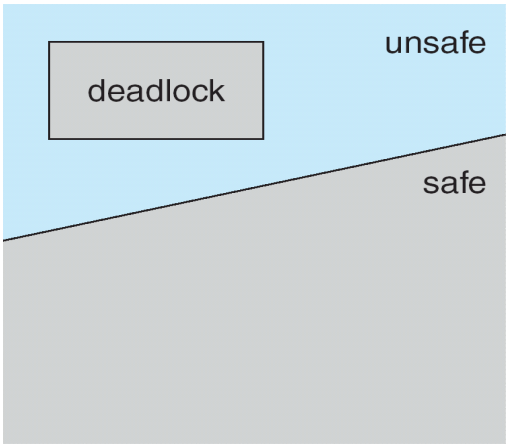


Fig. Safe, Unsafe & Deadlock State

If no such sequence exists, then the system state is said to be unsafe.

6.5.2 Resource-Allocation Graph Algorithm

Suppose that process Pi requests resource Rj. The request can be granted only if converting the request edge Pi → Rj to an assignment edge Rj → Pi does not result in the formation of a cycle in the resource-allocation graph.

6.5.3 Banker's Algorithm

The resource-allocation graph algorithm is not applicable to a resource-allocation system with multiple instances of each resource type. The deadlock-avoidance algorithm that we describe next is applicable to such a system, but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the banker's algorithm.

6.6 DEADLOCK DETECTION

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur.

• An algorithm that examines the state of the system to determine whether a deadlock has occurred.

• An algorithm to recover from the deadlock.

6.6.1 Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph. We obtain this graph from the resource-allocation graph by removing the nodes of type resource and collapsing the appropriate edges.

6.6.2 Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.

The algorithm used are :

• **Available**: A vector of length m indicates the number of available resources of each type.

• **Allocation**: An n x m matrix defines the number of resources of each type currently allocated to each process.

• **Request**: An n x m matrix indicates the current request of each process. If Request [i, j] = k, then process P, is requesting k more instances of resource type Rj.

6.6.3 Detection-Algorithm Usage

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken.

6.7 RECOVERY FROM DEADLOCK

When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has spurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

6.7.1 Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

• **Abort all deadlocked processes**: This method clearly will break the dead – lock cycle, but at a great expense, since these

processes may have computed for a long time, and the results of these partial computations must be discarded, and probably must be recomputed.

• **Abort one process at a time until the deadlock cycle is eliminated:** This method incurs considerable overhead, since after each process is aborted a deadlock-detection algorithm must be invoked to determine whether a processes are still deadlocked.

6.7.2 Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until he deadlock cycle is broken.

The three issues are considered to recover from deadlock

1. **Selecting a victim**
2. **Rollback**
3. **Starvation**

6.8 SUMMARY

A deadlocked state occurs when two or more processes are waiting indefinitely for an event that can be caused only one of the waiting processes. There are three principal methods for dealing with deadlocks:

- Use some protocol to prevent or avoid deadlocks, entering that the system will never enter a deadlocked state.
- Allow the system to enter a deadlocked state, detect it, and then recover.
- Ignore the problem altogether and pretend that deadlocks never occur in the system.

Deadlock prevention is a set of methods for ensuring that at least one of the necessary condition cannot hold. Deadlock avoidance requires additional information about how resources are to be requested. Deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that a circular wait condition can never exist. Deadlock occur only when some process makes a request that cannot e granted immediately.

7

MEMORY MANAGEMENT

Unit Structure

7.0 OBJECTIVE

- To provide a detailed description of various ways of organizing memory hardware.
- To discuss various memory-management techniques, including paging and segmentation.
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging.

7.1 INTRODUCTION

Memory is central to the operation of a modern computer system. Memory is a large array of words or bytes, each with its own address.

A program resides on a disk as a binary executable file. The program must be brought into memory and placed within a process for it to be executed Depending on the memory management in use the process may be moved between disk and memory during its execution. The collection of processes on the disk that are waiting to be brought into memory for execution forms the input queue. i.e. selected one of the process in the input queue and to load that process into memory. We can provide protection by using two registers, usually a base and a limit, as shown in fig. 7.1. the base

register holds the smallest legal physical memory address; the limit register specifies the size of the range. For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

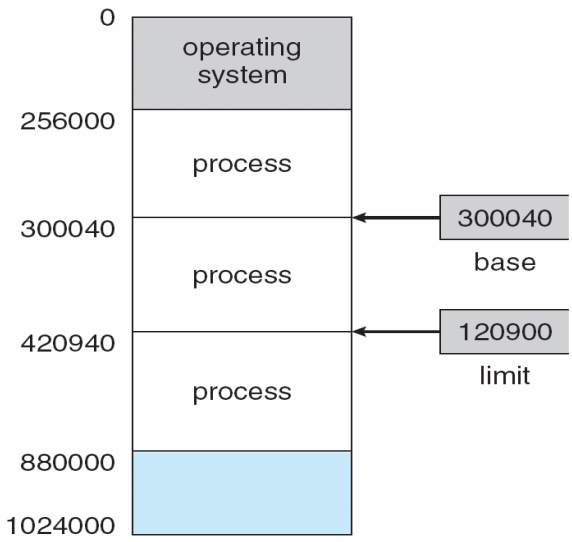


Fig 7.1 A base and limit register define a logical address space.

7.2 MEMORY PARTITIONING

The binding of instructions and data to memory addresses can be done at any step along the way:

- **Compile time:** If it is known at compile time where the process will reside in memory, then absolute code can be generated.

- **Load time:** If it is not known at compile time where the process will reside in memory, then the compiler must generate re-locatable code.

- **Execution time:** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

7.2.1 Dynamic Loading

Better memory-space utilization can be done by dynamic loading. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a re-locatable load format. The main program is loaded into memory and is executed.

The advantage of dynamic loading is that an unused routine is never loaded.

7.2.2 Dynamic Linking

Most operating systems support only static linking, in which system language libraries are treated like any other object module and are combined by the loader into the binary program image. The concept of dynamic linking is similar to that of dynamic loading. Rather than loading being postponed until execution time, linking is postponed. This feature is usually used with system libraries, such as language subroutine libraries. With dynamic linking, a stub is included in the image for each library-routine reference. This stub is a small piece of code that indicates how to locate the appropriate memory-resident library routing.

The entire program and data of a process must be in physical memory for the process to execute. The size of a process is limited to the size of physical memory. So that a process can be larger than the amount of memory allocated to it, a technique called overlays is sometimes used. The idea of overlays is to keep in memory only those instructions and data that are needed at any given time. When other instructions are needed, they are loaded into space that was occupied previously by instructions that are no longer needed.

Example, consider a two-pass assembler. During pass 1, it constructs a symbol table; then, during pass 2, it generates machine-language code. We may be able to partition such an assembler into pass 1 code, pass 2 code, the symbol table

1, and common support routines used by both pass 1 and pass 2.

Let us consider

Pass1	70K
Pass 2	80K
Symbol table	20K
Common routines	30K

To load everything at once, we would require 200K of memory. If only 150K is available, we cannot run our process. But

pass 1 and pass 2 do not need to be in memory at the same time. We thus define two overlays: Overlay A is the symbol table, common routines, and pass 1, and overlay B is the symbol table, common routines, and pass 2.

We add an overlay driver (10K) and start with overlay A in memory. When we finish pass 1, we jump to the overlay driver, which reads overlay B into memory, overwriting overlay A, and then transfers control to pass 2. Overlay A needs only 120K, whereas overlay B needs 130K.

As in dynamic loading, overlays do not require any special support from the operating system.

7.2.3 Logical versus Physical Address Space

An address generated by the CPU is commonly referred to as a logical address, whereas an address seen by the memory unit is commonly referred to as a physical address.

The compile-time and load-time address-binding schemes result in an environment where the logical and physical addresses are the same. The execution-time address-binding scheme results in an environment where the logical and physical addresses differ. In this case, we usually refer to the logical address as a virtual address. The set of all logical addresses generated by a program is referred to as a logical address space; the set of all physical addresses corresponding to these logical addresses is referred to as a physical address space.

The run-time mapping from virtual to physical addresses is done by the memory management unit (MMU), which is a hardware device.

The base register is called a relocation register. The value in the relocation register is added to every address generated by a user process at the time it is sent to memory. For example, if the base is at 13000, then an attempt by the user to address location 0 dynamically relocated to location 14,000; an access to location 347 is mapped to location 13347. The MS-DOS operating system running on the Intel 80x86 family of processors uses four relocation registers when loading and running processes.

The user program never sees the real physical addresses. The program can create a pointer to location 347 store it memory, manipulate it, compare it to other addresses all as the number 347.

The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addressed Logical addresses (in the range 0 to max) and physical addresses (in the range R + 0 to R + max for a base value R). The user generates only logical addresses.

The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

7.3 SWAPPING

A process, can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution. Assume a multiprogramming environment with a round robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been freed (Fig 7.2). When each process finishes its quantum, it will be swapped with another process.

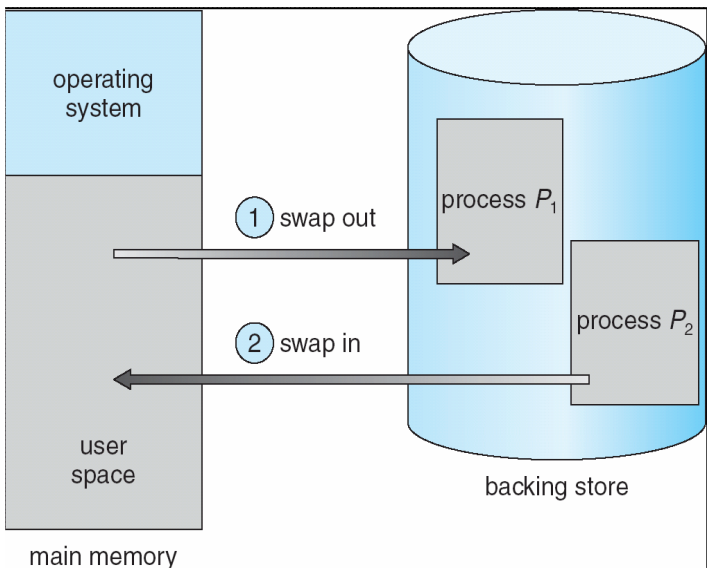


Fig 7.2 Swapping of two processes using a disk as a blocking store

A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process so that it can load and execute the higher priority process.

When the higher priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called rollout, roll in. A process is swapped out will be swapped back into the same memory space that it occupies previously. If binding is done at assembly or load time, then the process cannot be moved to different location. If execution-time binding is being used, then it is possible to swap a process into a different memory space.

Swapping requires a backing store. The backing store is commonly a fast disk. It is large enough to accommodate copies of all memory images for all users. The system maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run.

The context-switch time in such a swapping system is fairly high. Let us assume that the user process is of size 100K and the backing store is a standard hard disk with transfer rate of 1 megabyte per second. The actual transfer of the 100K process to or from memory takes

100K / 1000K per second = 1/10 second

= 100 milliseconds

7.4 CONTIGUOUS ALLOCATION

The main memory must accommodate both the operating system and the various user processes. The memory is usually divided into two partitions, one for the resident operating system, and one for the user processes.

To place the operating system in low memory. Thus, we shall discuss only me situation where the operating system resides in low memory (Figure 8.5). The development of the other situation is similar. Common Operating System is placed in low memory.

7.4.1 Single-Partition Allocation

If the operating system is residing in low memory, and the user processes are executing in high memory. And operating-system code and data are protected from changes by the user processes. We also need protect the user processes from one another. We can provide this 2 protection by using a relocation registers.

The relocation register contains the value of the smallest physical address; the limit register contains the range of logical

addresses (for example, relocation = 100,040 and limit = 74,600). With relocation and limit registers, each logical address must be less than the limit register; the MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory.

The relocation-register scheme provides an effective way to allow the operating system size to change dynamically.

7.4.2 Multiple-Partition Allocation

One of the simplest schemes for memory allocation is to divide memory into a number of fixed-sized partitions. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. When a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

The operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes, and is considered as one large block, of available memory, a hole. When a process arrives and needs memory, we search for a hole large enough for this process.

For example, assume that we have 2560K of memory available and a resident operating system of 400K. This situation leaves 2160K for user processes. FCFS job scheduling, we can immediately allocate memory to processes P1, P2, P3. Holes size 260K that cannot be used by any of the remaining processes in the input queue. Using a round-robin CPU-scheduling with a quantum of 1 time unit, process will terminate at time 14, releasing its memory.

Memory allocation is done using Round-Robin Sequence as shown in fig. When a process arrives and needs memory, we search this set for a hole that is large enough for this process. If the hole is too large, it is split into two: One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, we merge these adjacent holes to form one larger hole.

This procedure is a particular instance of the general dynamic storage-allocation problem, which is how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. The set of holes is searched to determine which

hole is best to allocate, first-fit, best-fit, and worst-fit are the most common strategies used to select a free hole from the set of available holes.

- First-fit: Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

- Best-fit: Allocate the smallest hole that is big enough. We must search the entire list, unless the list is kept ordered by size. This strategy-produces the smallest leftover hole.
- Worst-fit: Allocate the largest hole. Again, we must search the entire list unless it is sorted by size. This strategy produces the largest leftover hole which may be more useful than the smaller leftover hole from a best-t approach.

7.4.3 External and Internal Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when enough to the memory space exists to satisfy a request, but it is not contiguous; storage is fragmented into a large number of small holes.

Depending on the total amount of memory storage and the average process size, external fragmentation may be either a minor or a major problem.

Given N allocated blocks, another 0.5N blocks will be lost due to fragmentation. That is, one-third of memory may be unusable. This property is known as the 50- percent rule.

Internal fragmentation - memory that is internal to partition, but is not being used.

7.5 PAGING

External fragmentation is avoided by using paging. In this physical memory is broken into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames. Every address generated by the CPU is divided into any two parts: a page number(p) and a page offset(d) (Fig 7.3). The page number is used as an index into a page table. The page table contains the base address of each page

in physical memory. This base address is combined with the gage offset to define the physical memory address that is sent to the memory unit.

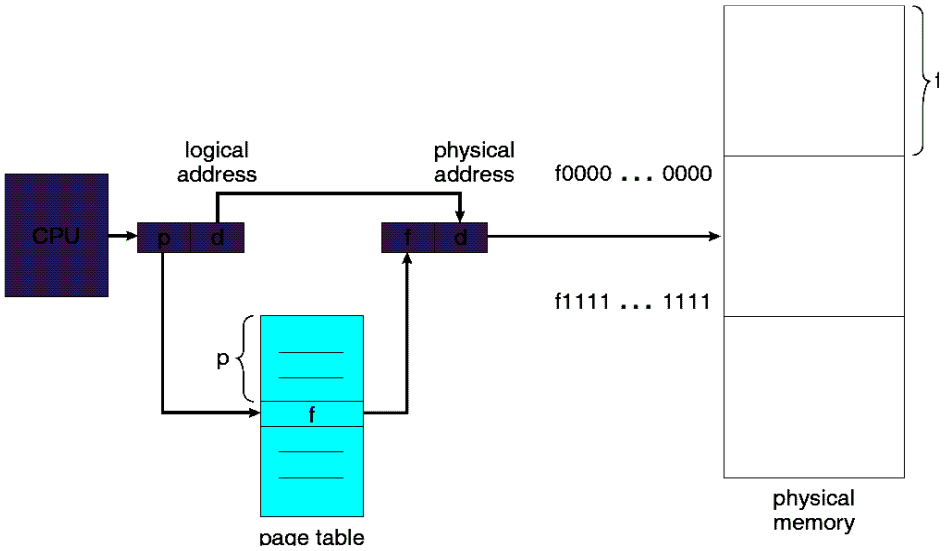


Fig 7.3 Paging Hardware

The page size like is defined by the hardware. The size of a page is typically a power of 2 varying between 512 bytes and 8192 bytes per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset. If the size of logical address space is 2^m, and a page size is 2^n addressing units (bytes or words), then the high-order m - n bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:

page number

page	offset
p	d
m – n	n

where p is an index into the page table and d is the displacement within the page.

Paging is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address.

When we use a paging scheme, we have no external fragmentation: Any free frame can be allocated to a process that needs it.

If process size is independent of page size, we can have internal fragmentation to average one-half page per process.

When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires n pages, there must be at least n frames available in memory. If there are n frames available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames and the frame number is put in the page table for this process. The next page is loaded into another frame, and its frame number is put into the page table, and so on.

The user program views that memory as one single contiguous space, containing only this one program. But the user program is scattered throughout physical memory and logical addresses are translated into physical addresses.

The operating system is managing physical memory, it must be aware of the allocation details of physical memory: which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a data structure called a frame table. The frame table has one entry for each physical page frame, indicating whether the latter is free allocated and, if it is allocated, to which page of which process or processes.

The operating system maintains a copy of the page table for each process. Paging therefore increases the context-switch time.

7.6 SEGMENTATION

A user program can be subdivided using segmentation, in which the program and its associated data are divided into a number of **segments.** It is not required that all segments of all programs be of the same length, although there is a maximum segment length. As with paging, a logical address using segmentation consists of two parts, in this case a segment number and an offset.

8

VIRTUAL MEMORY

Unit Structure

8.0 OBJECTIVE

- To describe the benefits of a virtual memory system.
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames.
- To discuss the principle of the working-set model.

8.1 VIRTUAL MEMORY

- Virtual memory is a technique that allows the execution of process that may not be completely in memory. The main visible advantage of this scheme is that programs can be larger than physical memory.

- Virtual memory is the separation of user logical memory from physical memory this separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Fig 8.1).

- Following are the situations, when entire program is not required to load fully.

1. User written error handling routines are used only when an error occurs in the data or computation.

2. Certain options and features of a program may be used rarely.
3. Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.

- The ability to execute a program that is only partially in memory would counter many benefits.

1. Less number of I/O would be needed to load or swap each user program into memory.
2. A program would no longer be constrained by the amount of physical memory that is available.
3. Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

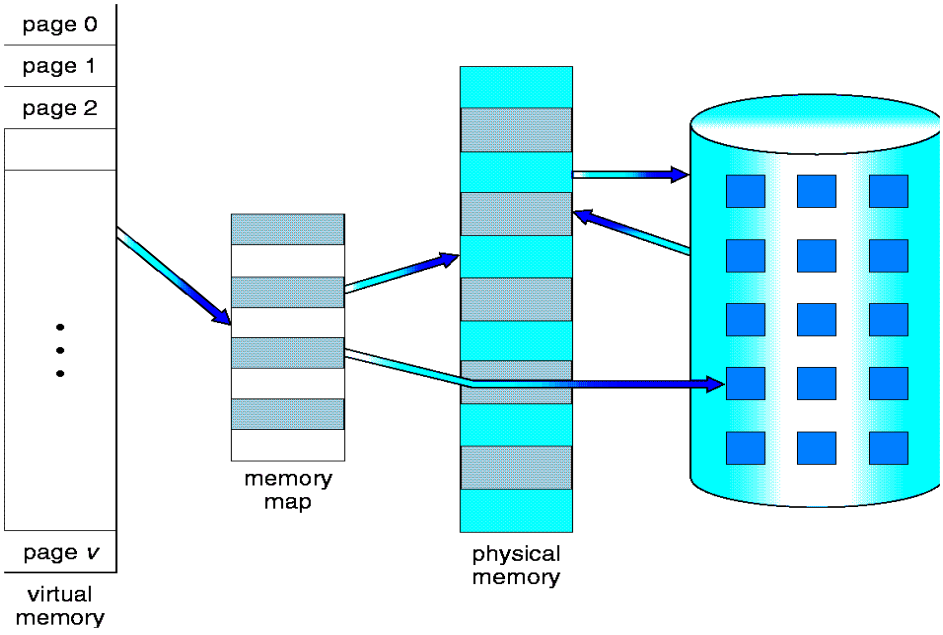


Fig. 8.1 Diagram showing virtual memory that is larger than physical memory.

Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.

8.2 DEMAND PAGING

A demand paging is similar to a paging system with swapping(Fig 8.2). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory.

When a process is to be swapped in, the pager guesses which pages will be

used before the process is swapped out again Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used in anyway, decreasing the swap time and the amount of physical memory needed.

Hardware support is required to distinguish between those pages that are in memory and those pages that are on the disk using the valid-invalid bit scheme. Where valid and invalid pages can be checked checking the bit and marking a page will have no effect if the process never attempts to access the pages. While the process executes and accesses pages that are memory resident, execution proceeds normally.

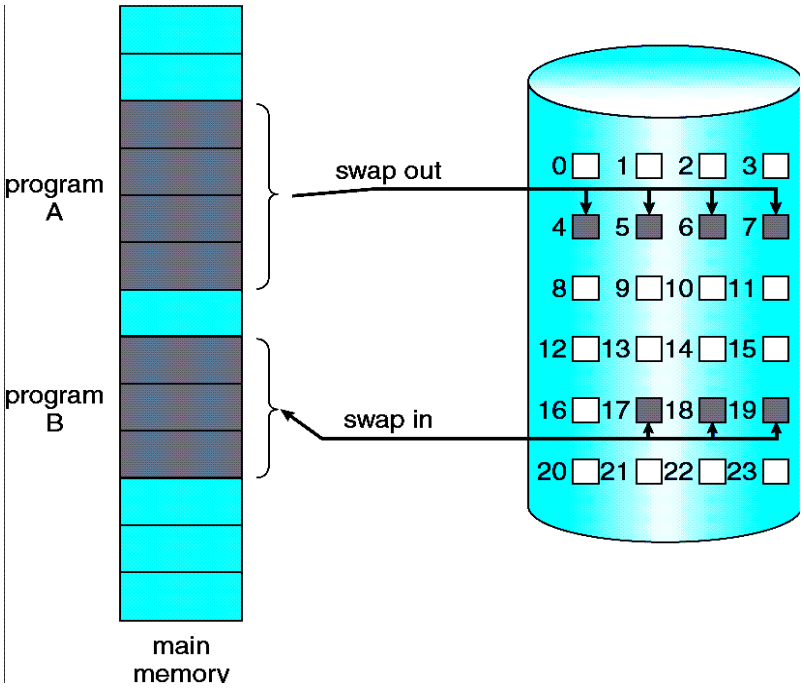


Fig. 8.2 Transfer of a paged memory to continuous disk space

Access to a page marked invalid causes a page-fault trap. This trap is the result of the operating system's failure to bring the desired page into memory. But page fault can be handled as following (Fig 8.3):

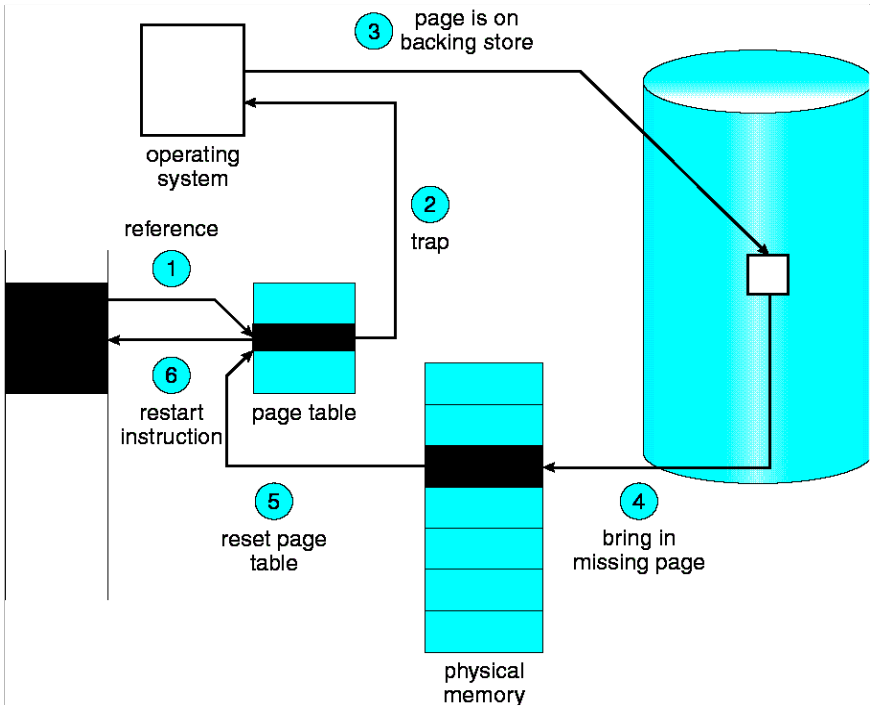


Fig. 8.3 Steps in handling a page fault

1. We check an internal table for this process to determine whether the reference was a valid or invalid memory access.

2. If the reference was invalid, we terminate the process. If .it was valid, but we have not yet brought in that page, we now page in the latter.

3. We find a free frame.

4. We schedule a disk operation to read the desired page into the newly allocated frame.

5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.

6. We restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been memory.

Therefore, the operating system reads the desired page into memory and restarts the process as though the page had always been in memory.

The page replacement is used to make the frame free if they are not in used. If no frame is free then other process is called in.

8.2.1 Advantages of Demand Paging:

1. Large virtual memory.
2. More efficient use of memory.
3. Unconstrained multiprogramming. There is no limit on degree of multiprogramming.

8.2.2 Disadvantages of Demand Paging:

1. Number of tables and amount of processor over head for handling page interrupts are greater than in the case of the simple paged management techniques.
2. due to the lack of an explicit constraints on a jobs address space size.

8.3 PAGE REPLACEMENT ALGORITHM

There are many different page replacement algorithms. We evaluate an algorithm by running it on a particular string of memory reference and computing the number of page faults. The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference. The latter choice produces a large number of data.

1. For a given page size we need to consider only the page number, not the

entire address.

2. if we have a reference to a page p, then any immediately following references to page p will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.

Eg:- consider the address sequence

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0104, 0101, 0609, 0102, 0105 and reduce to 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 frames available increase, the number of page faults

will decrease.

8.3.1 FIFO Algorithm

The simplest page-replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. We can create a FIFO queue to hold all pages in memory.

The first three references (7, 0, 1) cause page faults, and are brought into these empty eg. 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1 and consider 3 frames. This replacement means that the next reference to 0 will fault. Page 1 is then replaced by page 0.

8.3.2 Optimal Algorithm

An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal page-replacement algorithm exists, and has been called OPT or MIN. It is simply Replace the page that will not be used for the longest period of time.

Now consider the same string with 3 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 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. Optimal replacement is much better than a FIFO.

The optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

8.3.3 LRU Algorithm

The FIFO algorithm uses the time when a page was brought into memory; the OPT algorithm uses the time when a page is to be

used. In LRU replace the page that has not been used for the longest period of time.

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time.

Let S^R be the reverse of a reference string S, then the page-fault rate for the OPT algorithm on S is the same as the page-fault rate for the OPT algorithm on S^R.

8.3.4 LRU Approximation Algorithms

Some systems provide no hardware support, and other page-replacement algorithm. Many systems provide some help, however, in the form of a reference bit. The reference bit for a page is set, by the hardware, whenever that page is referenced. Reference bits are associated with each entry in the page table Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware.

8.3.4.1 Additional-Reference-Bits Algorithm

The operating system shifts the reference bit for each page into the high-order or of its 8-bit byte, shifting the other bits right 1 bit, discarding the low-order bit.

These 8-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, then the page has not been used for eight time periods; a page that is used at least once each period would have a shift register value of 11111111.

8.3.4.2 Second-Chance Algorithm

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page gets a second chance, its reference bit is cleared and its arrival e is reset to the current time.

8.3.4.3 Enhanced Second-Chance Algorithm

The second-chance algorithm described above can be enhanced by considering troth the reference bit and the modify bit as an ordered pair.

1. (0,0) neither recently used nor modified best page to replace.

The Linux kernel is implemented as a traditional monolithic kernel for performance reasons, but it is modular enough in design to allow most drivers to be dynamically loaded and unloaded at run time.

15.6 MODEL QUESTION

Q.1 List different components of a linux system?

Q.2 What are Kernel modules?

Q.3 Explain Basic architecture of UNIX/Linux system?

Q.4 Explain basic features of UNIX/Linux?

Q.5 Define and explain Shell?