
Operating Systems

(605364)

Assist. Prof. Dhia Alzubaydi

*Faculty of Science and IT
Al-Isra Private University
Amman, Jordan
2007-2008*

Chapter 1: Introduction to Operating Systems

1-1 Definition of Operating System (OS)

Os is a set of programs that controls effectively the computer resources and makes them conveniently available to users i.e easy to use.

Os is rather complicated software and hence designed usually by professional software companies and sold with computer system as part of it. During computer operation, some basic OS programs (Called Os Core or Kernel) are resident in main memory while others are stored on hard disk and loaded into memory when needed.

1-2 Functions of OS

The functions can be summarized as follows (will be explained later in more details):

- 1- Management of computer resources (processors, memory, disks, I/O devices, programs, etc.).
- 2- Scheduling resources among users (time sharing).
- 3- Protection of programs being executed in memory from one another.
- 4- Providing a proper user interface e.g Graphics User Interface (GUI).
- 5- File management.
- 6- Network communication.
- 7- Many others.

1-3 History of OS

This history can be summarized as follows:

- 1- **NO OS:** In 1940s, computers were simple; programs and data were entered via mechanical switches.
- 2- **Single programming batch processing Os :** In 1950s, operating systems were introduced and allowed users to submit their data and programs as groups of punched cards called " Batches". At that time, one job (program) was allowed to be executed until completed.
- 3- **Multiprogramming batch processing Os:** In early 1960s, OS allowed several jobs to be executed on time sharing basis. This technique was called " multiprogramming" and was introduced to make effective use of computer resources.
- 4- **MultiUser OS:** In late 1960s, Users were able to make interactive communication to computer via their terminals. The OS introduced at that time allowed each user to develop his programs independent of others which means that computer resources were time shared among users.
- 5- **Network OS :** In 1970s, computers were able to communicate via a proper communication network. The OS was also developed to make effective use of network resources.

-
- 6- **Disk Operating System (DOS):** In 1980S, personal computers (PCs) were introduced and proper operating systems were developed for such computers. These OSs were stored on disks.
 - 7- **Windows Operating System :** In 1990s, OSs were developed to allow GUI to users which encouraged non specialized persons to use computers and buy them.
 - 8- **Present OSs :** Nowadays, OSs are quite advanced and allow multithreading, multiprogramming, multiprocessing, virtual organization, client/server application, etc.

1-3 OS Examples

MS-DOS: Single user, single programming, no GUI.

Windows: Single user, multiprogramming, with GUI.

Unix : Multi user, multiprogramming.

End of ch1

Chapter 2: Basic Concepts of Computer System

2-1 Introduction

Studying basic concepts of computer system is essential for understanding OS concepts.

2-2 Block Diagram of Simple Computer System

A simple computer system is shown in fig 2.1. The components of this diagram can be explained as follows:

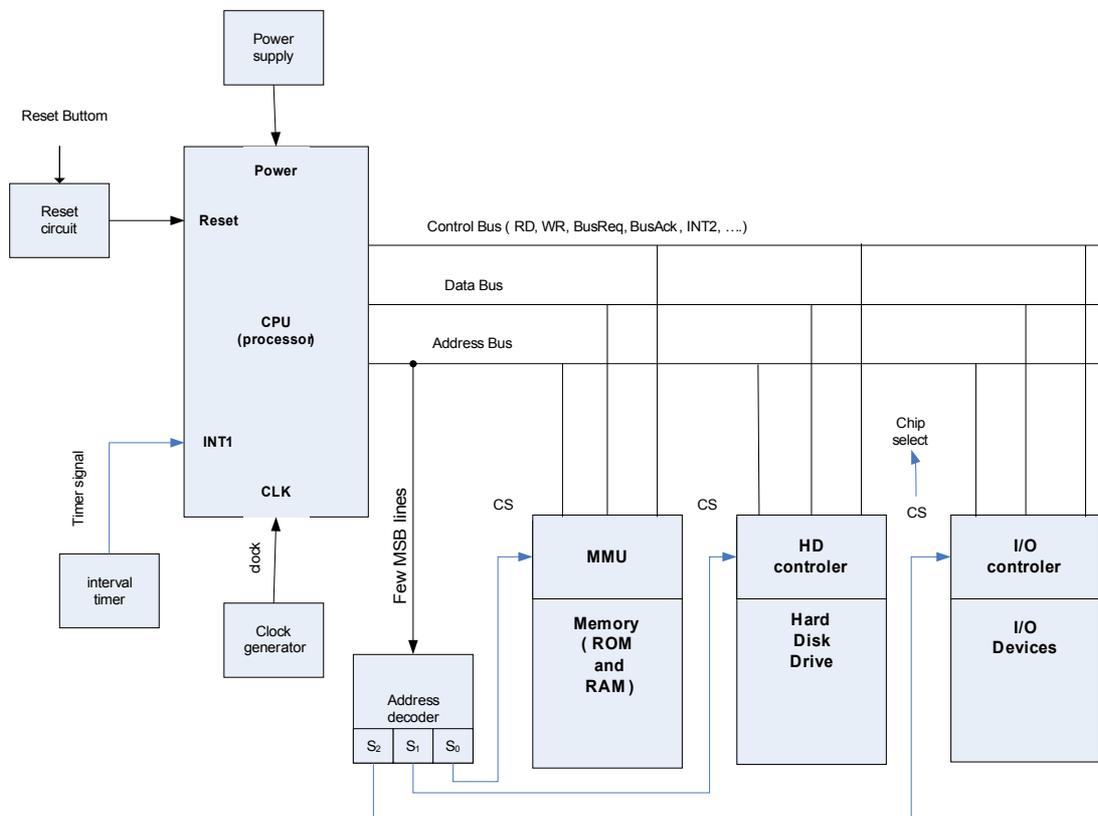


Fig 2.1 Simple Computer System.

-
- 1- **CPU**: Central processing Unit is the main unit of any computer system. The CPU consists of Arithmetic Logic Unit (ALU) and Control Unit (CU).the main registers included in CPU are:
- **Sp** : Stack Pointer Register which holds an address pointing at a location in memory called " Stack Top".
 - **PC** : Program Counter Register which holds address of the next instruction to be executed.
 - **A** : Accumulator Register which holds the result of any arithmetic or logic operation.
 - **PSW** : Processor Status Word Register which holds flags showing the status of any arithmetic or logic operation (Zero, overflow, carry, etc).
 - **Data Registers** (B, C, D,E) which act as fast storage for temporary data.
- 2- **Clock Generator** : it is essential electronic circuit generating periodic pulse signal called "clock" as shown in fig 2.2. It is worth remembering that all actions in a computer system are timed precisely and synchronized with clock edges. The clock frequency ($1/T$) determines the computer speed to a large degree in addition to other factors.

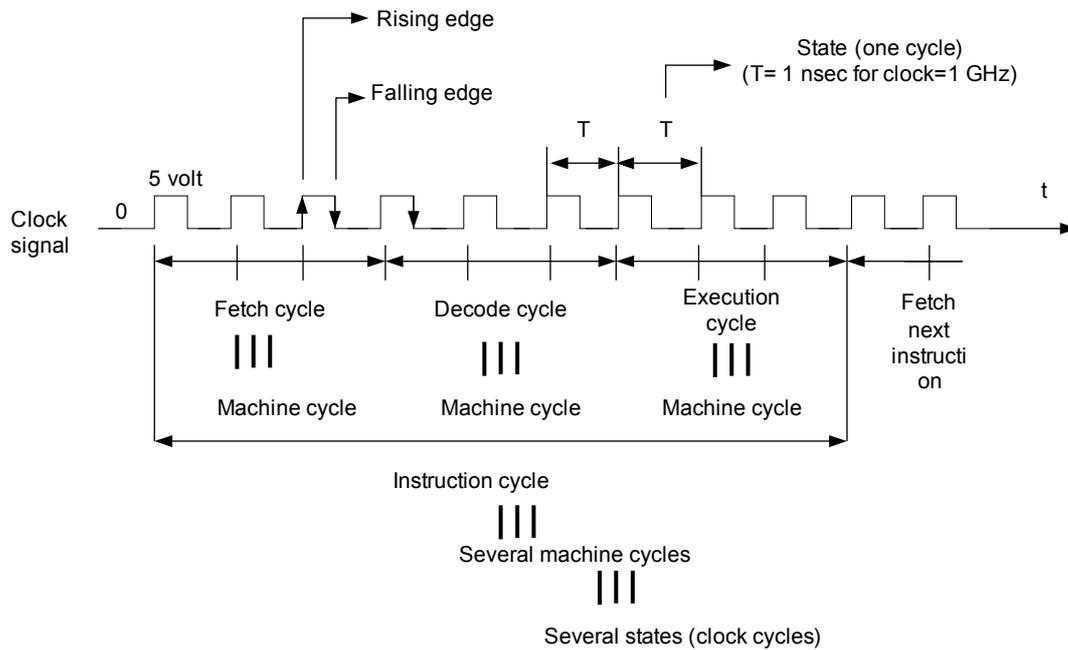


Fig 2.2 Instruction Cycle

- 3- **Reset Circuit** : when a reset button is pressed, a pulse signal is issued to CPU which forces a value of Zeros to PC and an instruction fetch cycle is started from a location in memory pointed at by pc. This means that "Reset" forces computer to start execution of a program loaded at address zero (Usually BIOS program stored in Read Only Memory ROM).
- 4- **Interval Timer** : It generates a periodic pulse but much slower than clock generator. Each Timer period is called "Time Slice" and equivalent to millions of clock periods (states). This means that during a Time Slice, it is possible to execute part of a program. The interval timer, as will be shown later, has a big role in multiprogramming system.

- 5- **Power Supply** : It is necessary for CPU and all other components.
- 6- **Memory** : It is the second important unit in any computer system (after CPU). It is very fast addressable storage. It is used to hold programs when they are being executed in addition to other functions (e.g stack,...) there are two types of memory:

- Random Access Memory (RAM) used for storing programs and data temporarily.
- Read Only Memory (ROM) used to hold programs and data permanently (e.g BIOS).

The memory can be visined as shown in fig 2.3 i.e set of addressable locations that may hold data.

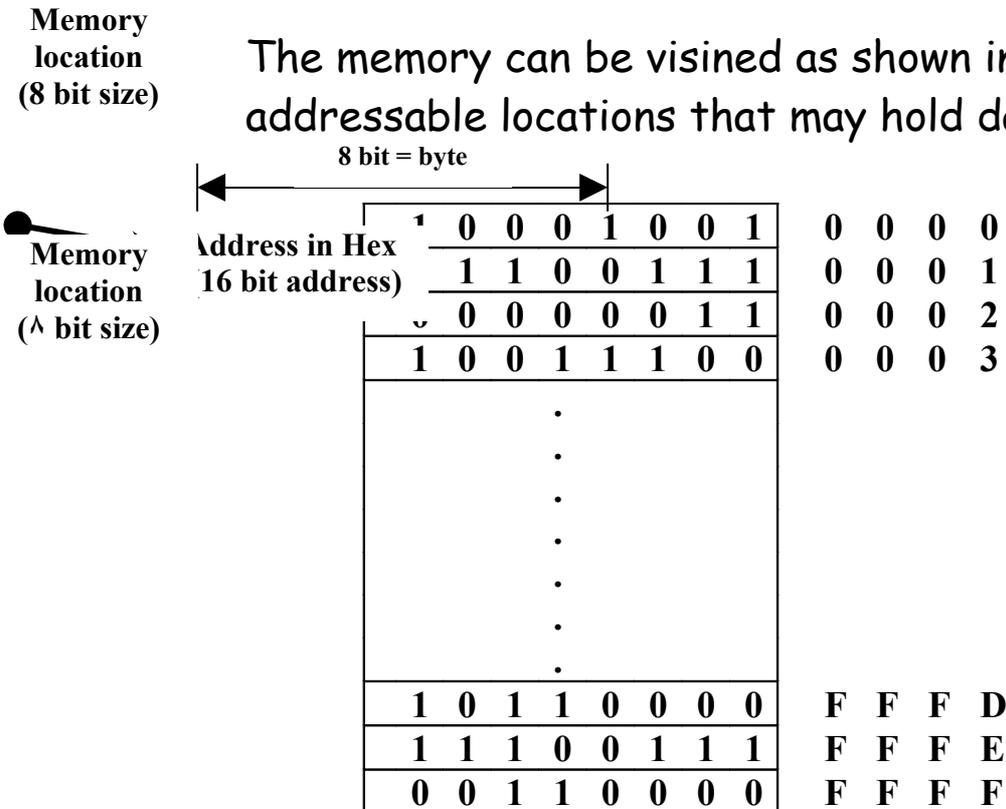


Fig 2.3 Memory of 64 K bytes.

-
- 7- **Hard Disk Drive** : It is a large volume of permanent storage for programs and data but it is not addressable and of slow access as well.
 - 8- **I/O Devices** : the computer is not useful unless interfaced to external world such as printers, keyboards, displays, scanners, mouse, camera, etc.
 - 9- **Controllers** : To connect any device to CPU, it is necessary to use a proper interface circuit (card) called "controller". The controller is usually driven by a proper program called "Driver". This means that "Controller" is a hardware (H/W) while "driver" is a software (S/W). Here, it is worth noting that a "Drive" is different from "Driver" as it represents the instrument itself e.g disk drive which consists of disk, motor, electronic circuits, heads, etc. Also it is worth noting that a memory controller is usually called as Memory Management Unit (MMU).
 - 10- **Address Decoder** : Few of most significant address signals are decoded to enable selection of one device that should respond to CPU.
 - 11- **Address Bus** : Set of copper lines carrying electrical signals (voltages) which are used to select one location only in the whole computer system whether for "Read" or "Write" cycles.
 - 12- **Data Bus** : several lines carrying data to or from addressed location.

13- **Control Bus** : Several Lines having several functions. One line, for example, is used to carry "Read" signal to declare that a cycle is a "Read" one.

14- **Interrupt (INT)** : It is input signal to CPU which when "active" forces certain value (address) to PC and initiates instruction cycle and hence the CPU starts executing " Interrupt Subroutine".

2-3 I/O Data Transfer

There are several methods for transferring data between I/O devices and memory as follows:

- Programmed I/O.
- DMA I/O.

2-3-1 Programmed I/O

There are two types:

- Polling I/O
- Interrupt I/O

In polling, CPU executes a program to scan I/O device periodically to check its need for service. In interrupt I/O, there is no scan at all, however, when I/O device requires service, it activates an interrupt signal to CPU and hence interrupt subroutine will be executed which should provide the requested service.

2-3-2 Direct Memory Access (DMA)

There are two useful control signals for this operation which are: BusReq and BusAck.

BusReq is an input signal to CPU and when activated, it forces CPU to separate itself from its external "Buses" at the end of machine cycle. When separation occurs, CPU activates BusAck signal to inform I/O devices that they can use all the Buses.

Now, it is possible to explain DMA as follows:-

- 1- I/O device activates BusReq line and wait till BusAck is activated.
- 2- When BusAck is activated, I/O device can master the Buses and hence use them to address memory and make data transfer between Memory and I/O device.

From the above, it is clear that programmed I/O is implemented by making CPU executes a proper program while in DMA, the CPU does not interfere as it separates it self from its Buses.

2-4 CPU Modes of Operation (States)

There are two main modes:

- **Supervisor Mode** : In this mode, the CPU can execute all instructions including " user "and" privileged" instructions.
- **User Mode** : In this mode, the CPU can execute " user" instructions only and can not execute privileged instructions.

The CPU modes are useful for programs protection in multi programming environment.

2-5 Bootstrapping

When a computer is powered up , or Reset activated, BIOS is started and loads " Boot Sector" from disk (Hard, CD, floppy). The Boot Sector contains a program which enables loading of main OS components into memory which then started. the above operations are called "Bootstrapping" which aims at loading OS into memory and running it.

2-6 Types of Interrupts

Interrupt means forcing CPU to change its mode to supervisor and make subroutine call at any time of program execution. The causes of interrupts may be external or internal and hence we have the following types:

-
- 1- **Hardware Interrupt** : This is caused by activating interrupt input to CPU by an I/O device or interval timer.
 - 2- **Software interrupt** : This has two types :
 - a- **Exception** : this occurs due to any of the following events:
 - Divide by zero (overflow)
 - Address violation as will be explained later.
 - Page fault as will be explained later.
 - Others.
 - b- **System Call** : These are normal user instructions but have the effect of interrupt which is basically changing CPU mode from " User" to "Supervisor" and making subroutine call. The call does not cause address violation as the CPU becomes in a supervisor mode.

2-7 Application Programming Interfaces (APIs)

APIs are set of subroutines that control computer resources and considered as part of OS. A user program (application) can use these APIs by calling them via special instructions called "System Calls". A system call acts as a " software interrupt" and hence changes CPU mode from " User" to " Supervisor" which in turn prevents an " address violation exception" from occurring.

Here, it should be noted that if user program tries to use normal subroutine call instruction then an address violation exception will occur as the called address is outside user program " address space" as will be explained later.

2-8 Multiprocessing

It is equivalent to " multiprocessors" and not to " multi processes". Multiprocessing means that CPU consists of several processors sharing memory and controlled by single OS for the purpose of speeding up computer operation.

2-9 Buffering

A " Buffer" is an area of memory for holding data temporarily during data transfer between running program and I/O device. This technique speeds up the computer operation. Examples of these buffers are:

a- Hard Disk Buffer :

In this case, OS writes data quickly to disk buffer (i.e to memory) and later on data will be transferred to the slow disk drive. When reading, the reverse occurs.

b- Keyboard buffer:

It is used to hold characters typed by a slow user and later the running program will process these characters.

2-10 Spooling

It is , also, a technique for speeding up computer operation. In this case, the data to be written to a very slow device (e.g printer) are written, first, to intermediate medium speed device (such as disk) and later on transferred to the slow device.

2-11 Structured Program Development Cycle

This cycle is shown in fig 2.4. In this figure, we notice

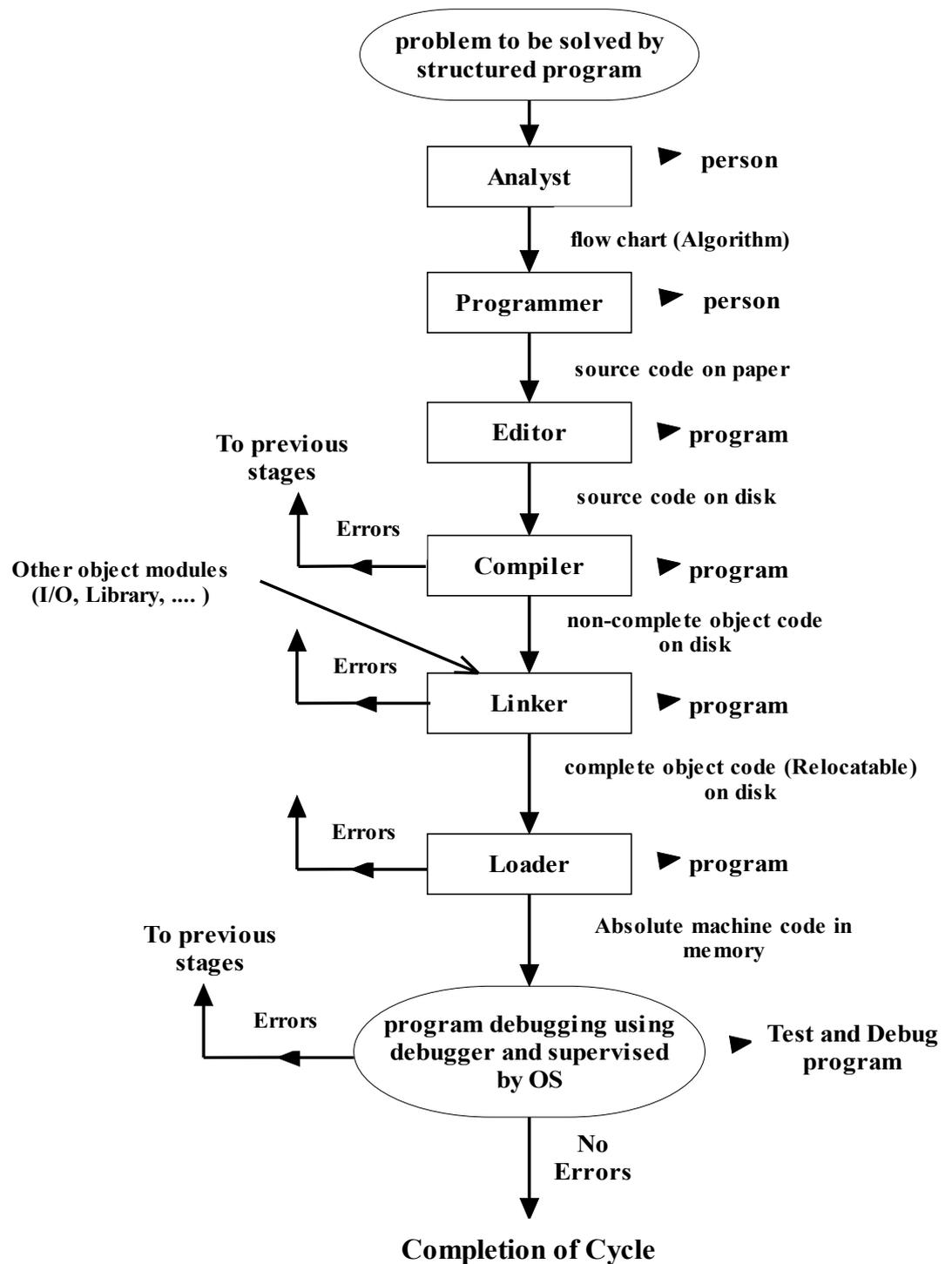


Fig 2.4 structured Program Development

The followings:

- 1- Object code is non-executable machine code as it is stored on disk and its addresses are relocatable i.e have to be modified when loading into memory for execution.
- 2- Absolute machine code is executable code as it is loaded into memory and its addresses have been modified accordingly.

2-12 Basic Property of Absolute Machine Code

The basic property can be stated as follows:

"Absolute Machine Code" can not be displaced in memory, in other words, if this code is moved to another area of memory then it will not be executed correctly.

The reason behind this property is the addresses used in the code as shown in fig 2.5. In this figure, the symbol "START" is replaced after compilation, linking, and loading by the address "00010000". If we move this code to

Address	Machine Code	Mnemonic
0 0 0 1 0 0 0 0	1 1 0 0 0 1 1 1	START: MOVE R1, R2
0 0 0 1 0 0 0 1	1 0 0 0 0 1 0 0	ADD R3, R4
0 0 0 1 0 0 1 0	0 0 1 1 0 0 1 1	JC START
0 0 0 1 0 0 1 1	0 0 0 1 0 0 0 0	ADD R1, R3
·	·	·
·	·	·
·	·	·
·	·	·
·	·	·
·	·	·

Fig 2.5 Absolute machine Code Program

A memory area starting at address 10000000, then it will not be executed correctly because of "START" value as shown in fig 2.6.

Address								Machine Code							Action	
1	0	0	0	0	0	0	0	1	1	0	0	0	1	1	1	Jump operation will be to address (00010000) while it should be to (1000000) in order to get correct execution
1	0	0	0	0	0	0	1	1	0	0	0	0	1	0	0	
1	0	0	0	0	0	1	0	0	0	1	1	0	0	1	1	
1	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	
1	0	0	0	0	1	0	0	1	0	0	0	0	1	1	1	
.	

Fig 2.6 Displaced Absolute Machine Code Program (Execution will not be correct).

2.13 Useful Terms

- 1- **single programming** : One program is executed by CPU until completed.
- 2- **Multiprogramming** : Several programs are executed by single CPU on timesharing basis.
- 3- **Single User** : One user is allowed to use a computer at any one time. The OS can be single programming or multiprogramming.
- 4- **Multi User** : Several users are allowed to use a computer at any time. The OS must be multi programming.
- 5- **Multiprocessing** : The CPU consists of several processors sharing same memory and controlled by one OS.

-
- 6- **Sequential programming** : A program consists of one stream (thread) of instructions to be executed sequentially.
 - 7- **Parallel programming** : A program may have some parallel streams (threads) that can be executed concurrently.
 - 8- **Task** : Single thread or sequential program being executed.
 - 9- **Multitasking** : Several tasks may be executed either on timesharing basis (case of single processor CPU) or in parallel (case of multiprocessing). Parallel programming will be explained later.

10- Multiprocesses :It is the same as multiprogramming and not as multiprocessors.

11-Interactive Computing : The user is directly connected to computer and works " On- Line".

12- Batch Computing: The user is off line and not connected to computer but submits his job as a batch of cards or on other media (paper tape, disk, magnetic tape,...)

13- Real Time : The computer is connected to a factory and commanding it in real time using input and output signals.

End of

ch2

Ch3 Process Concepts

3-1 Introduction

Process concept is necessary for multiprogramming implementation. In multiprogramming, a CPU has to be time shared between several programs and hence extra work (overhead) is necessary from the side of OS to achieve the timesharing operation. The overhead includes process creation, context switching, process state transitions, etc.

3-2 Motivations for Multiprogramming

In single programming, each job (program) has to be completed before starting new one. We all know that any program is actually useless without I/O activities. These activities are usually very slow compared to CPU activity which means that a lot of CPU time will be wasted in waiting I/O requests to be completed as shown in fig.3.1. To solve this problem, a multiprogramming concept was introduced. This introduction created many new problems to be solved by OS such as processor scheduling, memory organization, memory management, protection, and actually all topics to be studied in this course.

The CPU utilization in multiprogramming is high as there is no waiting and overhead is comparatively small as shown in fig 3.2.

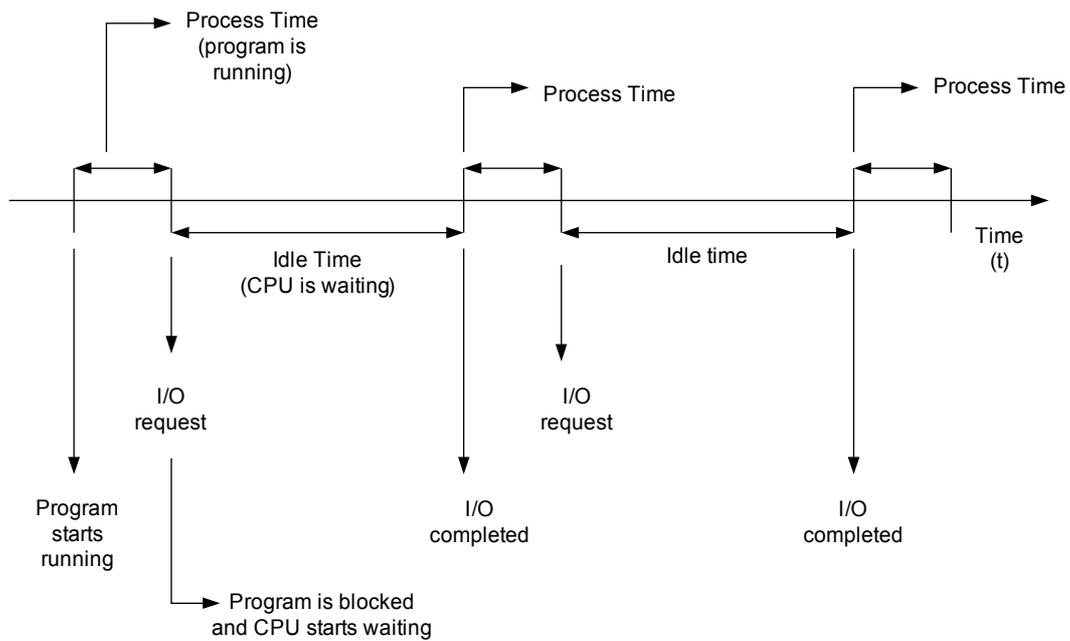


Fig 3.1 CPU Utilization in Single Programming (Idle Time \gg Process Time)

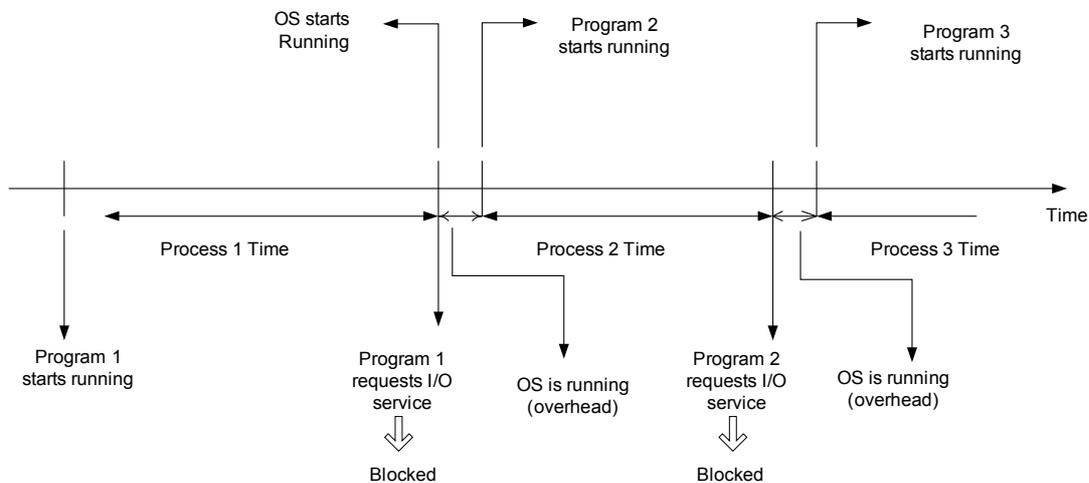


Fig 3.2 CPU Utilization in Simplified Multiprogramming Case. (Overhead \gg Process Time)

In fig 3.2, the diagram is simplified to show the main idea, however, multiprogramming is more complicated as will be shown later.

3-3 Definition of a Process

A process is a program in execution, described by a Process Control Block (PCB), and has several states (Ready, Running, Blocked, etc.).

From this definition, we can conclude the main differences between program and process as shown in fig 3.3 (PCB will be explained be later).

Program	Process
Stored on disk	Resident in memory
Addresses are relocatable	Addresses are absolute
Can be moved to another area on disk	Can't be displaced to another area in memory
Has one state which is " non-executable"	Has several execution states (Ready, Running,....)
Does not have PCB	Has PCB
Can't be allocated CPU time	Allocated CPU time
Contains object code	Contains absolute machine code
Program image includes object code only	Process image includes machine code, data, and stack
Occupies smaller area on disk (image size is smaller)	Occupies larger area in memory (image size is larger)

Fig 3.3 Comparison of Program and Process.

3-4 Basic Process State Diagram

This diagram is shown in fig 3.4 where we notice the followings:

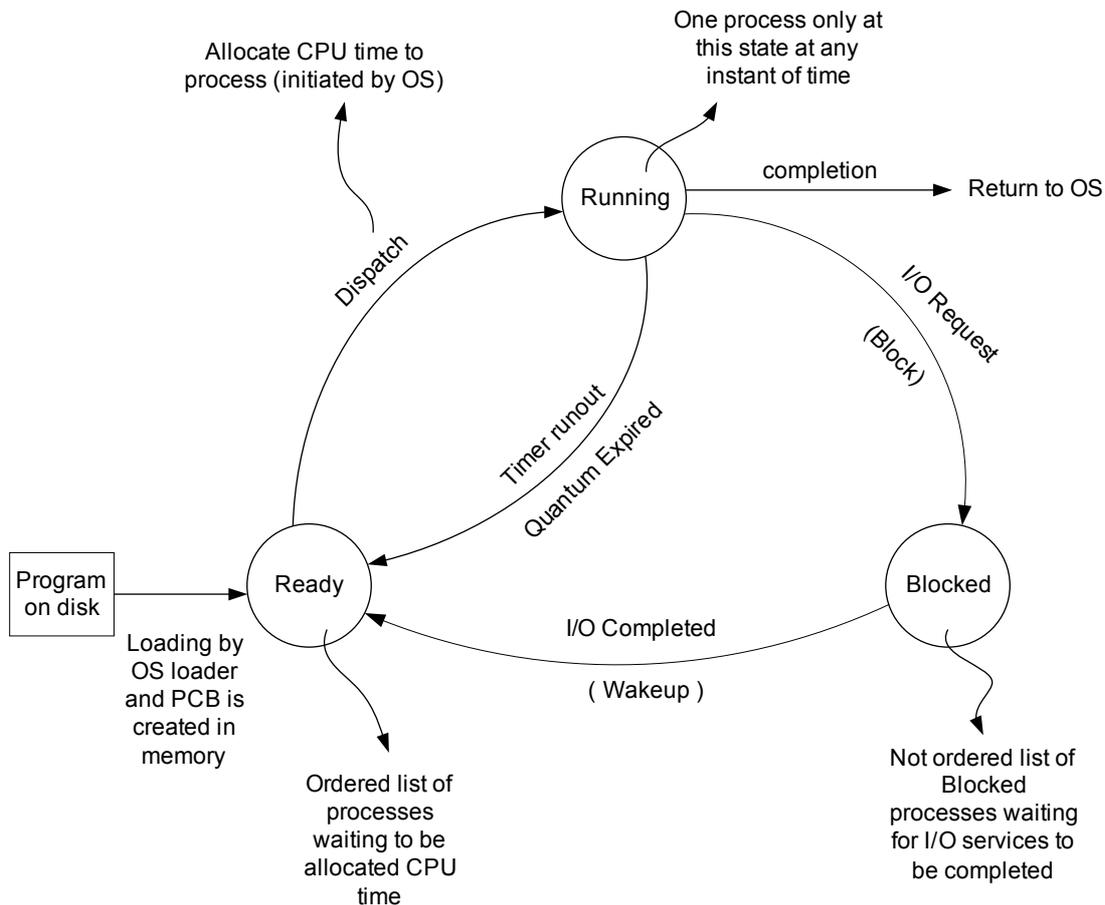


Fig 3.4 Basic Process State Diagram

1. At any instant of time, there is only one process running i.e allocated CPU time.
2. Exit from Running state may occur as a result of any of following events:
 - Completion of process.
 - Request of I/O service by a process.

-
- Time slice determined by interval timer has expired and hence an interrupt is activated which forces CPU to run OS instructions.
3. The transfer from Ready to Running state (dispatch) is carried out by OS according to certain criteria as will be shown later when studying " Processor Scheduling".
 4. The term " Execution " means generally, " Ready", " Running", or " Blocked".

3-5 Process Control Block (PCB)

PCB is a data structure describing a process and resident in memory as shown in fig 3.5.

Each process has its own PCB in memory and assigned a process Identification Number (PID).

The locations of PCBs is kept in a special table called " Process Table" which is resident in memory and used by OS.

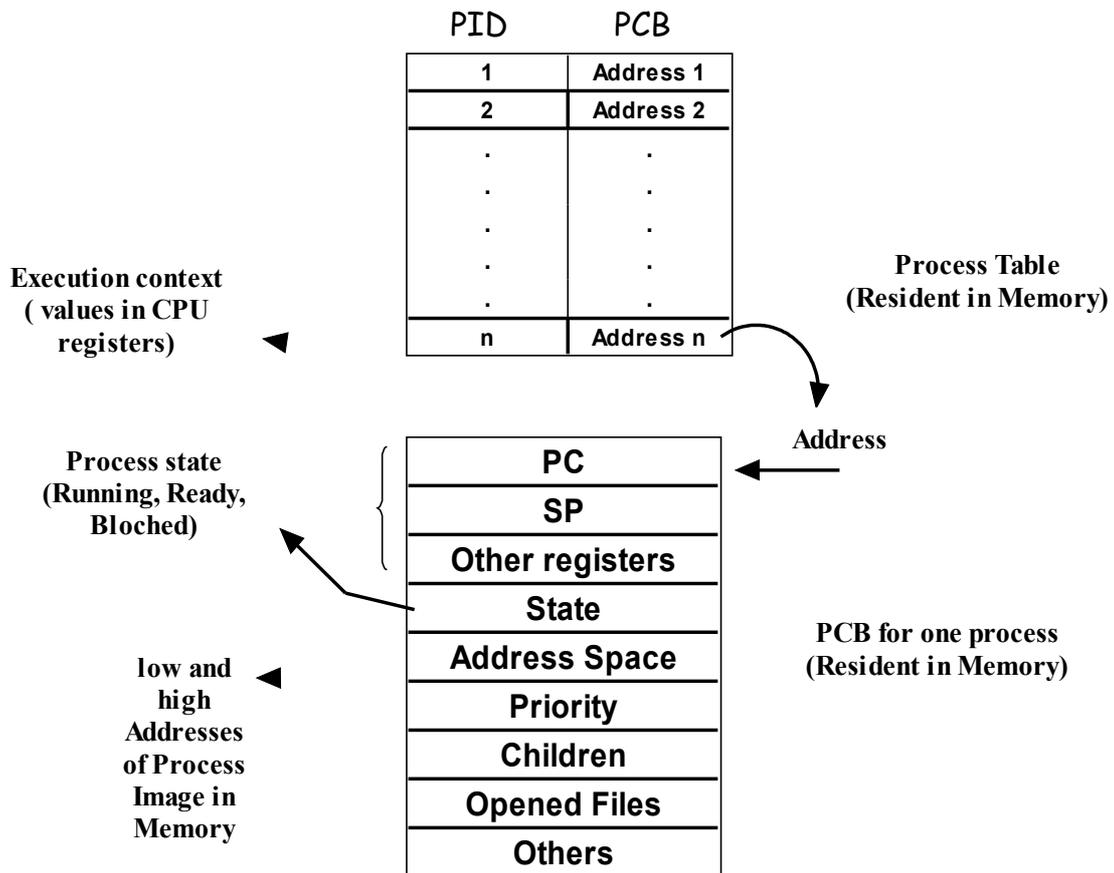


Fig 3.5 Process Table and PCB

Now, it is very useful to show the different components resident in memory in a from called " Memory Map" as shown in fig 3.6.

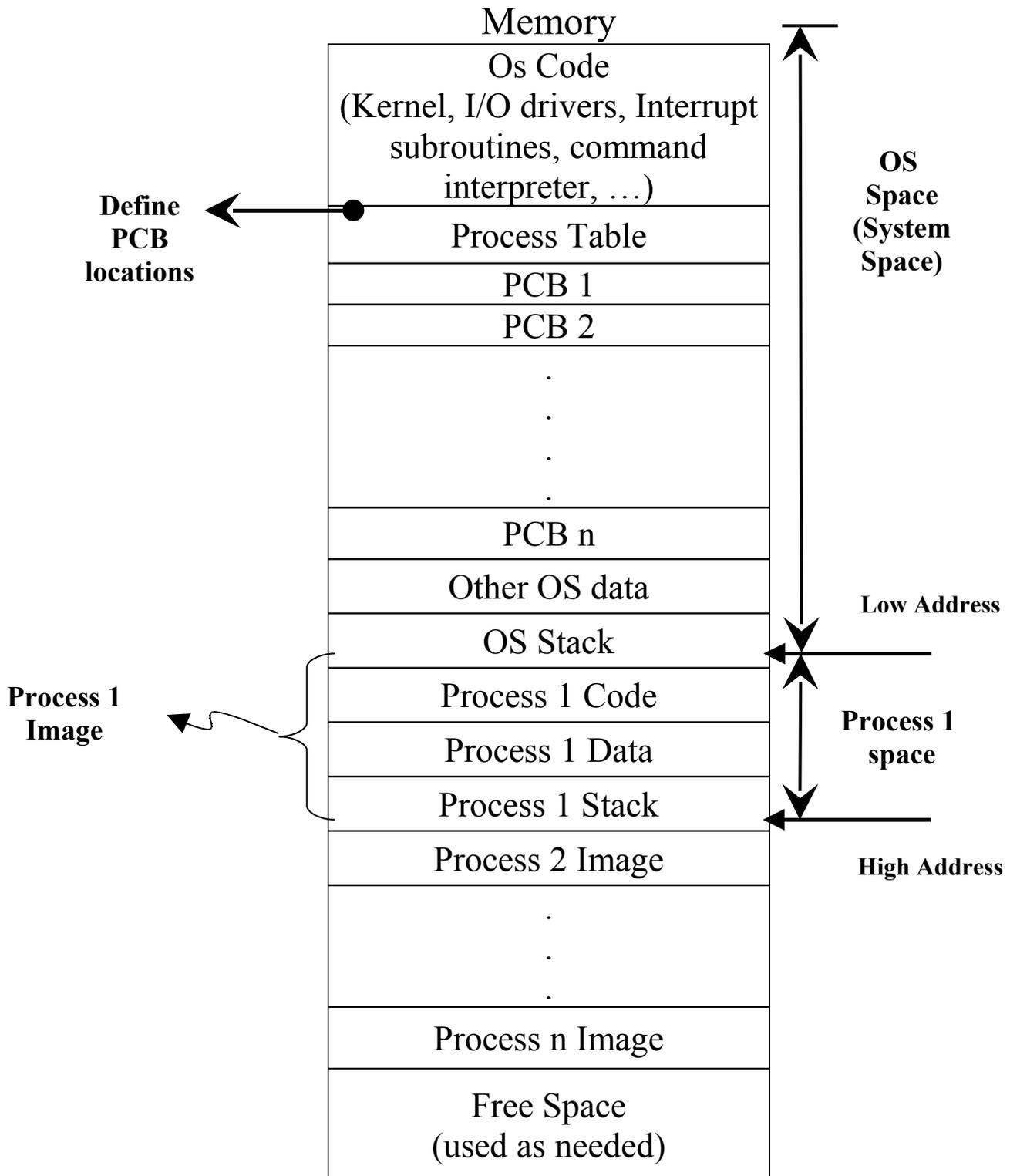


Fig 3.6 Memory Map.

3-6 Context Switching

Context means contents of CPU registers at certain instant. During process running, a context changes with time. When a process exits running state to ready or blocked, the context has to be saved to PCB so that it can be reloaded when the process return to running state. Context switching means saving a process 1 context to PCB1 and reloading process 2 context from PCB2. context switching is carried out by OS when process exits running state and process 2 is dispatched to this state as shown in fig 3.7.

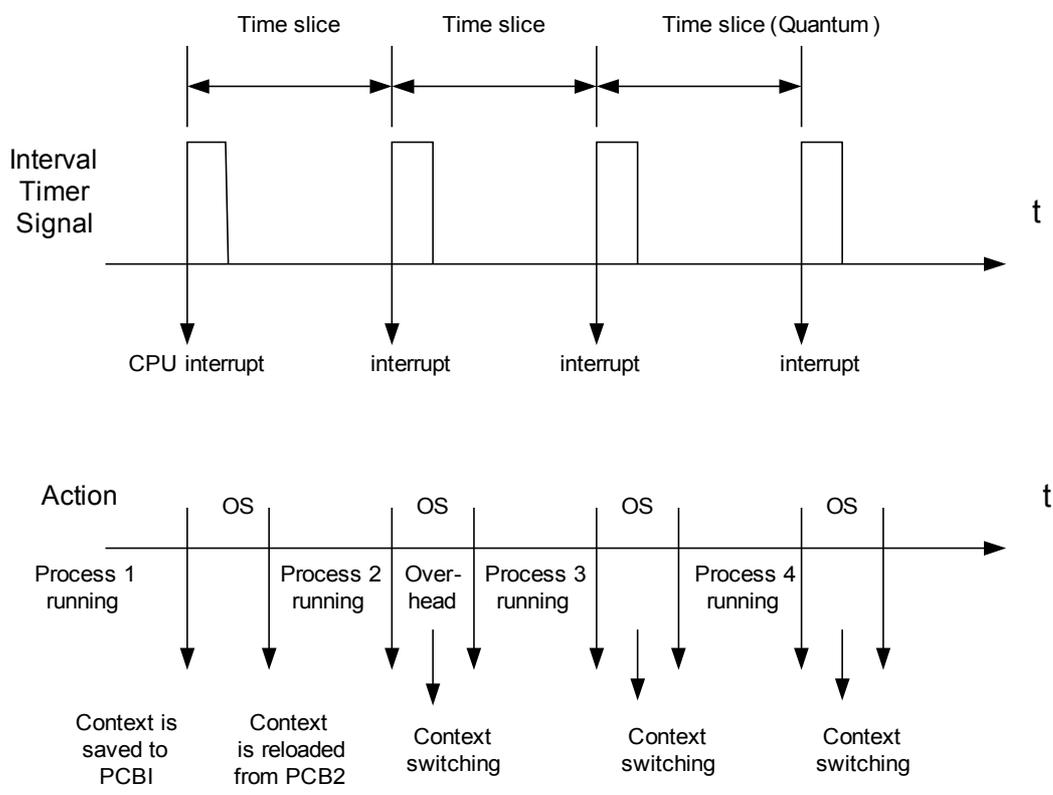


Fig 3.7 Context Switching.

3-7 Swapping

When memory is full and new program is required to be executed then OS has to carry out a swapping process as follows (see fig 3.8) :

1. Save resident process image to disk. The process should be in ready or blocked state (or suspended state which is preferred as will be shown later). The memory space of that image becomes free (empty) and may be used for new process. This free space is called " Swapping Area".
2. Load the new program from disk to swapping area and create a PCB for it (create new process).
3. When the old process is needed to run again, it has to be reloaded to its original space i.e to the swapping area, however, after saving the new process to disk.

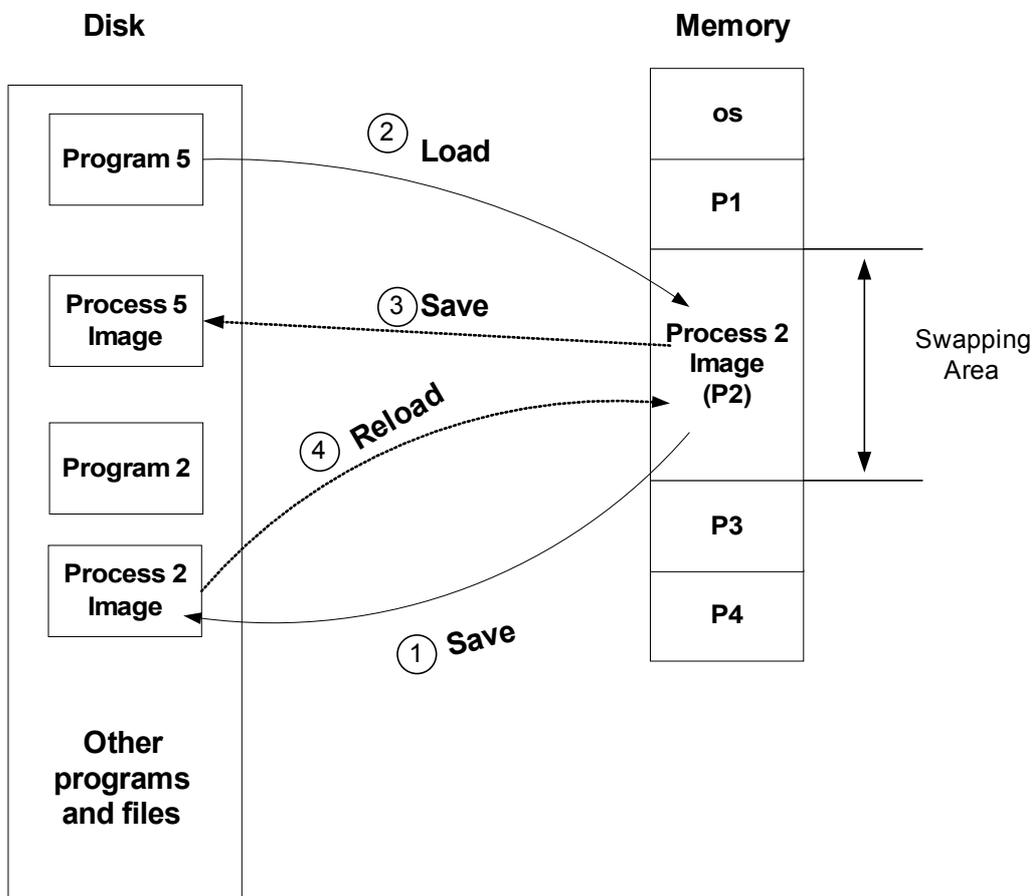


Fig 3.8 Swapping Sequence.

It should be noted that a swapping area may be used for swapping several processes, also, it is possible to have more than one swapping area when necessary.

3-8 Inter Process Communication (IPC)

IPC is sometimes necessary but it presents two main problems:

1. Address violation problem :IPC means sharing some data (access common locations in memory). The shared data will be outside the address space of at least one process which, in turn, creates address violation problem. This problem may be solved by using "System Calls" for shared variables.
2. Write Access Problem : If the shared variable is of type Read/Write then another problem has to be solved in order to keep data integrity. This topic will be discussed later when studying "Asynchronous Concurrent Execution".

3-9 Process State Diagram with Suspend and Resume

Some OSs allow two other process states called "Suspended Ready" and "Suspended Blocked" as shown in fig 3.9. A process may be suspended up on user request (if allowed) or upon OS needs. The suspended process will not compete for CPU time and may be swapped out to disk if memory is full and free space is needed for new processes. The suspended process may be resumed later.

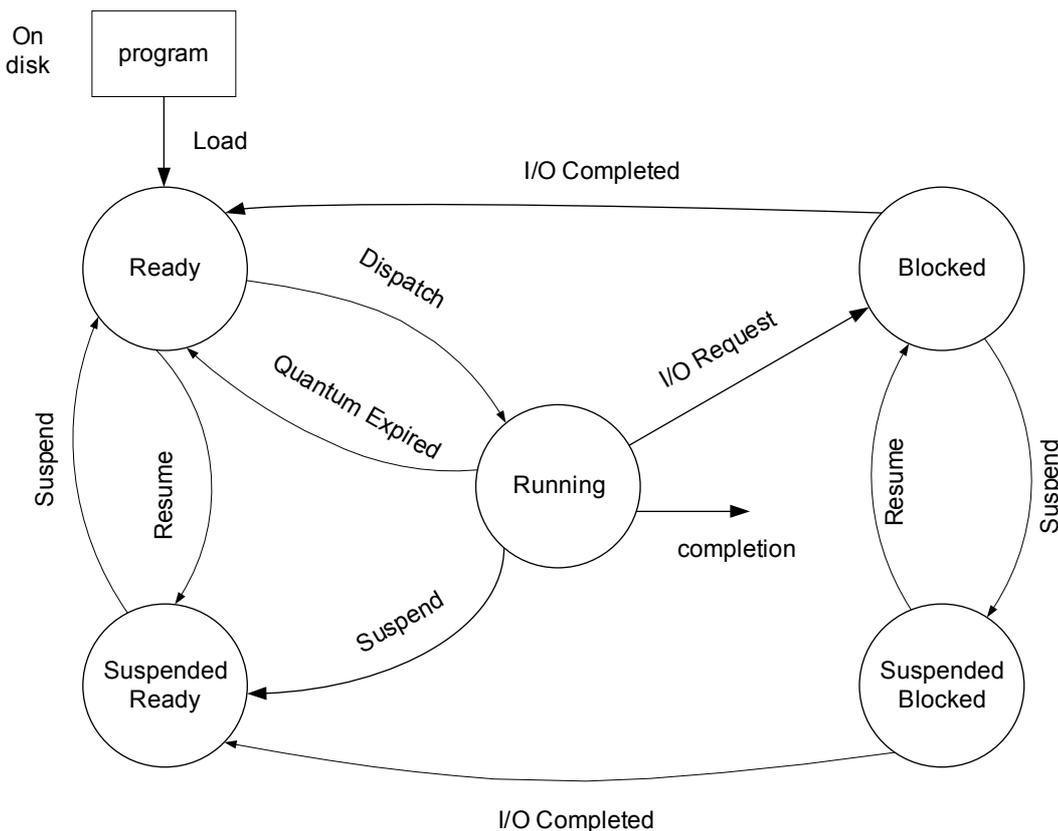


Fig 3.9 Process State Diagram with Suspend and Resume.

3-10 Kernel of OS (Core, Nucleus)

Kernel is the most important part of OS, therefore, it remains resident in memory while other parts of OS may be shuttled (swapped) between memory and disk. The main functions of kernel are:

- Process creation and destruction.
- Process state switching and context switching.
- Dispatching, suspension, and resumption.
- Manipulation of PCBs and process table.
- Interrupt handling.
- I/O handling.
- Memory allocation and deallocation.
- Processor Scheduling.
- IPC.
- File system management.
- Many others.

It should be noted that user applications (programs) can communicate with kernel via using system call instructions only (Remember that Kernel runs in CPU supervisor mode while applications run in CPU user mode).

End of ch3

Chapter 4: Processor Scheduling

4-1 Introduction

Processor scheduling means assignment of CPU time to processes, in other words, allocation of CPU time slots to processes. This operation is dynamic and gets more complicated in case of multiprocessing system.

4-2 Type of Processes

The applications have different time requirements, therefore, can be classified as follows (see fig 4.1):

1. **Batch** : It is off line processing and time is not an issue here.
2. **Interactive** : It is on line processing and time is not critical. Delays are allowed as long as they don't exceed certain limit.
3. **Real Time** : It is a real time processing and the time here is critical because the application may be critical such as a computer controlling a nuclear factory. The computer response should be immediate at certain instants of time.
4. **Dead line** : A process (job) should be completed before certain instant of time.

5. **Streaming** : A process needs regular attention as the case of audio and video.

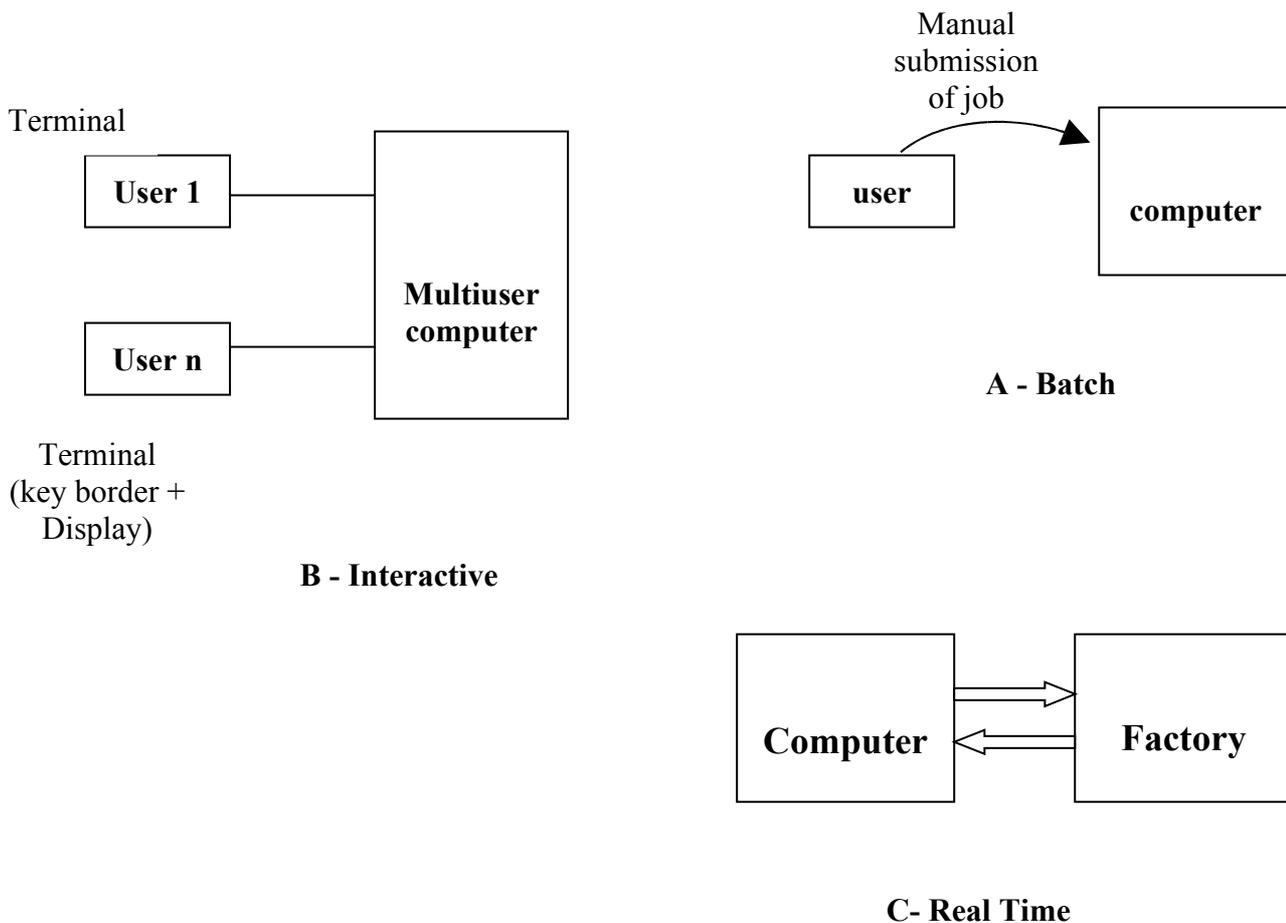


Fig 4.1 Some Types of Processes

4-3 Scheduling levels

The main levels are:

1. High level : determines which jobs (programs) shall be allowed to compete for system resources and hence called " Admission level " or " Admission scheduling" . when a program is admitted, it is converted to a process i.e process is created in Ready state.

2. Intermediate levels : determines which processes to be suspended or resumed in order to balance load on computer resources (CPU, memory,....).

3. Low level : determines when to transfer processes from state to another i.e from Ready to Running, from Running to Blocked, etc.

4-4 Scheduling Objectives

The main objectives are:

1. Be Fair between processes.
2. Maximize number of completed processes per unit time (Called Throughput).
3. Avoid indefinite postponement of any process (Called Starvation Free).
4. Minimize "Overhead" i.e minimize wasted time in scheduling itself.
5. Balance resources usage i.e makes use of all resources all the time.
6. Enforce priority scheme to allow some processes to get more CPU time.
7. Degrade gracefully under heavy loads.
8. Others.

To achieve these objectives, scheduling algorithm should consider the following factors (called scheduling criteria):

- How much total running time has been spent on a process and how much is needed to complete it (if can be estimated).

-
- Does a process generate a I/O request before quantum expired.
 - Real time jobs (processes) are more important (higher priority) than interactive, also, interactive are higher priority than batch.
 - How frequently a processes is generating page faults (will be studied later).
 - Other factors.

4-5 Preemptive and Non preemptive Scheduling

Once a process starts running, it has full control over CPU until an interrupt occurs. Usually, the interval timer is used to interrupt CPU in order to make context switching from a process to another. If interrupt is masked out then a process will continue controlling CPU without allowing other processes to share CPU time. This means that we have two types of processes:

1. **Non preemptive** : once a process has been given a CPU, the CPU can't be taken away from it i.e interrupt is make out. This type is useful for OS operation which masks out interrupt until it completes its work then it enables interrupt and transfer control to normal process. Here, it should be noted that masking and enabling interrupt are privileged instructions and can't be executed in user mode (it is common mode for user processes).

-
2. **Preemptive** : The CPU can be taken away from a process i.e interrupt is enabled (Usual user process operation).

4-6 Time Slice (Quantum)

As mentioned earlier, CPU in multiprogramming is interrupted regularly by interval timer signal which is a periodic of period called " Time slice".

If time slice is small then interrupt will occur very often which means large overhead time is wasted in frequent context switching from process to process. In the other hand, if time slice is too large then short time jobs have to wait long time before completion.

4-7 Priority

If jobs are not of equal importance then priority scheme has to be enforced in order to schedule more CPU time to higher priority jobs. Priority scheme can be " static" i.e fixed with time or " dynamic" i.e changes with time. On the other hand, if jobs are of same priority then "quantums" are assigned periodically to them.

4-8 Types of Scheduling (Scheduling Algorithms):

The main type are:

- FIFO
- Round Robin
- Multilevel Feed back Queues
- Others

4-8-1 First In First Out Scheduling (FIFO)

In this type, processes are dispatched according to arrival time on "Ready Queue". Once a process has the CPU, it runs to completion. FIFO is a non preemptive scheduling and hence it can be used in "single programming" environment (see fig 4.2).

In multiprogramming, it can not be used as a master scheme but as part of it.

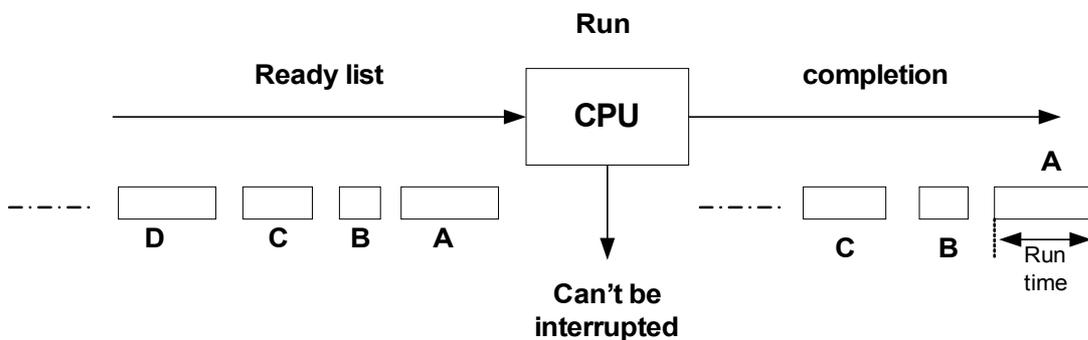


Fig 4.2 FIFO Scheduling (non preemptive)

4-8-2 Round Robin Scheduling

Processes are dispatched FIFO but given a slice time only each turn (round) as shown in fig 4.3. The scheme is useful in multiprogramming. However, it does not support priority as shown in fig 4.4.

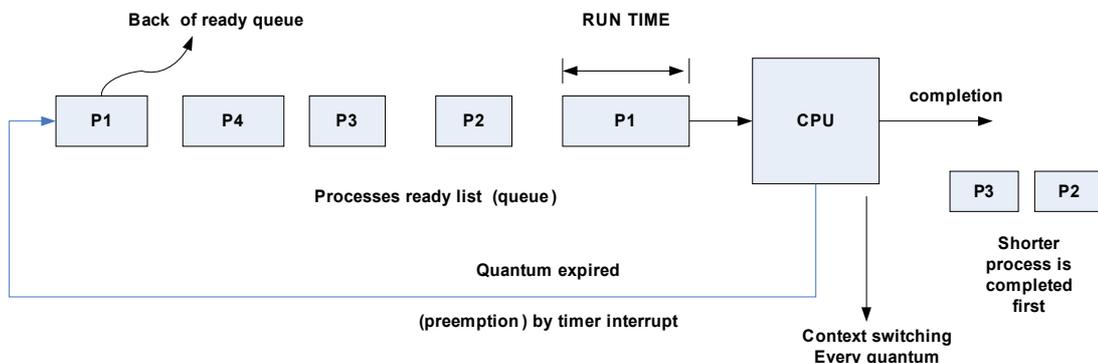


Fig 4.3 Round Robin Scheduling

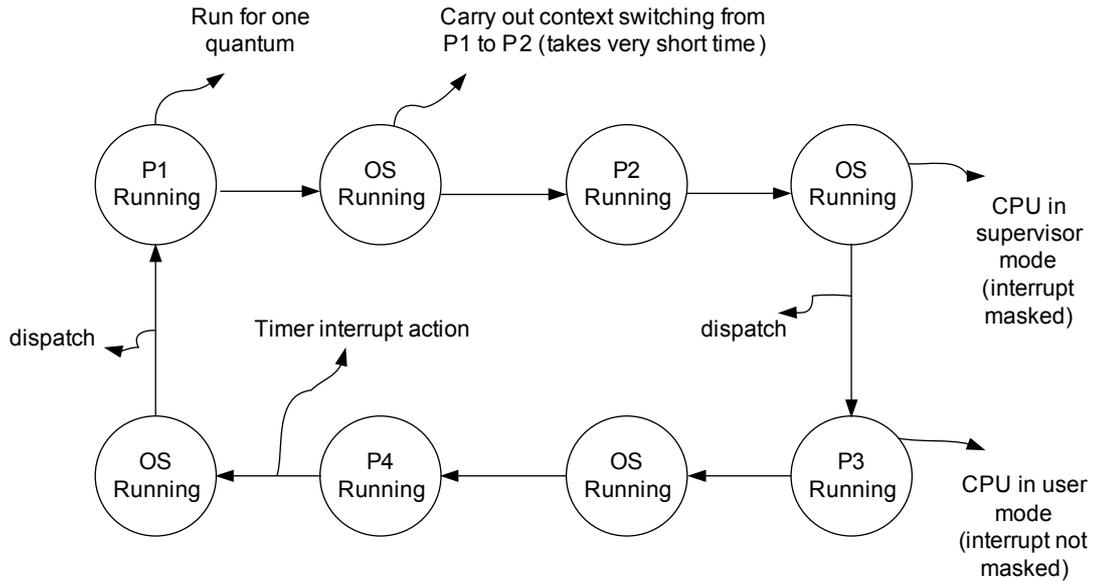


Fig 4.4 Running Sequence in RR

4-8-3 Multi Level Feed back Queue Scheduling

It is preemptive scheduling with the following features (see fig 4.5) :

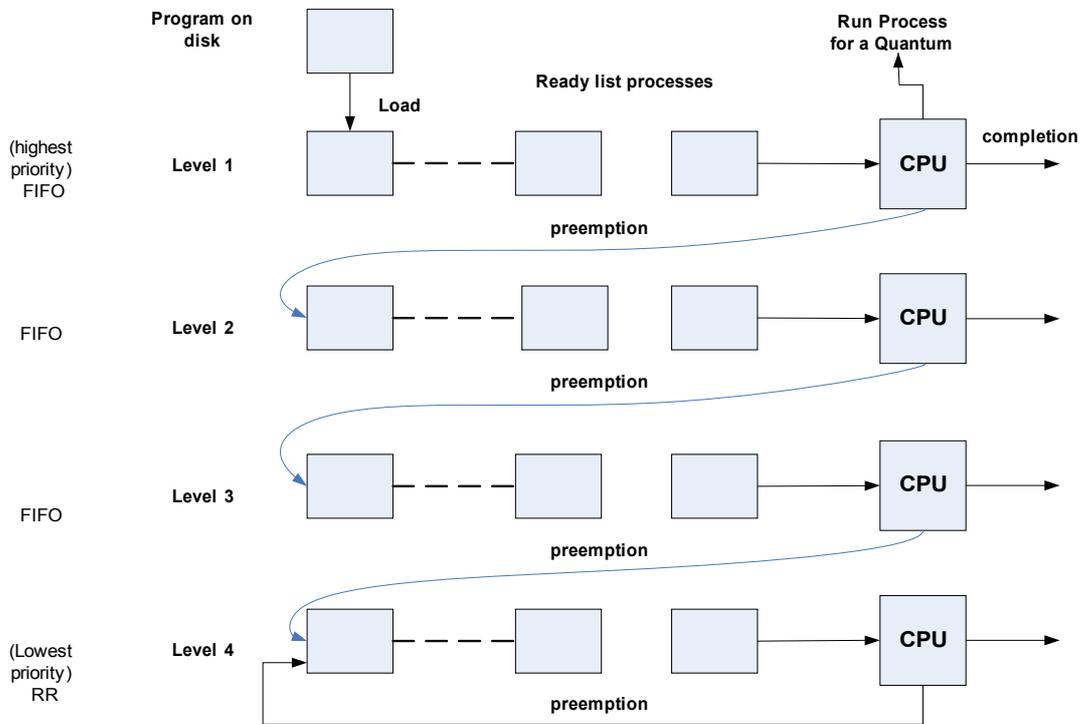


Fig 4.5 Multilevel Feed back Queue Scheduling (preemptive)

-
- 1- Supports dynamic priority where newly created process is given the highest priority level.
 - 2- Priority has several levels. This means that any process will not be assigned any quantum as long as there are processes not completed (waiting) in higher priority levels.
 - 3- Each process will be given several quanta before reaching lowest priority level. This means that short jobs will be favored and completed quickly while long running time jobs (Called CPU bound jobs) will not be favored and given least priority after several quanta.

End of

ch4

Chapter 5:

Memory Organization and Management in Real Memory Systems

5-1 Introduction

There are two types of computer systems differing in CPU design and consequently in memory organization and management.

The first type is called "Real Memory System" while the other is "Virtual Memory System".

Both systems are quite similar as far as the main memory is concerned which means that the term "Virtual Memory" is misleading as will be shown later.

5-2 Definition of Organization and Management

Memory organization means "Logical Partitioning" of memory and program while management means how to use memory partitions to load program partitions so that memory usage becomes very efficient.

5-3 Program Execution Conditions in Real Memory Systems

As mentioned earlier, these conditions can be summarized as follows:

- 1- Program should be in absolute machine code.
- 2- Program should be resident in memory.
- 3- Program addresses should be identical to memory addresses where it is resident.

-
- 4- Program should be loaded in contiguous locations in memory.
 - 5- Program should not be partitioned.
 - 6- Whole program should be resident in memory.
 - 7- In multiprogramming, additional conditions need to be considered such as creating PCB and other stuff.

5-4 Memory Organization in Real Memory Systems

The memory can be logically partitioned in fixed manner (fixed partitions) or variable one. As programs are generally of variable sizes, fixed organization (static) is not very suitable and variable organization (dynamic) is much preferred.

The variable partitioning differs with the type of OS, whether, single or multiprogramming.

5-4-1 Variable Organization in Single Programming OS

In this case the memory map looks like Fig 5.1 where we notice the following:

- 1- OS occupies fixed partition while process occupies variable one.
- 2- Program (Process) always starts at address (a) and hence linker output can be absolute machine code ready for direct loading without address relocation.
- 3- If process volume is larger than available memory then "overlay" technique has to be used (will not be discussed as it is rarely used).
- 4- To protect OS from program, a boundary register has to be available in CPU and loaded with address (a).

Each program generated address is compared with boundary value and if it is smaller than it then address violation exception occurs.

- 5- Memory usage is not efficient as there may be an empty area that is not occupied with a program.

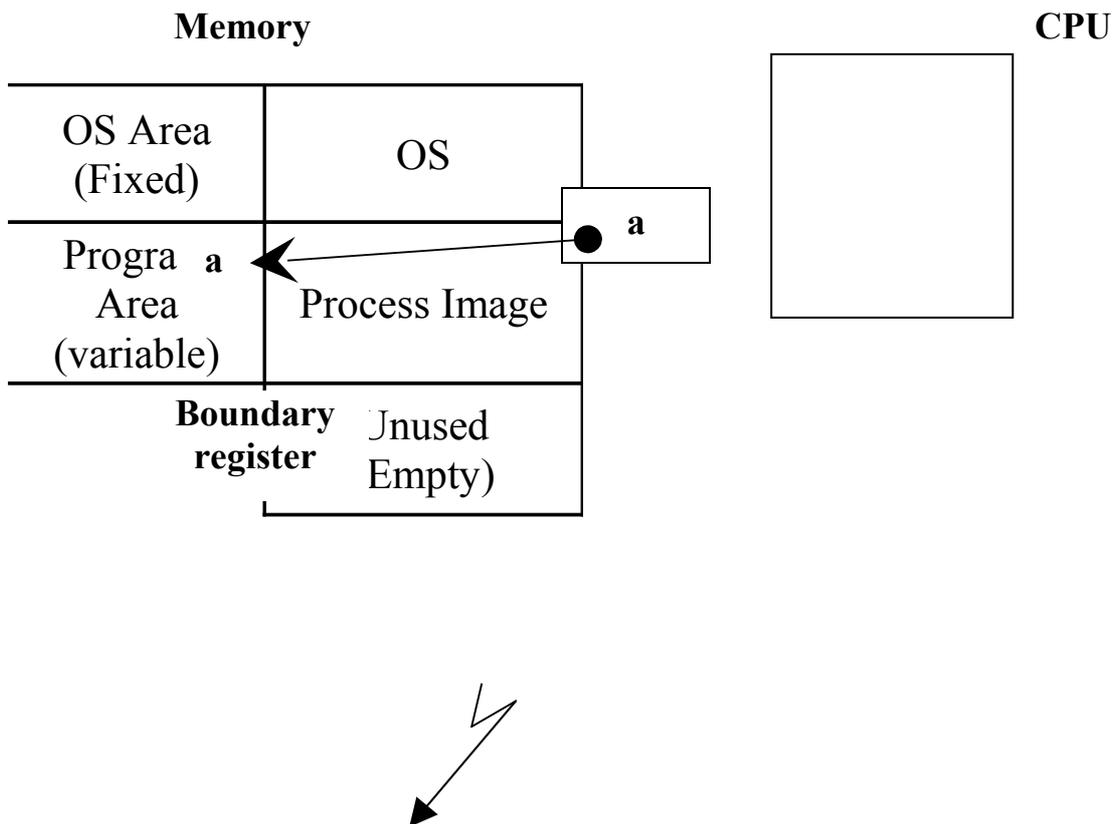


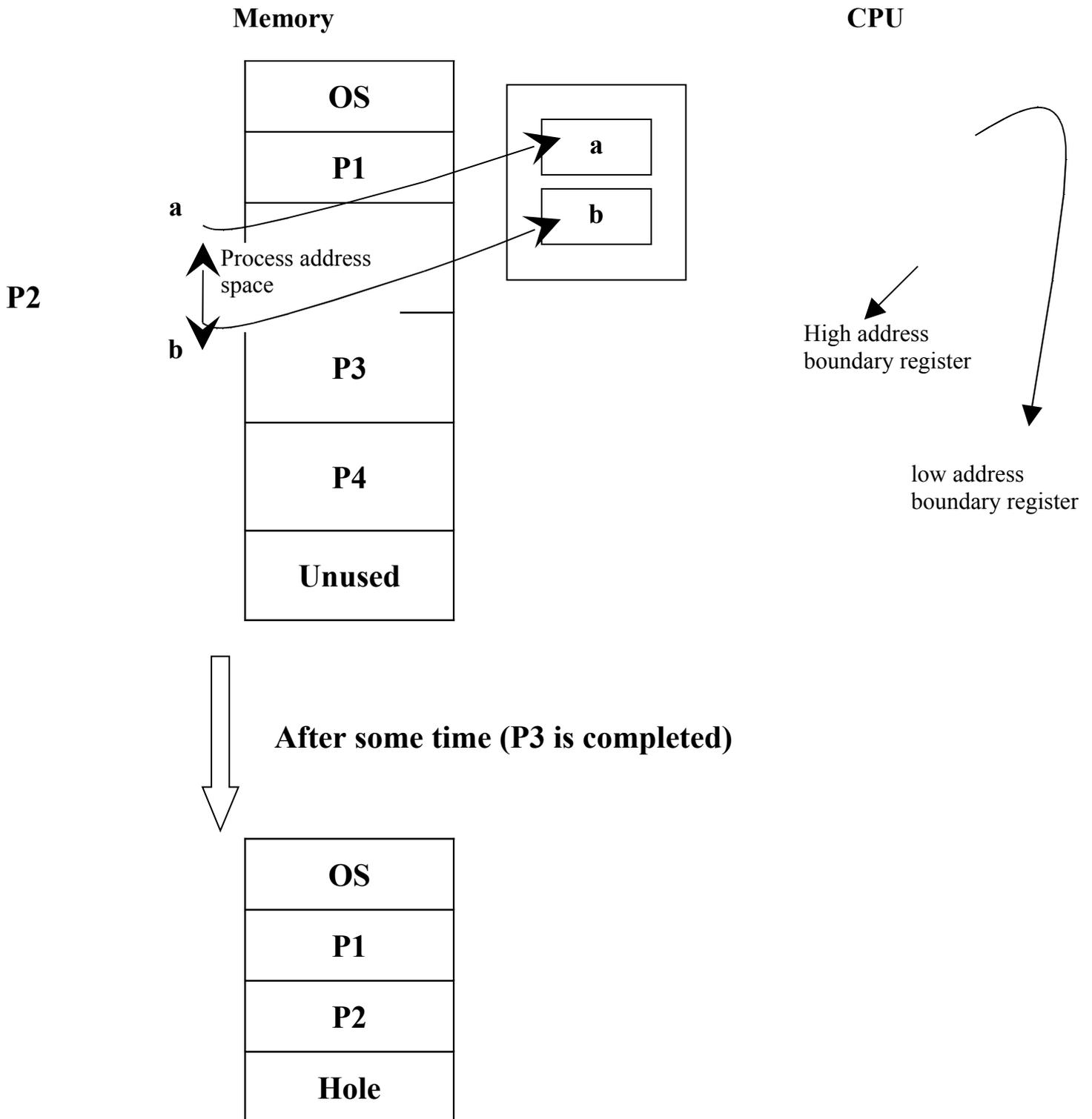
Figure 5.1 Memory Organization in Single Programming OS

5-4-2 Variable Organization in Multiprogramming OS

In this organization, the memory map looks like fig 5.2 where we notice the following:

- 1- The process images are of variable sizes.

-
- 2- When a process is completed, an empty area is left (called Hole).
 - 3- When a program is loaded into memory, the loader has to relocate the addresses according to memory origin where the program is to be resident.
 - 4- Adjacent holes can be merged together by OS to create larger hole "Merging" as shown in fig 5.3.
 - 5- Program protection requires two boundary registers in CPU. Each generated address is compared with this boundary registers and if it is outside them then an address violation exception occurs.
 - 6- All holes may be moved by OS to one area (compaction). The compaction operation is difficult as it requires address relocation of all processes images to the other memory area as shown in fig 5.4.
 - 7- When memory is full and new program needs to be executed then swapping operation is carried out by OS.



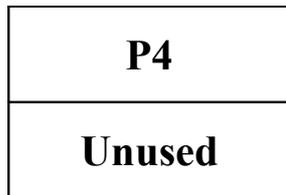


Fig 5.2 Memory Organization in Multiprogramming OS.

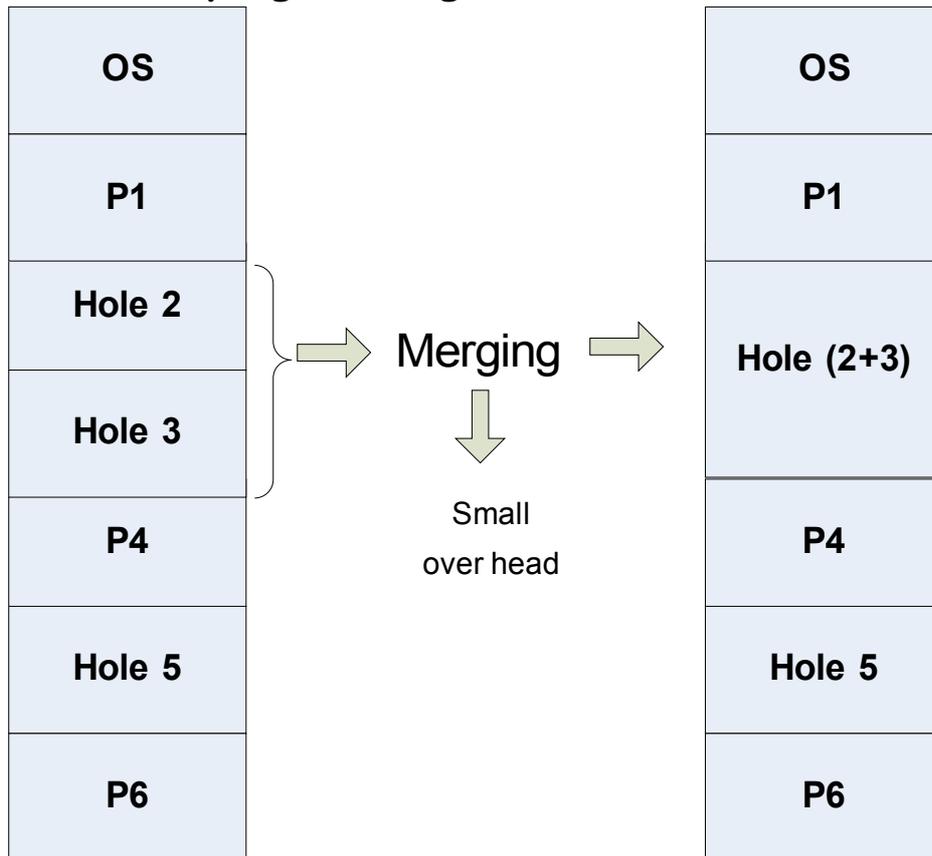


Fig 5.3 Merging Operation

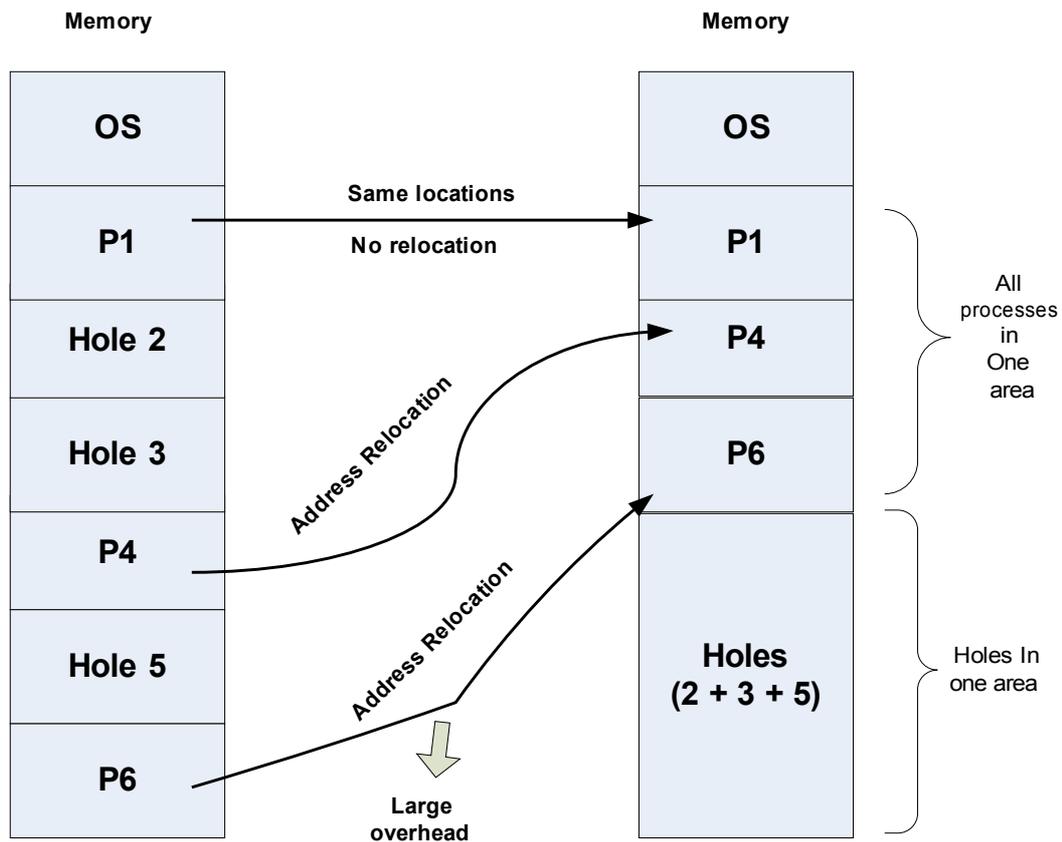


Fig 5.4 Compaction Operation

5-5 Memory management in Real Memory Systems

There are several strategies to manage memory in order to get efficient use of it. These strategies are applicable to multiprogramming systems and can be summarized as follows:

5-5-1 Fetch Strategies

These determine when to fetch the next piece of program or data i.e. when to load them from disk to memory. Here, there are two schemes:

- 1- **Demand fetch:** Load the next piece when needed.
- 2- **Anticipatory fetch:** Load the piece when it is expected to be needed. This early fetch may speed up the system when the expectation is correct, otherwise, it is mere waste of time.

5-5-2 Placement Strategies

These determine the proper memory space (hole) to place job (process) into it. The main strategies are:

- 1- **Best fit:** place the job in the smallest possible hole. The disadvantage is that the rest of hole will not be enough for new job (see fig 5.5).
- 2- **First fit:** place the job in the first suitable hole. The advantage is low overhead i.e. small CPU wasted time in implementing the strategy (see fig 5.5)
- 3- **Worst fit:** place the job in the largest available hole. The rest of hole may be still enough for new job (see fig 5.5)

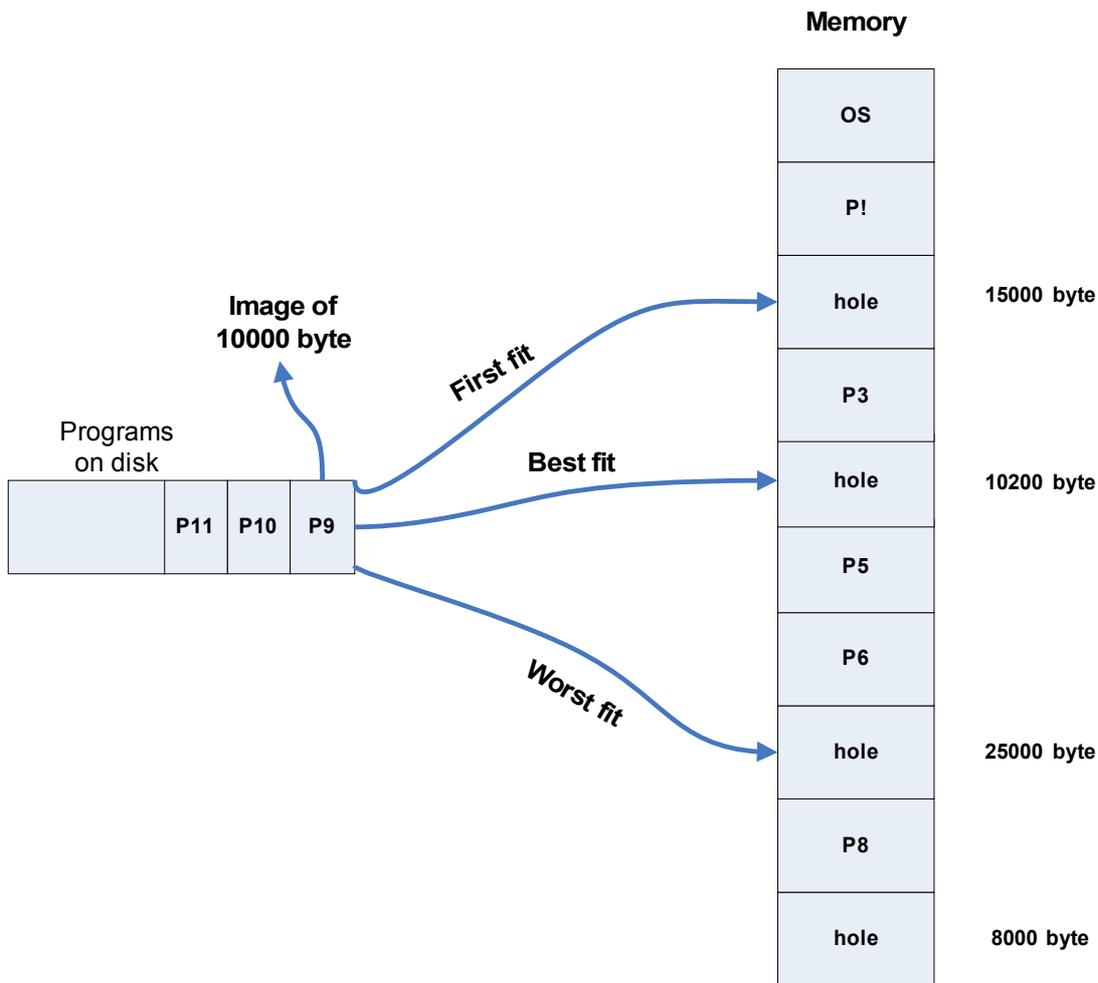


Fig 5.5 Placement Strategies

5-5-3 Replacement Strategies

These determine which piece of data or program has to be swapped out to disk when memory is full and a new program is needed to be executed.

These strategies will not be explained here, however, similar strategies will be discussed in virtual memory systems.

End of Chapter 5

Chapter 6: Virtual Memory Organization

6-1 Introduction

As mentioned earlier, "Virtual" is misleading name and a better name could be "Dual Addressing computer systems" because of the following reasons:

- Virtual memory systems are similar to real memory systems in having a physical main memory and memory address bus.
- Both types of systems have physical CPU but the addressing scheme is different in both.
- In real systems, there is only one type of addresses and hence the program generated address is the same as the one sent to memory address bus (lines).
- In virtual systems, there are two types of addresses which are virtual addresses and real addresses. The program generated address (also called virtual address) is not sent directly to memory address bus but has to be converted by CPU to a new address (Called Real Address) which then sent to memory bus. The conversion process is called "Mapping" and has to be done to each virtual address and hence a Virtual CPU is relatively slower than Real CPU.

6-2 Addressing Scheme in Paging System

Suppose that we use 32 bits for address then the address space will be as shown in fig 6.1.

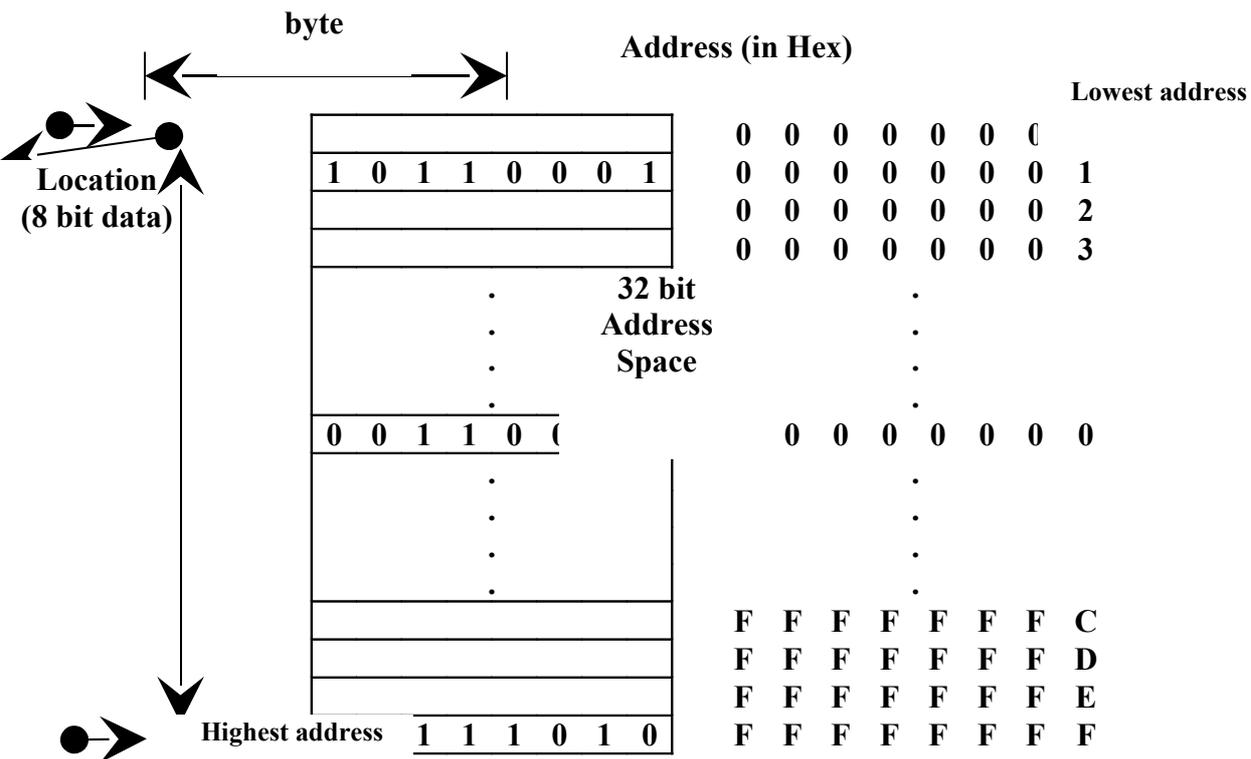


Fig 6.1 32 bit Address space

Now, let us partition logically this space into equal size partitions called "Pages" then the single number address can be written as two numbers [p, d] where p represents "page number" and d represent "displacement" within this page as shown in fig 6.2

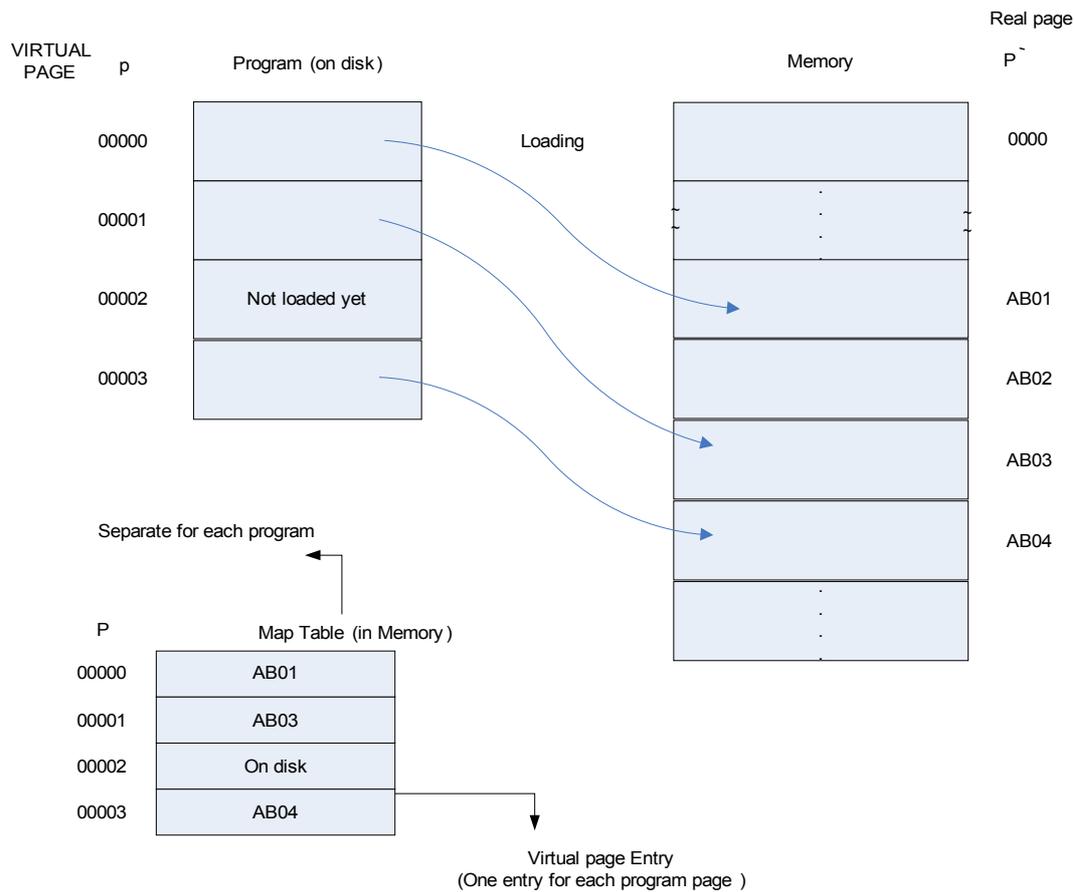


Fig 6.3 Virtual Paging System

6-4 Direct Mapping of Virtual Paging System

In this type of mapping, the map tables are stored in main memory and each program generated address has to be translated to memory address in real time during program running and this activity is called "Dynamic Address Translation DAT". The mapping activity is carried out by CPU as shown in fig 6.4.

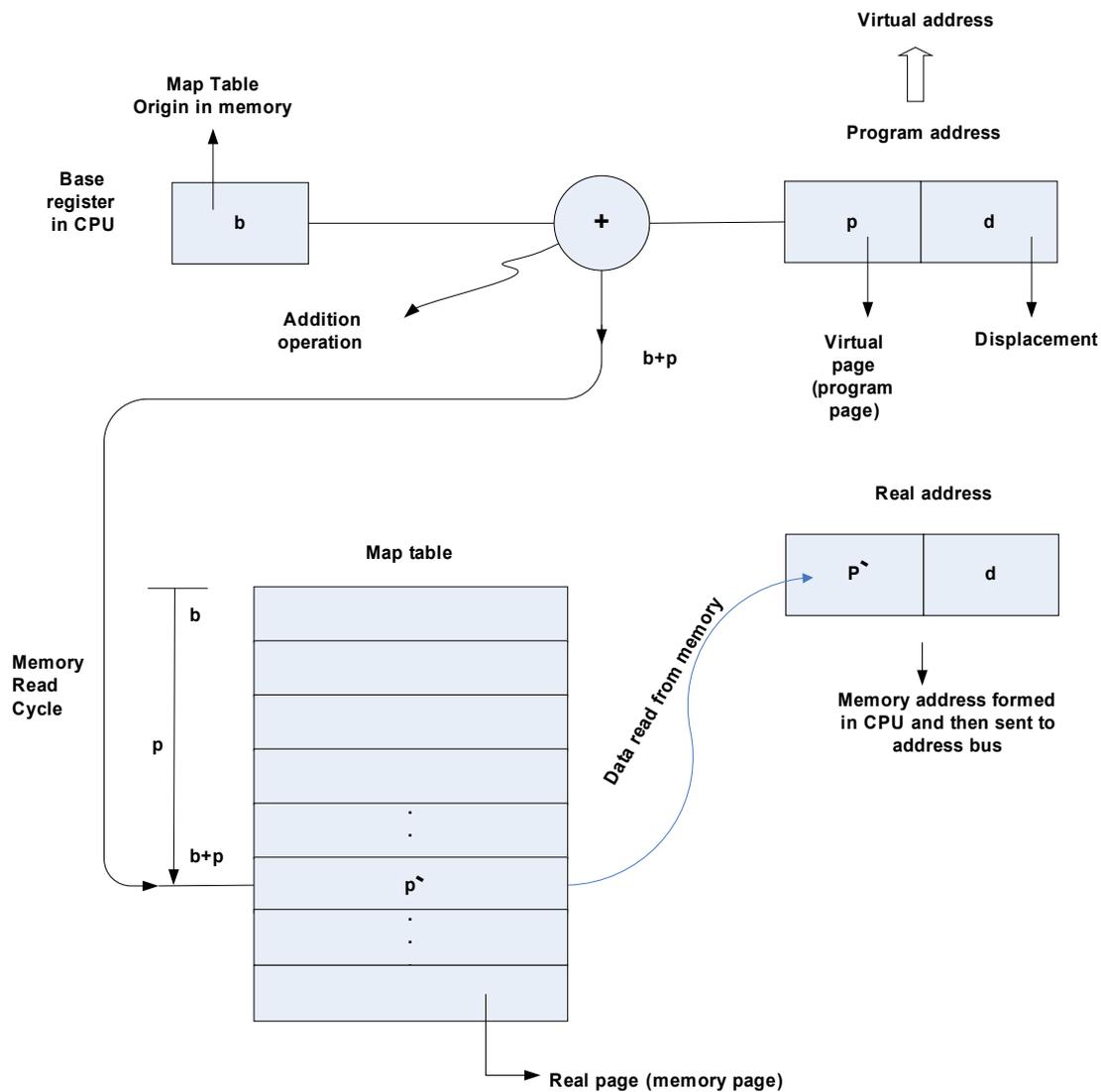


Fig 6.4 Direct Mapping in Virtual Paging System

6-5 Advantages of Virtual Memory System

The main feature of virtual system is the independency of program and memory addresses and this introduces many advantages as follows:

- 1- The program can be executed without having it all loaded into memory. When a generated address points at a page still stored on disk then this page has to be loaded into memory. This property means that a program can be larger in size than available memory.

-
- 2- The program pages should not be loaded into contiguous memory pages and hence no need to merge memory holes at all which reduces overhead.
 - 3- The program addresses are absolute and hence no need to relocate them during loading program pages into memory.
 - 4- The program pages can be displaced into memory very easily i.e. without address relocation as long as the page changes are registered in map table.
 - 5- Memory management will be simplified and hence low overhead because any program page can be loaded into any memory page.
 - 6- Sharing program and data pages will be simplified by using the map tables as shown later.
 - 7- Security and protection will be simplified using map tables as will be shown later.

The main disadvantage of virtual system is that it is comparatively slower than real system because of address mapping.

6-6 Map Table

As mentioned earlier, each program being executed has a separate map table resident into memory for the time duration of execution and then removed. Each program page has separate entry called program page entry or "Virtual

Page Entry". The page entry includes real page number and some other data as shown in fig 6.5.

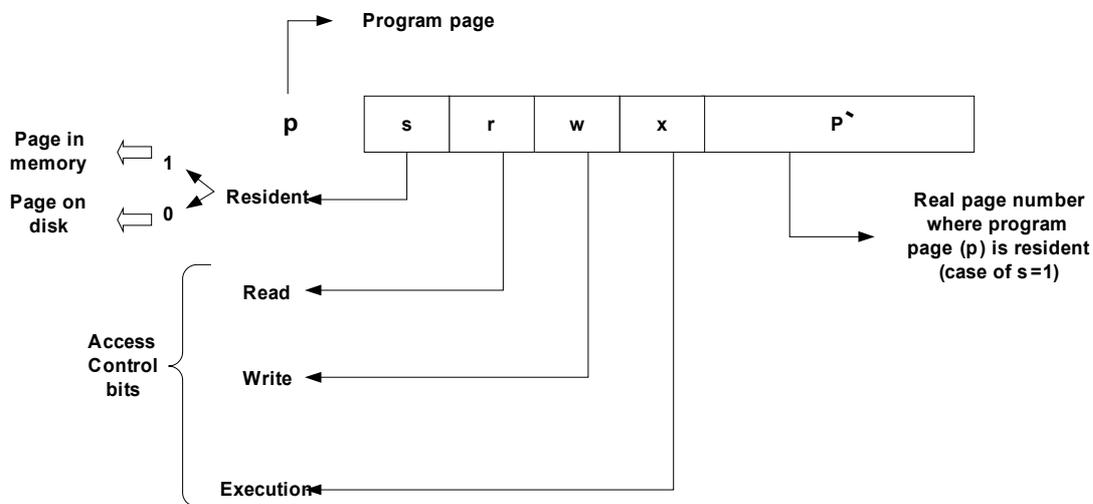


Figure 6.5 Page Entry in Map Table for Virtual page P

In this figure, we notice the following:

- Access control bits are very useful for achieving protection scheme of programs and data.
- If $S=0$ then a program page is still on disk and hence P^{\sim} is replaced with track and sector numbers on disk.
- The map table is accessed for each address translation and hence it is preferred to be resident in fast access memory such as "Cache".

6-7 Sharing in Virtual Paging System

Sharing is easily implemented without causing address violation exception as shown in fig 6.6.

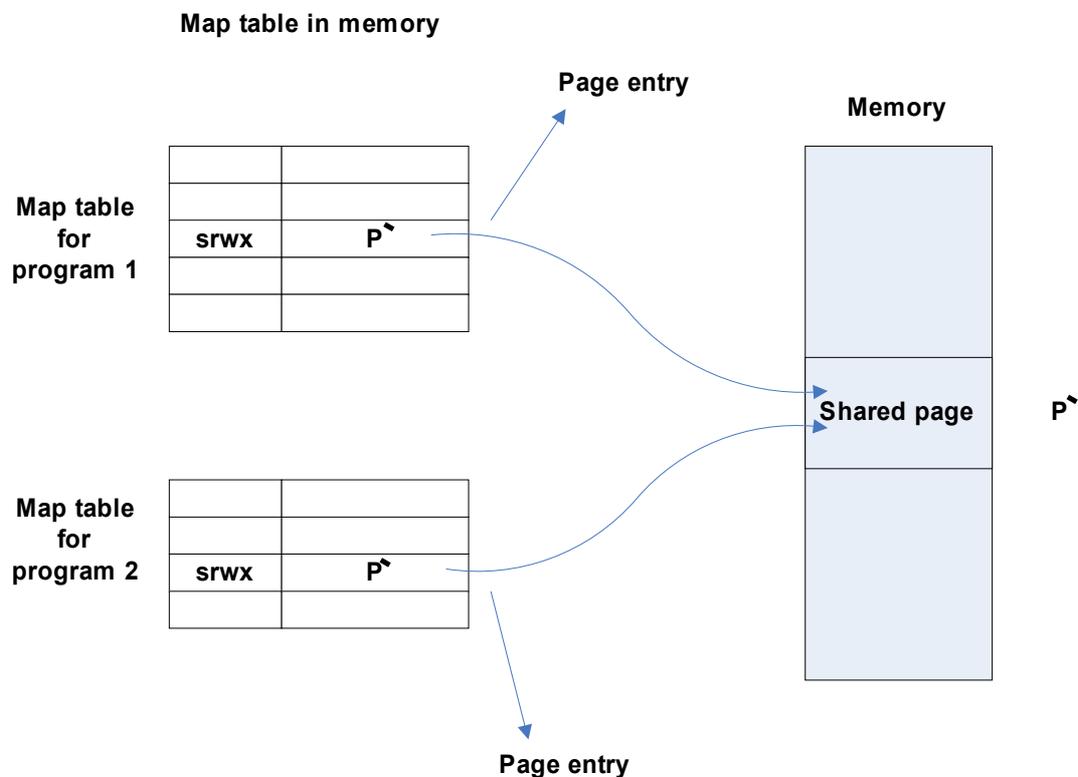


Fig 6.6 Sharing in Virtual Paging System

6-8 Associative Virtual Paging System

In direct mapping, the map table is stored in main memory but here it is stored in "Associative memory" which also called "Content Addressable Memory".

In this type of memory, there are two parts for each location and if we know the first part then the second part will be known. The map table, here, includes entries for pages that are resident in memory i.e. there are no entries for pages that are still on disk as shown in fig 6.7.

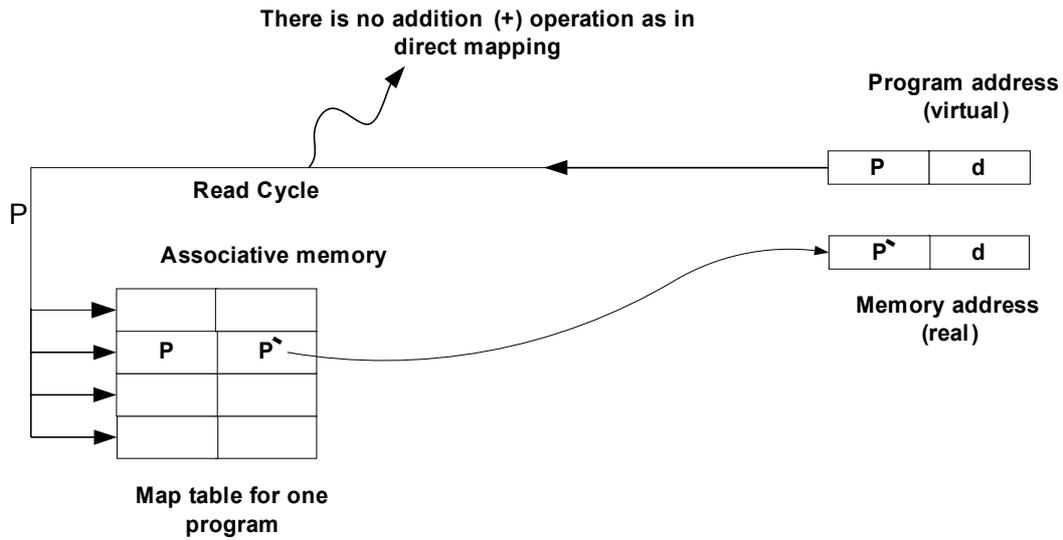


Fig 6.7 Associative Mapping

It should be noted that the associative memory is quite expensive. Also, it's worth noting that there are other virtual systems not only the "paging" one.

6-9 Exercise

We have the map table shown in fig 6.8. Calculate the real address corresponding to virtual address of 00002 ABF.

P					P'			
0	0	0	0	0	7	8	A	3
0	0	0	0	1	On disk			
0	0	0	0	2	B	2	3	C
0	0	0	0	3	On disk			
0	0	0	0	4	On disk			
0	0	0	0	5	5	4	B	B
0	0	0	0	6	B	4	2	3

Fig 6.8 Map Table Example

Solution:

From above figure, we conclude that the virtual address (program address) can be written as:

[00002, ABF].

For $P = 00002 \implies P' = B23C \implies$

Real address = [B23C, ABF] which can be written as:

B23C ABF.

End of Chapter 6

Chapter 7: Memory Management in Virtual Memory Systems

7-1 Introduction

In virtual system, management should answer the following questions:

- When to fetch pages
- Where to place fetched Pages
- Which pages to be replaced when memory is full.

These questions will be answered below but after discussing some related issues such as page fault, working set, locality, etc.

Note : We are studying paging systems and hence memory organization is simple as memory is partitioned into equal pages.

7-2 Page Fault

When a running program references a page that it is not resident in memory then a "page fault" exception (interrupt) occurs which fetches that page and places it into memory. Management strategy should aim at decreasing page faults i.e. increasing the time between page faults (interfault time).

7-3 Working Set

A working set of a program (process) is a collection of pages that a process is actively referencing and it changes with time. For a program to run efficiently (less page faults), its working set should be maintained in memory, otherwise, excessive page activity (Called thrashing) might occur. To

avoid thrashing, it is recommended to keep half program pages into memory.

7-4 Locality

Locality means the tendency of a process to reference storage in non uniform pattern but highly localized in time and space and hence we have:

- **Time Locality (Temporal Locality):** The recently referenced pages are more likely to be referenced again in the near future & hence these pages should not be swapped out to disk.
- **Space Locality (Spatial Locality):** When a location is referenced, it is likely the nearby locations will be referenced as well and hence should not be swapped out to disk (case of arrays, program code, etc.).

Locality principle helps in achieving "**optimality**" i.e. the swapped out page is not referenced soon.

7-5 Page Size

Small page size means a large number of pages and hence large map tables and excessive overhead. Large page size decreases map tables size, however, it causes other problems such as thrashing and less number of jobs in the multiprogramming environment.

7-6 Fetch Strategies

The main strategies are;

- 1- **Fetch on demand:** Fetch page when it is needed i.e. when page fault occurs.
- 2- **Anticipatory fetch:** Fetch a page when it is expected to be needed. Here, it is possible to make use of spatial locality principle.

7-7 Placement Strategies

It is very simple as any program page can be placed in any free memory page under the condition of registering (recording) the page numbers in the related map table.

7-8 Replacement Strategies (Swapping Strategies)

When memory is full and OS needs to load (fetch) new page from disk then an already resident page has to be swapped out to disk and replaced by the new incoming page. The selection of page to be swapped out can be done using one of following strategies:

7-8-1 Random Page Replacement

- Replace any page at random
- Low overhead (advantage)
- Replaced page may be referenced soon (disadvantage), therefore, it is rarely used.

7-8-2 First In First Out Replacement (FIFO)

Each page is "time stamped" when fetched to memory as in fig 7.1. The older page (first fetched) is replaced first even if it has been used recently.

The disadvantage is that elder page may be heavily used by several programs (users) e.g. text editor.

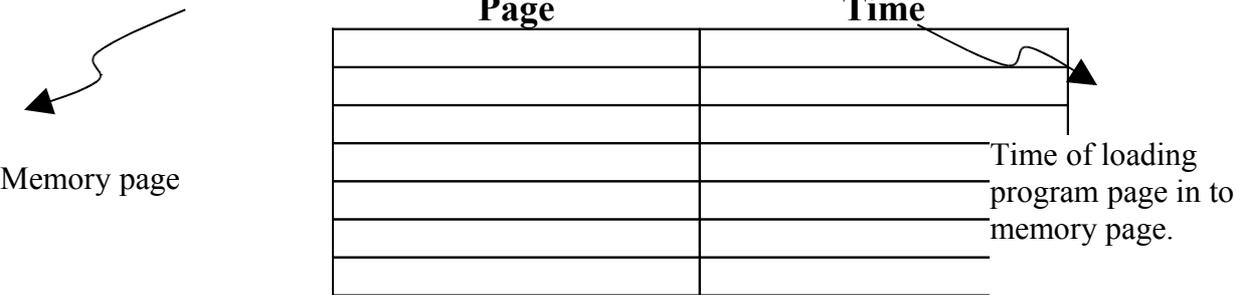


Fig 7.1 Time Stamp of Loading

7-8-3 Least Recently Used (LRU)

We replace the page that has not been used for the longest time without considering its loading time. Here, it is necessary to use time stamp as in fig 7.2 where time indicates "referencing" time and not "loading" time

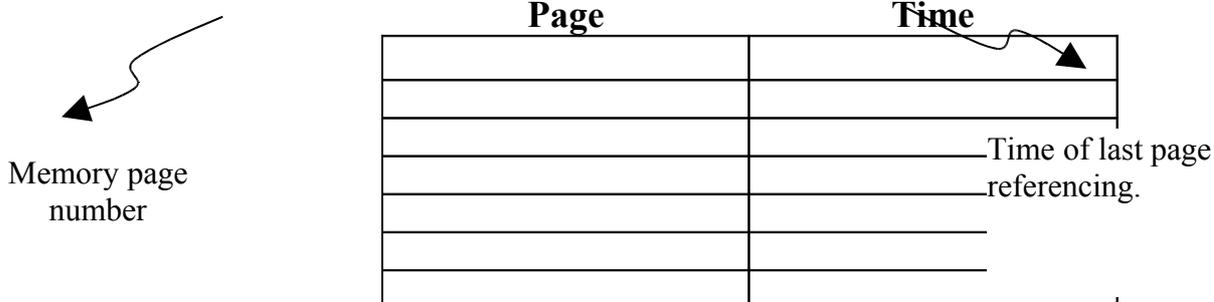


Fig 7.2 Time Stamp of Referencing

7-8-4 Least Frequently Used (LFU)

Here, it is necessary to count number of page references and replace the one with least count i.e. less used. Count stamp is needed as in fig 7.3.

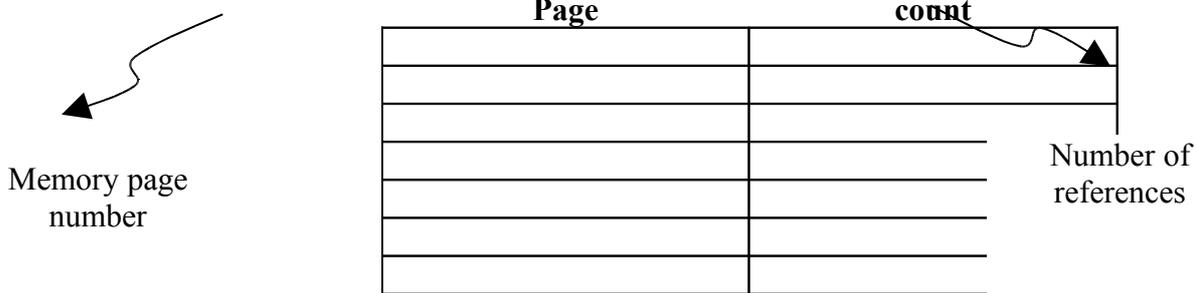


Fig 7.3 Count Stamp

End of Chapter 7

Chapter 8: Thread Concepts

8-1 Introduction

There are two types of programming languages:

- 1- **Single threaded:** Allows single thread of control in the program and hence concurrent activities are not possible within the same program. Examples of such language are: C, C++, VB, etc.
- 2- **Multithreaded:** Allows several threads of control in the program and hence concurrent activities are quite possible within the same program. Examples of such languages are: C#, VB.NET, JAVA, ADA, etc.

It is worth noting that "single threaded languages" are also called "non-threaded" or "sequential" while "multithreaded" are called "threaded" or "parallel".

8-2 Non-Threaded and Threaded Algorithms

Suppose we want to calculate the following expressions:

$$Y = (a1+x)^3 + (a2+x)^4$$

where a1, a2 are constants and x is input variable. This calculation can be done as follows:

1- **Using Non-Threaded Algorithm:**

The calculation is shown in fig 8-1 and we notice that it takes a total of 7 arithmetic operations

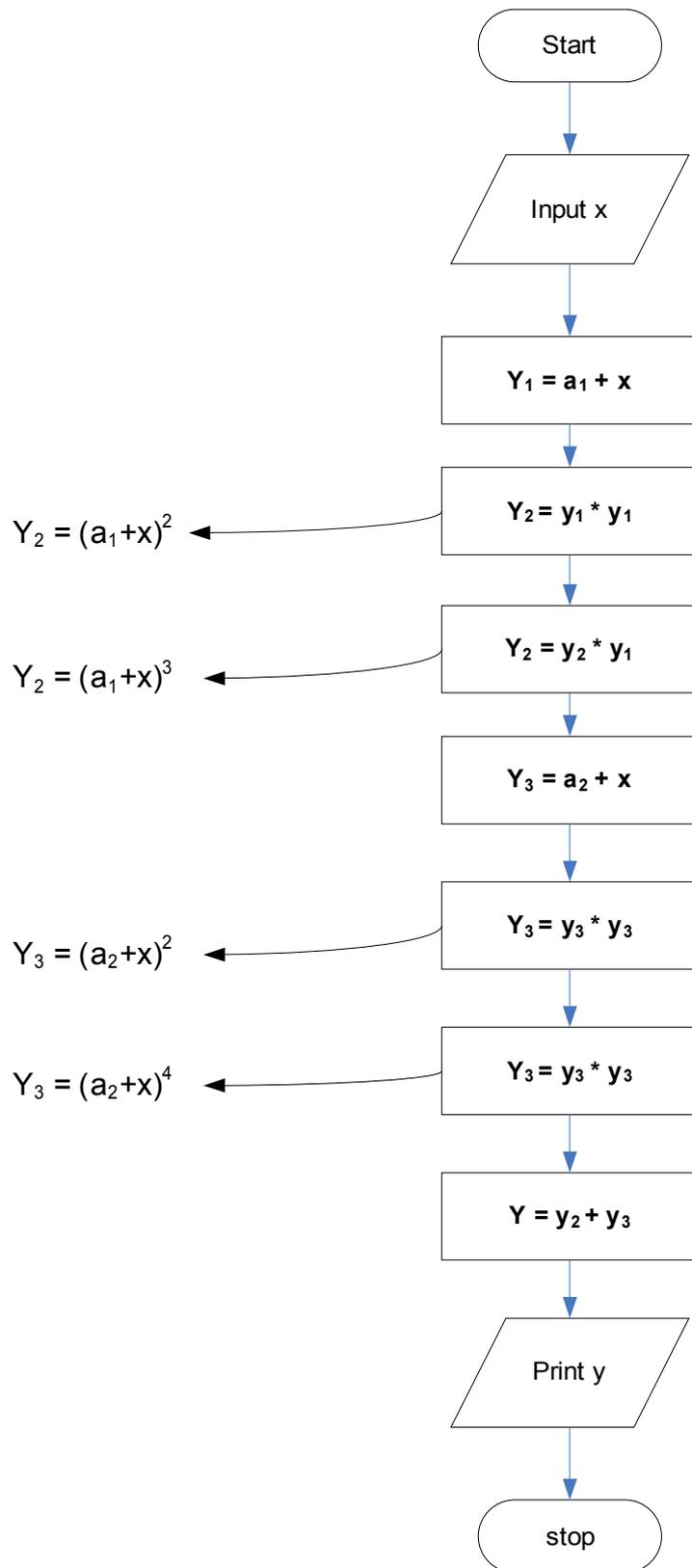


Fig8.1 Non Threaded Algorithm

2- Using Threaded Algorithm:

The calculation is shown in fig 8.2. The number of arithmetic operations in Thread1 is 3, and in Thread2 is 3.

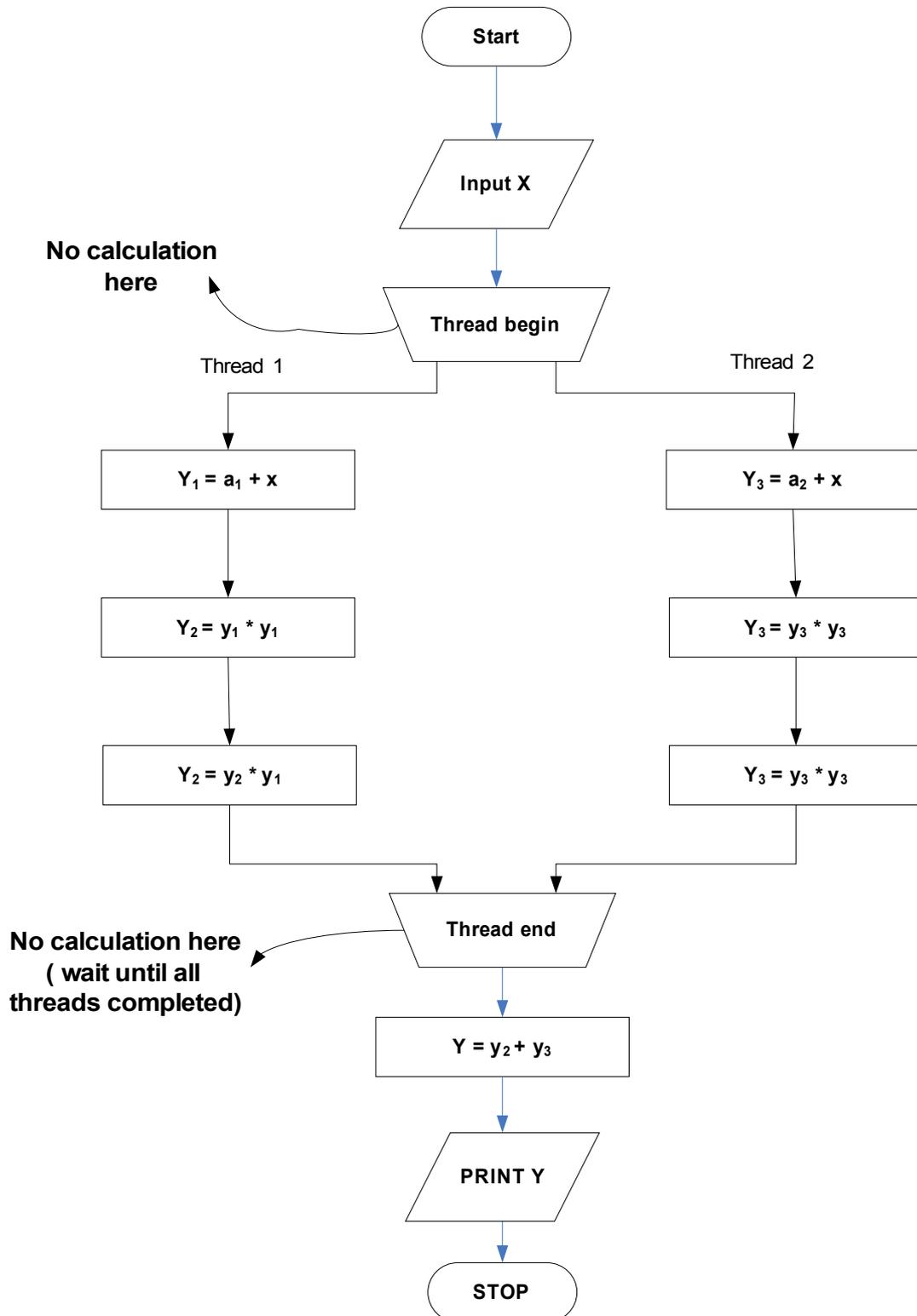


Fig8.2 Threaded Algorithm

Thread1 and Thread2 can be executed concurrently and hence the equivalent number of operations is 3 only. The total number of operations is 4 which is less than 7 needed in non threaded algorithm

8-3 Definition of Thread in OS

A thread is a stream of instructions (line of control) within a process that can be executed independently of other threads. This means that a process may create at sometimes several threads that can be executed concurrently by several processors or each thread is dispatched for one time slice.

Another definition of thread is a "Light Weight Process LWP" as it simulates the original process "Called Heavy Weight Process HWP" in the running for one time slice when it is dispatched.

As the thread runs in a process environment, therefore, it shares a process address space which means that communication between threads is very simple and variable sharing is possible without causing address violation problem.

8-4 Motivations of Threads

From above discussion, we deduce that thread motivations can be summarized as follows:

- 1- Fast execution of a program as it can make use of several processors at the same time (case of multiprocessing) or dispatched more time slices (Case of Single Processor CPU)
- 2- Easy communication between threads as they share the same process address space that created them.

3- Easier design of some applications which have a lot of parallel activities such as a "Word" program.

❖ **Note:** The usefulness of multithreading can be made clear by considering a "Word" program.

Each time a user types a character at the keyboard, OS receives a keyboard interrupt and issues a signal to the word program (process). The word process responds by storing the character in memory and displaying it on the screen. Because today's computers can execute hundreds of millions of instructions between successive keystroke, a word process can execute several other threads between keyboard interrupts. For example, a word process may detect misspelled words as being typed and periodically save a copy of document to disk. Each feature may be implemented by separate thread. As a result, the Word process (processor) can respond to keyboard interrupts even if one or more of its threads are blocked due to I/O operation (e.g. saving copy of file to disk).

8-5 Thread State Diagram (Thread Life Cycle)

A simple thread state diagram is shown in fig 8.3 where we notice the followings:

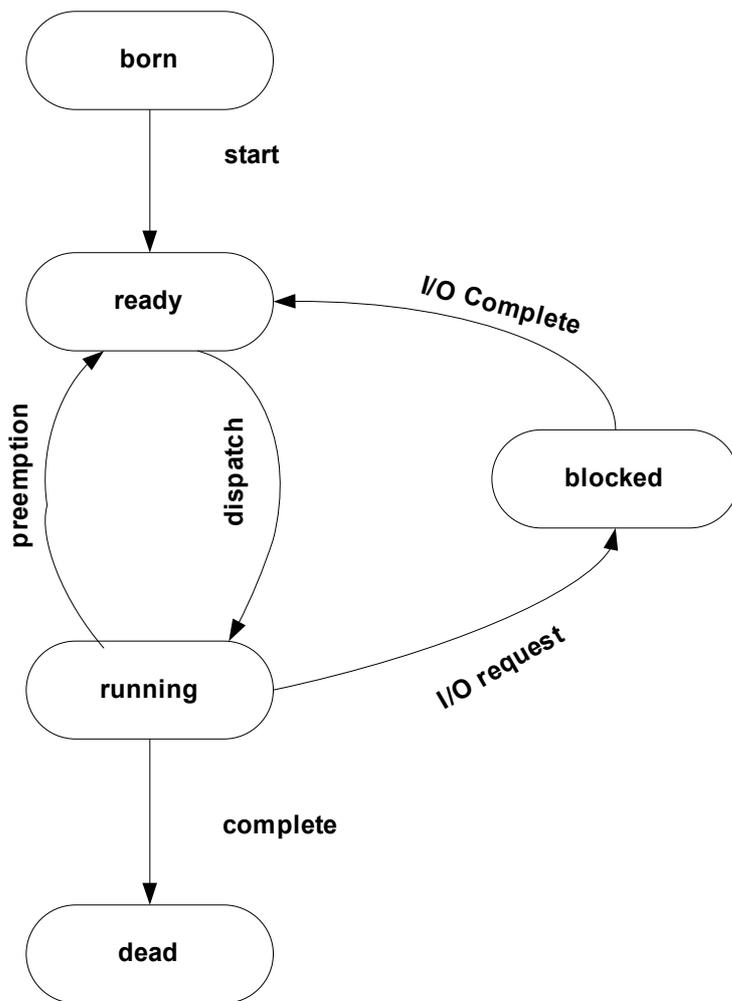


Fig 8.3 Simple Thread State Diagram

- A thread has several states in similar way to process.
- A thread is dispatched for a time slice when transferred from Ready to Running state.
- When a thread is completed it becomes dead and will not be allocated CPU time.

8-6 Variable Sharing Between Threads

As mentioned earlier, the sharing activity is very simple as all threads share the same process address space, however, when a variable sharing is of "Read-Modify-Write" type then complicated problems may occur and have to be solved as will be discussed in the next chapter.

End of Chapter 8

Chapter9: Asynchronous Concurrent Execution

9-1 Introduction

As mentioned earlier, in threaded languages, several threads belonging to one process may be executed concurrently (Concurrent Execution) and independently (Asynchronously). These threads share process address space and hence can share any variable in that space without causing address violation exception (interrupt). If variable sharing is of "Read" type then there is no problem but if it is of "Read-Modify-Write" type then there is a big problem that has to be solved, otherwise, errors may occur in the execution results.

Also, it is worth noting that a similar problem may occur in the multiprogramming environment when several processes share a variable of "Read-Modify-Write" type.

From the above, it is clear that we can name this chapter as: "Access Control to a Write shared variable in a multitasking environment" or as: "Synchronization of Asynchronous Tasks in a Multitasking Environment".

- ❖ **Note:** A task usually indicates a thread in a multithreaded environment or a process in a multiprogramming environment. This means that a multitasking OS could be:
 - Multithreaded, Single programming.
 - Single threaded, Multi programming.
 - Multi threaded, Multi programming.

Also, the OS in all above cases may support single processor CPU or multiprocessing i.e. Multiprocessor CPU. (Note that multi processes means multiprogramming but multiprocessing means multiprocessors).

9-2 Variable Sharing in Multithreaded Environment

Suppose we have a variable V in memory which can be accessed by two threads as shown below (assume single processor CPU):

```
Thread1      //definition of thread 1
  Begin
    ----- //instructions
    -----
    -----
  Load V     //Load V from memory to Accumulator
  Add1       //Add 1 to Accumulator
  Store V    //Store accumulator to V in memory
    ----- //instructions
    -----
    -----
  end        //end of thread1 definition

Thread2      //definition of thread2
  Begin
    -----
    -----
  Load V     //access to shared available V
  Add1       //Modify
  Store V    //Write
    -----
```

```
-----  
end
```

```
Program      //main program  
begin  
    V=20     // initial value of V  
    While true do      //Continuous Looping  
        Threads begin  
            Thread1; //Concurrent execution  
            Thread2; //Concurrent execution  
        Threads end  
    end.        //end of program
```

It is clear from the above program that the function of each Thread is to increment V in each program Loop. Let us suppose that Thread1 and Thread2 can be completed in less than a time slice (Quantum) then after one program Loop the value of V will be 22 which is expected and correct (Single processor CPU is assumed).

Now, let us suppose that thread1 and thread2 can not be completed in a quantum then we may have the following situation:

Thread1 reads V to accumulator, increment accumulator to become 21 and then quantum expires before storing accumulator to V. Thread 2 starts and will also read V, increment accumulator, store accumulator to V and then quantum expires.

This means that V becomes 21. Now, thread1 will be dispatched again for another quantum and continue its task and hence will store its accumulator value to V i.e. store 21. This means that the value of V after Thread1 and Thread2 completed is 21 and not 22 as it should be.

After this example, the variable sharing problem is clear and we have to find solution to it.

9-3 Critical Section, Mutual Exclusion, Primitives

Now, let's define the following terms:

- Critical Section: Part of the thread that modify the shared variable.
- Mutual exclusion: When a thread enters its critical section, it should prevent (exclude) other threads from doing that and hence no errors occur.
- Mutual Exclusion Primitives: These are the statements (instructions) that will be used to enforce the mutual exclusion situation.

The previous program can be rewritten after enforcing mutual exclusion as follows (assume single processor CPU):

Thread1

begin

enter mutual exclusion //set of instructions

critical section1 //Load V, Add1, Store V

exit mutual exclusion //set of instructions

end

Thread 2

Begin

```

-----
-----
enter mutual exclusion //primitive
critical section 2
exit mutual exclusion //primitive
-----
-----
end
Program //No errors
begin
  V=20
  While true do
    Threads begin
      Thread 1;
      Thread2;
    Threads end
  end.

```

- ❖ **Note:** The program is for demonstration purpose assuming single processors CPU. The function of the program is not important.

9-4 Implementation of Mutual Exclusion Primitives

As mentioned above, the mutual exclusion primitives are set of instructions that enforce mutual exclusion.

This means that there are many methods of implementation depending on the type of instructions provided by CPU and on number of concurrent threads. The famous methods are:

- a- Case of no special CPU instructions:
 1. For two threads, there are:

- Decker algorithm
 - Peterson algorithm
2. For n threads, there are:
- Dijkstra algorithm (using semaphores)
 - Other algorithms

b- Case of using special CPU instructions:

If the CPU is designed to provide special instructions suitable for implementing primitives, then other algorithms may be used. These algorithms are, sometimes, called as "hardware algorithms".

9-5 Example of Primitive Implementation

The above algorithms will not be discussed here, however, two implementation examples will be given as follows:

9-5-1 Simple example using normal instructions:

In this example, special CPU instructions are not available and hence normal instructions are used. It should be noted that this is a simple example to show the idea only and hence it can not be considered as a standard algorithm. The implementation has several disadvantages as will be shown later. The example is as follows (assume single processor CPU):

```

Program example;    //mutual exclusion primitives
Var threadnumber: integer;
Thread 1;
begin
  -----
  -----
  -----
  While thread number=2 do;    //wait loop

```

```

Critical section 1; //modify shared variable
Threadnumber=2; //exit mutual exclusion
-----
-----
-----
end;

Thread 2;
begin
-----
-----
While thread number=1 do; //enter mutual exclusion
Critical section 2; //Modify shared variable
Thread-number= 1; //exit mutual exclusion
-----
-----
end;

begin //start of main program
thread number=1; //initialization
V=20 ; shared variable
While true do

Threads begin
Thread1;
Thread2;
Threads end
end. //end of program

```

Studying the above example program, we notice:

- Mutual exclusion is guaranteed
- Execution of critical sections are alternating i.e. if critical section 1 is executed once then it can not be

executed again until critical section 2 is executed. This alternating behavior may lead to "Dead lock" if either threads is terminated for some reason.

9-5-2 Simple Example Using Special Instruction

The implementation of primitives becomes easier if CPU has special instructions such as:

Test and set (a, b) //a, b are Boolean

When executed: b a
 True b

These two operations can not be divided, hence, the quantum can not expire in the middle of them.

An example program, using test and set, is shown below (assume single processor CPU)

Program example; //mutual exclusion by testandset

```

    Var active: Boolean;
    Thread1;
    Var onecannotenter : Boolean;
    begin
        While true do //start looping
            begin
                Enter mutual exclusion primitive
                Onecannoteneter=true;
                While one cannotenter do
                    Testandset (onecannotenter,active);
                Critical section1; //active is true
                Exit mutual exclusion primitive
                active=false;
            end
        end
    end

```

```

        other stuff1
    end
end;

```

```

Thread2:
    Var twocannotenter=boolean;
    begin
        While true do //thred2 looping
            begin
                twocannotenter=true;
                while twocannotenter do
                    testandset (twocannotenter,
                        active);
                    critical section 2; //active is true
                active=false;
                othersuff2;
            end
        end;
    end;

```

```

begin //program start
    active=false;
    threads begin //concurrent execution
        Thread1; //continuous Looping
        Thread2; //continuous looping
    threads end
end. //end of program

```

- ❖ **Note:** In the above examples, we assumed single processor CPU and time slices are dispatched to threads. Also, the example programs are for demonstration purposes and have no useful functions.

End of Chapter9

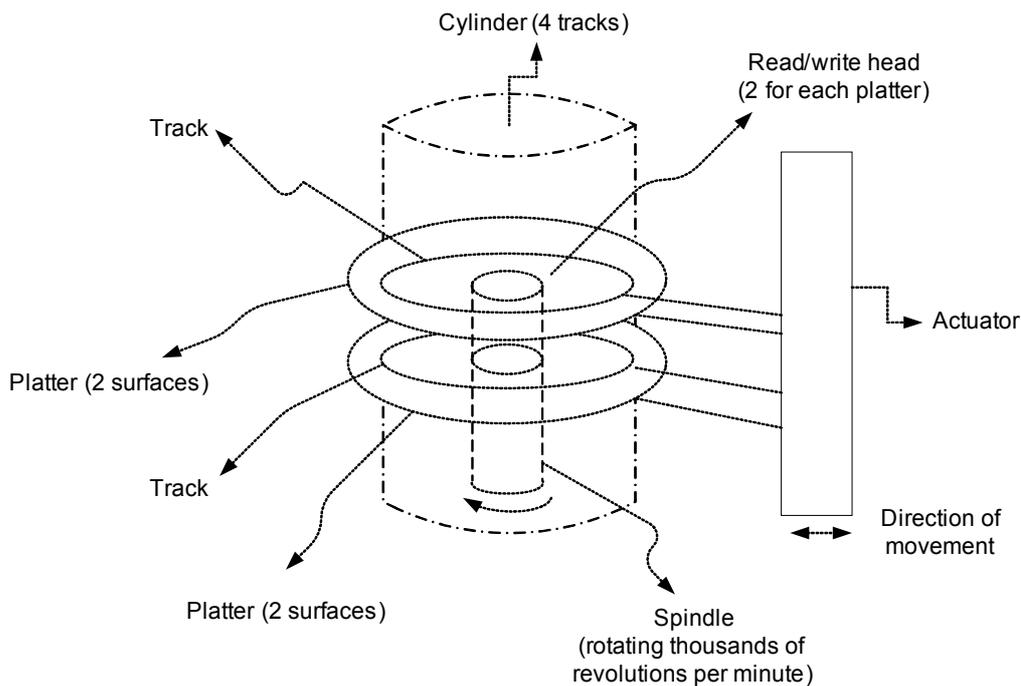
Chapter 10: Disk Performance Optimization

10-1 Introduction

In recent years, processors & memory speeds have increased more rapidly than those of hard disk. As a result, processes requesting data from disk tend to experience long service delay. In this chapter, we discuss how to optimize disk performance by recording disk requests to increase throughput, decrease response time & reduce the variance of response times. We also discuss how OSs reorganize data on disk & exploit buffers & caches to boost performance. Finally, we discuss Redundant Arrays of Independent Disks (RAIDs), which improve disk access times & fault tolerance by servicing requests using multiple disks at once.

10-2 Characteristics of Moving-Head Disk Storage

The general structure of hard disk is shown in fig 10.1. In this figure, we notice the followings:



**Fig 10.1 Disk Structure
(Schematic Side View)**

- The disk storage may consist of several platters & each has a separate read/write moving-head. All heads are fixed to the same actuator & hence move together to select certain cylinder. The cylinder is a set of tracks on all surfaces. Usually, at one time, only one head is active & deals with one track of the whole cylinder. This means that OS has to select the proper head to read/write (r/w) data.
- Each track is divided to several sectors as shown in fig 10.2 each sector is of 512 byte size.

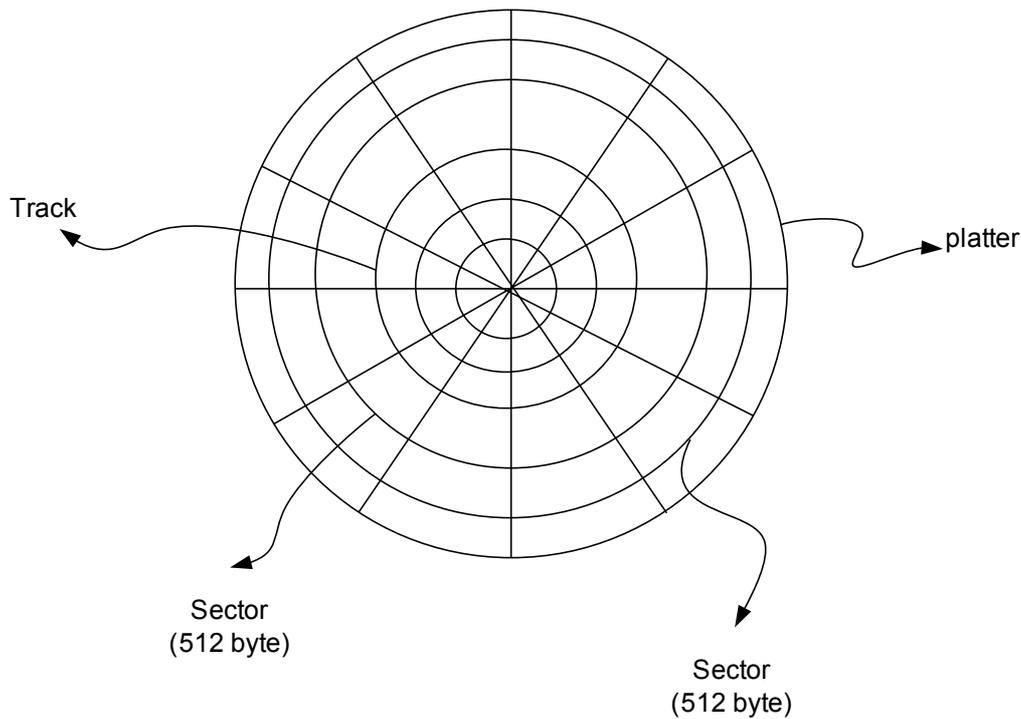


Fig 10.2 Tracks & Sectors of Disk

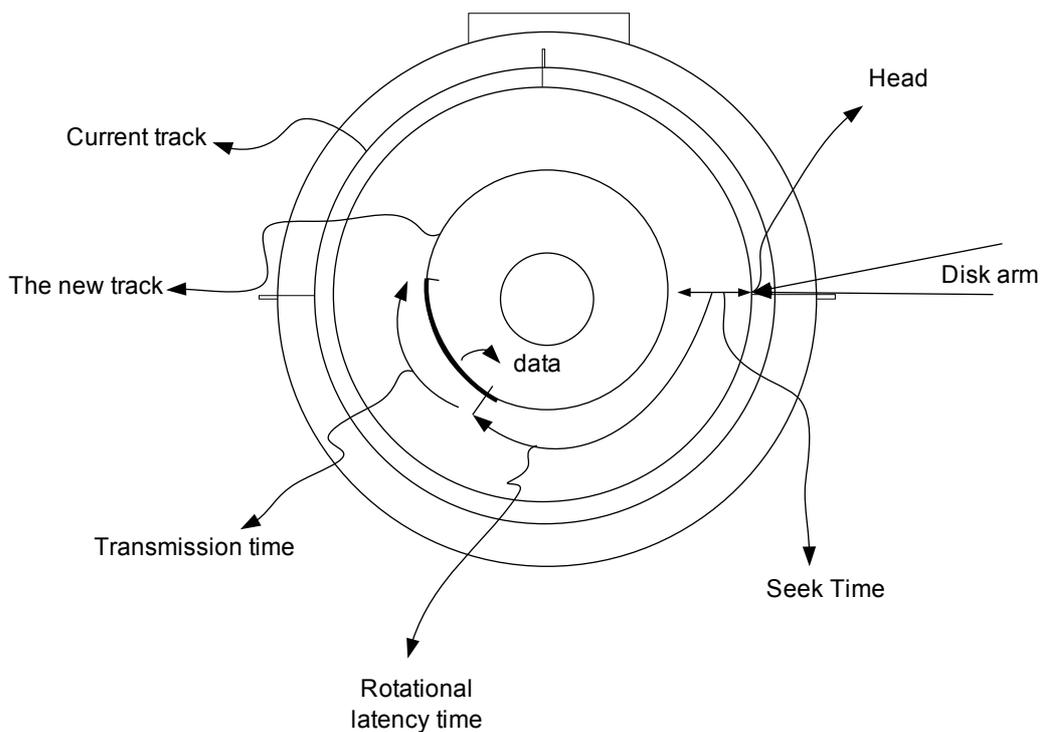
From the above, it is clear that for the OS to R/W data from disk, it needs to:

- 1- Specify the proper surface containing the data & hence the proper moving head.
- 2- Specify the track & sectors containing the data on that surface.
- 3- Instruct actuator to move head to the proper track. This movement takes time which is called "Seek Time" & its average value is in the range of few milliseconds (e.g. 7 msec).
- 4- The platter has to be rotating & the head should wait for the proper sector in the track to get the data. This time depends on revolution speed & its average value is half of one revolution period & is usually of few

milliseconds value (e.g. 4 msec). This time is called "Latency Time".

- 5- When the head is on the proper sector, it starts reading/writing data & also this process takes time depending on number of sectors to be read. This time is called "Transmission Time" as shown in fig 10.3.

From the above, It is clear that a few milliseconds are necessary to R/W data from the disk while the CPU can execute millions of instructions in thatv time.



**Fig 10.3 components of disk access
(Total time of few milliseconds) e.g. 10 msec**

10-3 Why Disk Scheduling is Necessary

Many processes can generate requests for reading & writing data on a disk simultaneously. Because these processes sometimes makes requests faster than they can be serviced by the disk, waiting lines or queues build up to hold disk requests. Some early computing systems simply serviced these request on a "First Come First Served FCFS" basis, in which the earliest arriving request is serviced first. FCFS exhibits a random seek pattern in which successive requests can cause time consuming seeks from the innermost to the outermost cylinders (tracks). To reduce the time spent seeking records, it seems reasonable to reorder the request queue in some manners other than FCFS. This process, called disk scheduling, can significantly improve throughput.

The two most common types of scheduling are "Seek optimizing" & "Rotational Optimizing". Because seek times are usually greater than latency times, most scheduling algorithms concentrate on minimizing total seek time for a set of requests.

10-4 Disk Scheduling Strategies

The strategies are evaluated by the following criteria:

- Throughput: The number of requests serviced per unit time. The maximum number is the better.
- Mean response time: The average time spent waiting for a request to be serviced. The minimum time is the better.
- Variance of response time: The difference between the request waiting time and the

mean response time. The minimum variance is the better.

Here, it is necessary to check the possibility of a request indefinite postponement.

There are many strategies & we shall discuss some of them as follows:

10-4-1 First Come First Served (FCFS) Disk Scheduling

This has been already discussed & it suffers from long seek time & hence low throughput especially under heavy loads.

10-4-2 Shortest Seek Time First (SSTF)

In this strategy, the next request to be serviced is the one that is closest to the R/W head & thus incurs the shortest seek time.

The main problem is the possibility of indefinite postponement for the innermost & outermost tracks especially under heavy loads i.e. many requests are coming all the time.

10-4-3 Scan Disk Scheduling

Here, the disk head moves from the outer track to the inner & then in the opposite direction. The request to be serviced is the one that its track is ahead of the head in the motion direction.

This means that the requests coming in front of the head in the motion direction are serviced first.

The scheduling may suffer indefinite postponement or long waits for requests of innermost and outermost tracks under heavy load.

10-4-4 C-Scan Disk Scheduling

C-Scan mean circular scan & it is similar to SCAN but the head doesn't service requests when moving in the opposite direction i.e. it service requests in only one direction & hence decrease the possibility of indefinite postponement of outside tracks.

10-4-5 Other scheduling strategies

There are also other strategies such as:

- Fscan
- N-Step Scan
- Look Scan
- C-Look Scan
- Shortest Latency Time First (SLTF)
- Shortest Positioning Time First (SPTF)
- Shortest Access Time First (SATF)

10-5 Caching & Buffering

Many systems maintain a "disk cache buffer", which is a region of main memory that the OS reserves for disk data. In one context, the reserved memory acts as cache, allowing processes quick access to data that would otherwise need to be fetched from disk. The reserved memory also acts as a buffer, allowing the OS to delay writing modified data until the disk experiences a light load or until the disk head is in a favorable position to improve I/O performance.

The disk cache buffer presents several challenges to OS designers such as:

- Size of cache buffer
- Replacement strategy
- Inconsistency of data when power or system fail.

Many of today's hard disk drives maintain an independent high-speed buffer cache (on board cache) of several megabytes it's not related to main memory i.e. not part of it (i.e. can't be addressed by CPU directly).

Also, some hard disk controllers (e.g. SCSI, RAID) maintain their own buffer cache (normal RAM) separate from main memory.

ALL buffers are used to enhance the disk performance i.e. increase the speed of data retrieval.

10-6 Redundant Arrays of Independent Disks (RAID)

Previously, we have been discussing non RAID disks that have the following features:

- The disk includes several platters. Each platter has two R/W heads. ALL the heads are mounted on one actuator & hence move together.
- Usually, the OS determines the location of data on which surface of which platter & instructs the proper head to R/W.
- At any one time, only one head is used for reading or writing i.e. it is not possible to make multiple accesses with several heads.

In other words, the disk has multiple heads but only one of them is used at any one time.

-
- The file is usually stored on one surface of one platter only unless it is very large.
 - The only objective of this disk structure is to get large storage volume.

In the RAID structure, the philosophy is completely different from the nonRAID as it has the following features:

- The disk includes several platters & heads as before but each head here has its own actuator and hence can move independently of the other heads. This will enable multiple reads & writes to be carried out at the same time & hence faster disk response.
- The file may be stored on one platter or on several ones & hence it is possible to read/write several parts of the same file at the same time & this means fast R/W.
- Reliability issue is handled here and hence we find out that an error correcting code (ECC) is being used & hence more storage is needed and this is reason for the term "Redundancy". The redundancy will help in correcting errors & hence the RAID system will be "fault tolerant" in this case.

From the above, we conclude that the RAID structure is equivalent to the use of multiple "Independent" disks & therefore we are going to use the word "disk" instead of platter when describing the RAID technology (RAID structure).

There are several methods for using the disks in the RAID system & these methods are identified by the following names: level0, level1, level2, etc.

We are going to discuss some of these levels in a brief way as showing later.

In RAID systems we use the following terms:

- Data Striping: entail dividing data into fixed size blocks called "strips".

Contiguous strips of a file are typically placed on separate disks so that request for file data can be serviced using multiple disks at once, which improves access times (see fig 10.4).

- Stripe: consists of the set of strips at the same location on each disk of the array.
- Fine grained strips: small size strips & this tend to spread file data across several disks & hence reduce access time.
- Coarse grained strips: large size strips & this enable some files, to fit entirely on one strip & hence the access time is as in the Non-RAID system, for that file, however, several requests for several files can be serviced together (Simultaneously).

Notes: The RAID systems (levels) take into consideration the following factors: Access time (Multiple access for one file), fault tolerance, Multiple accesses (for several files on several disks)

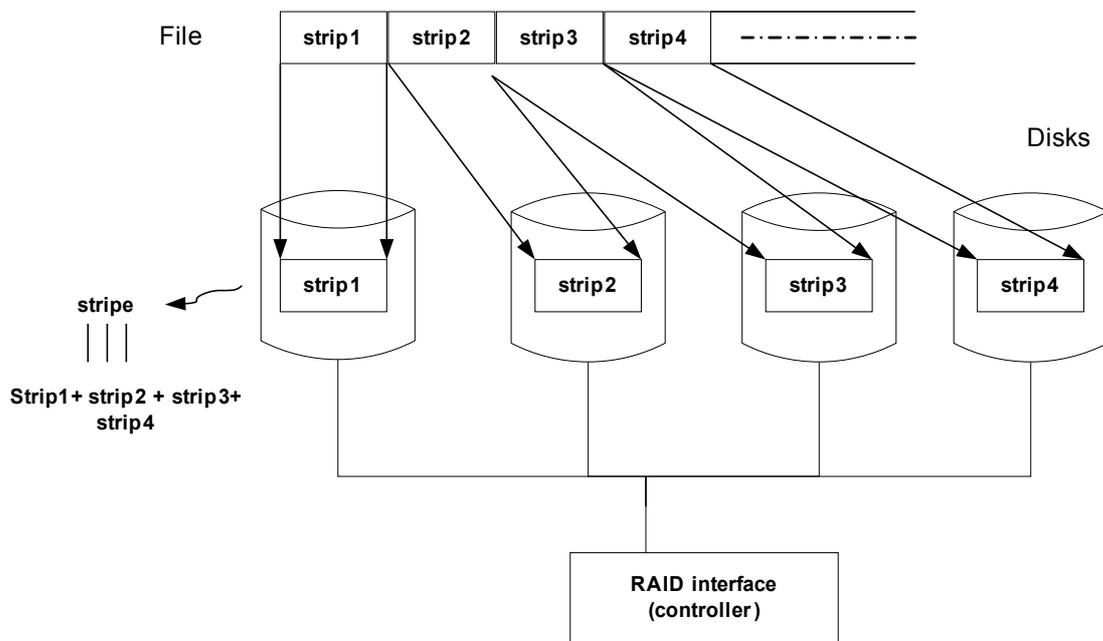


Fig 10.4 Strips & stripe in RAID systems

10-6-1 RAID Level :

RAID level uses a striped disk array with no fault tolerance and hence has no redundancy. The disk contains data only & there is no ECC data.

On level , multiple reads & writes are possible.

The striping level (size) is a block.

∅ Level is sometimes not considered as RAID as it has no redundancy (No ECC).

10-6-2 RAID Level1 (Mirroring)

This level employs disk mirroring (shadowing) to provide redundancy, so each disk in the array is duplicated. Stripes are not implemented in level1 & hence multiple access for the same file is only possible on reading but not on writing. On writing, the same data has to be written on both disks (the original & the mirror) but on reading, 2 different parts of the same file can be read at the same time from the original

& mirror disks. The mirror technology enhances reliability & restricted multiple access but doubling the cost.

10-6-3 RAID Level2

RAID level2 arrays are striped at the bit level, so each strip stores one bit. This means that adjacent bits of file are stored on different disks. Level2 arrays are not mirrored, which reduces the storage overhead incurred by Level1.

The fault tolerance is achieved here by using hamming error correcting codes (hamming ECCs). The error code bits are stored on separate disks (parity disks).

Of course, each stripe of one bit size on data disks has a stripe of one bit size on parity disks. This means that each group of data bits has a corresponding group of ECC bits. The clear problem here is that if the OS wants to write few bits of the group (stripe), it has to read all data stripes first & then modify the necessary data bits & then calculate the ECC bits & at last store the new data & ECC bits. This is called "read-modify-write" cycle.

From the above, we notice that the storage overhead is decreased compared to mirror system but the multiple access for several files is not possible as all disks will be occupied for one file request (remember that the adjacent file bits are distributed among the disks, also, it is necessary to read the parity disks).

- ✓ **Note:** In hamming, we can correct one data bit error. The number of ECC bits are as follows:

<u>Number of data bits</u>	<u>number of ECC bits</u>
4	3
11	4
26	5

It is clear that the larger data stripe, the better & hence the more data disks in the array (stripe) are the better.

10-6-4 RAID Level3

RAID level3 stripes data at the bit or byte level but use parity checks for fault tolerance instead of Hamming. In parity check, we use only one bit (even or odd parity) & this bit does not locate the place of error (as incase of Hamming) but indicates only its existence. When the error occurs, the OS will inform the user immediately who has to find out the erroneous disk and replace it. The data on the faulty disk can be regenerated automatically with the help of other data disks & the parity disk. The advantages of such system:

- 1- Large storage.
- 2- Fault tolerance with one extra disk (one parity disk).
- 3- Multiple access for one file is possible and hence fast access time.

Of course, multiple access for several files is not possible as any file request will occupy all of the disks.

10-6-5 RAID Level4

RAID Level4 systems are striped using fixed size blocks (typically much larger than a byte) & use one disk for parity (even or odd parity). The difference with level3 is that the

file may occupy fraction of disks & not all of them (remember the coarse grained stripes) & hence multiple requests for multiple files may be possible at the same time. Here, we should remember that when reading data from disks, it is not always necessary to read parity bits as these bits are stored not for error detection but mainly for error correction (of one bit -usually one disk may be faulty-).

- ✓ **Note:** Multiple writes is not possible because the parity disk will be occupied for one write.

10-6-6 RAID Level5

RAID Level5 arrays are striped at the block level & a parity check (even or odd) is used like in level4. The difference with level4 is that the parity bits are not located on one disk but distributed throughout the arrays of disks. This means that disk1 carries parity bit1, disk2 carries parity bit2, & so on. The advantage of this level is that multiple writes (writes for several files) are possible because the parity bits are not stored on one disk as the case of level4.

10-6-7 Other RAID Levels

There are other levels such as:

- RAID Level6
- RAID Level10+1
- RAID Level10
- RAID Level10+3, 0+5, 50, 1+5, etc.

10-6-8 comparison of RAID Levels

The properties of different RAID levels can be summarized as follows:

From multiple files



RAID Level	Read concurrency	Write concurrency	Redundancy	Striping Level
0	Yes	Yes	None	Block
1	Yes	No	Mirroring	None
2	No	No	Hamming ECC	Bit
3	No	No	Even or odd parity	Bit/Byte
4	Yes	No	Even or odd parity	Block
5	Yes	Yes	Distributed even or odd parity	Block

✓ **Notes:**

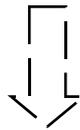
1- When striping level is bit or byte. This means that the file data are distributed on all the disk arrays & hence it is not possible to service multiple requests for several files, however, the single file will be read/write very quickly i.e. fast access (fast data transmission).

When block level is used then the file may occupy few disks of the array (stripe) & the others may be for other file, & hence multiple files may be serviced.

2- The main purpose of redundancy is not to detect errors but to correct it & hence, it is not always necessary to read parity disks during read cycle & this allows multiple read for several files when block (coarse grained striping) striping is used.

3- Hamming ECC is not necessary as the faulty disk may be discovered by other means & hence one parity (even or odd) bit is enough for error detection & correction. In other words, the parity bit will inform us about the existence of error and then by other means (electrical, mechanical) we can find out the faulty disk & hence regenerates its data from knowing the other data bits on the data disks & the parity bit from the parity disk.

- ✓ **Note:** in RAID, we discover that there are errors by using parity check. Then by some means we find the faulty disk & then we regenerate the data on the faulty disk by making parity of all other data disks (except the faulty one) & of the parity disk itself.



Conclusions:

The parity check can be used to correct data if the erroneous bit location is known.

Once we know this location, we can find the missing bit by making parity of all other data bits & the parity bit.

In RAID, we know location by knowing the faulty disk by different means & this starts by finding parity check error.

From the above we notice that RAID systems features are:

-
- Large storage volume as it uses arrays of disk.
 - Fast data transmission as many heads work together (Independent Disks).
 - Fault tolerance with low overhead storage by using parity check (redundancy).

End of Chapter 10

Chapter eleven: Deadlock and Postponement

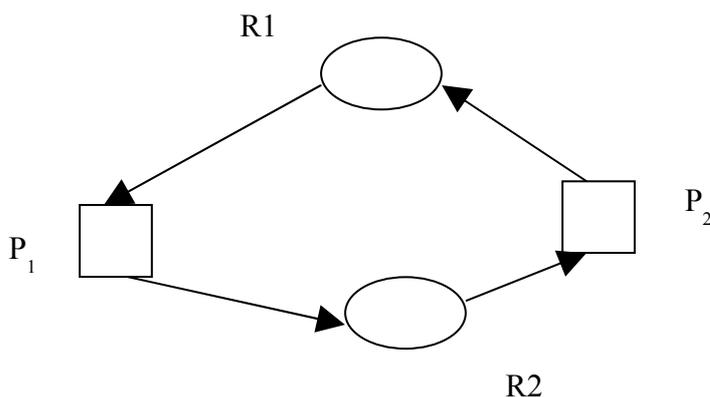
-One problem that arises in multiprogrammed systems is deadlock. A process or thread is in a state of deadlock if it is waiting for a particular event that will not occur.

-In multiprogrammed computing systems, resources sharing are one of the primary goals. When resources are shared among a set of processes, each process maintaining exclusive control over particular resources allocated to it, deadlocks can develop in which some processes will never be able to complete execution, the result can be loss of work and reduced system throughput and system failure.

11.1- Examples of Deadlock

Simple Resource Deadlock

-Most deadlocks in operating system develop because of the normal contention for dedicated resources (resources that may be used over time by many processes but only one process at a time, sometimes called serially reusable resources).



R₁ is allocated to P₁
R₂ is allocated to P₂
P₁ is requesting R₂
P₂ is requesting R₁

Resource allocation graph

-P₁ holds R₁ and needs R₂ to continue.

-P₂ holds R₂ and needs R₁ to continue.

Each process is waiting for the other to free a resource that the other process will not free.

This is circular wait is characteristic of deadlocked systems.

Deadlock in Spooling Systems

-A spooling system improves system throughput by disassociating from slow devices.

-To speed the program's execution, a spooling system routes output pages to a much faster device (hard disk) where they are temporarily stored until they can be printed.

-Some spooling systems require that the complete output from a program be available before printing can begin. Several partially completed jobs generating pages to a spool file can become deadlocked if the disk's available space fills before any job completes.

-The user or system administrator may kill one or more jobs to make sufficient spooling space available for the remaining jobs to complete.

-One way to make deadlocks less likely is to provide more space for spooling files than is to be needed.

-A more common solution is to restrain (hold down) the input spoolers so that they do not accept additional print jobs when the spooling files begin to reach some saturation threshold. This may reduce system throughput but it is the price paid to reduce the likelihood of deadlock.

-Today's systems might allow printing to begin before the job is completed. Spooling file can begin emptying while a job is still executing. (Play video film before download fully)

-In many systems spooling space allocation has been made more dynamic, so that if existing space starts to fill, then more space may be made available.

11.2- Related Problem

-In any system that requires processes to wait as a result of resource-allocation and process scheduling decisions. A

process may be delayed indefinitely while other processes receive the system's attention. This situation is called indefinite postponement, indefinite blocking or starvation, can be as devastating (overwhelming) as deadlock.

- Indefinite postponement may occur because of biases in a system's resource-scheduling policies. When resources are scheduled on a priority basis, it is possible for a given process to wait for a resource indefinitely, as processes with higher priorities continue to arrive.

-Some systems prevent indefinite postponement technique is called aging. The waiting process's priority will exceed the priorities of all processes and then it will be serviced.

11.3- Resource Concepts

-Resources that is preemptible such as processors and main memory. Processors are the most frequently preempted resources on a computer system.

-Processors must be rapidly switched among all active processes competing for system service to ensure that these processes progress at a reasonable rates.

-A user program currently occupying a particular range of locations in main memory may be removed or preempted by another program.

-Certain resources are non-preemptible; they can not be removed from the processes to which they are assigned until the processes voluntarily release them (scanner).

-Some resources may be shared among several processes, while others are dedicated to a single process at a time.

-Some resources may be shared among several processes, while others are dedicated to a single process at a time.

-If the operating system maintained in main memory a separate copy of the editor for each program, these would be a significant amount of redundant data and wasting memory.

-A better technique is for the operating system to load one copy of the code in memory and to make the copy available to each user.

-If a process were allowed to modify this shared code, as a result, this code must be reentrant meaning the code is not modified as it executes.

-Code that may be changed but is reinitialized each time it is used is said to be serially reusable.

-Reentrant code may be shared by several processes simultaneously, whereas serially reusable code may be used correctly by only one process at a time.

11.4- Four Necessary Conditions for Deadlock

- A resource may be acquired exclusively by only one process at a time (mutual exclusion)
- A process that has acquired an exclusive resource may hold that resource while the process waits to obtain other resources (wait-for-condition, also called the hold-and-wait condition)
- Once a process has obtained a resource, the system cannot remove it from the process's control until the process has finished using the resource (non-preemption condition)
- Two or more processes are locked in a circular chain, in which each process is waiting for one or more resources that the next process in the chain is holding (circular-wait condition)

Taken together, all four conditions are necessary and sufficient for deadlock to exist. If they are all in place, the system is deadlocked.

11.5- Deadlock Solutions

-**Deadlock prevention:** to remove any possibility of deadlock occurring. Prevention is a clean solution as far as deadlock itself is concerned, but prevention methods can often result in poor resource utilization.

-**Deadlock avoidance:** less conditions to get better resource utilization. Avoidance methods do not precondition the system to remove all possibility of deadlock. Instead they allow the possibility to loom (appear), but whenever a deadlock is approached, it is carefully sidestepped.

-**Deadlock detection:** methods are used in which deadlocks can occur. The goal is to determine if a deadlock has occurred, and to identify the processes and resources that are involved.

-**Deadlock recovery:** methods are used to clear deadlocks from a system so that it may operate free them, and so that the deadlocked processes may complete their execution and free their resources. Recovery requiring that one or more of the deadlocked processes be flushed from the system. The flushed processes are normally restarted from the beginning when sufficient resources are available.

11.6- Deadlock Prevention

-Each process must request all its required resources at once and cannot proceed until all have been granted.

-If a process holding certain resources is denied a further request, it must release its original resources and if necessary request them again together with additional resources.

-A linear ordering of resources must be composed on all processes, i.e., if a process has been allocated certain resources, it may subsequently (after) request only those resources later in the ordering.

11.6.1- Denying the "wait-for" condition

-The first strategy requires that all of the resources a process needs to complete its task must be requested at once.

-The system must grant them on an all or none basis.

-If all the resources needed by a process are available, then may grant them all to the process at once and the process may continue to execute.

-If they are not all available, then the process must wait until they are, while the process waits, it may not hold any resources.

-Thus the wait-for condition is denied, and deadlocks cannot occur, but it wastes resources.

-For example: a program executes 4 tape drives at one point in its execution must request and receive all 4 before it begins executing.

- If all 4 drives are needed throughout the execution of the program, then there is no serious waste.
- If the program needs only one tape to begin execution and does not need the remaining tape drives for several hours, means that substantial resources will sit idle for several hours.

-To get better resource utilization:

- Divide a program into several threads that run relatively independently of one another. Then resource allocation can be controlled by each thread rather than for the entire process. This can reduce waste but involves a greater overhead in application design and execution. This cause indefinite postponement.
- One way to avoid this is to handle the needs of the waiting processes in first-come-first-served order. (wasting of resources, or if gradually accumulating means sit idle, or the user should pay to get quick

service but this would destroy the predictability of resource charges)

11.6.2- Denying the "no-preemption" condition

-Suppose a system does allow processes to hold resources while requesting additional resources. As long as sufficient resources remain available to satisfy all requests, the system cannot deadlock.

-A process hold resources that a second process may need in order to proceed, while the second process may hold resources needed by the first process-a two process deadlock.

-When a process holding resources is denied to request for additional resources, it must release the resources it holds and if necessary, request them again together with the additional resources.

-This denies the no-preemption condition-resources can indeed be removed from the process holding them prior to the completion of that process.

-When a process releases resources, it may lose all of its work to that point (high price). If this occurs infrequently, then this strategy provides relatively low-cost means of preventing deadlocks.

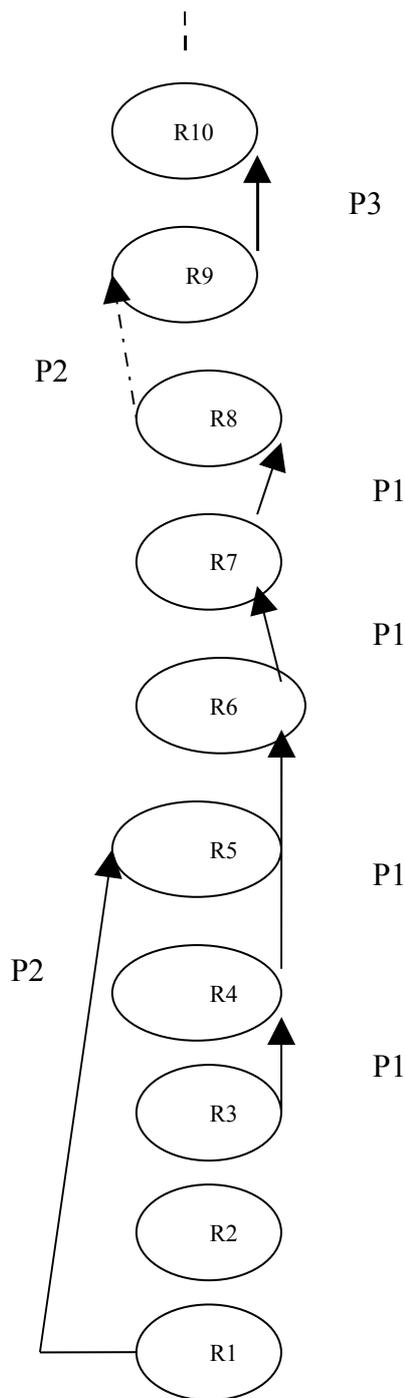
-If the system favors processes with small resource requests over those requesting substantial (considerable) resources that alone could lead to indefinite postponement. As a process requests additional resources, this strategy requests the process to give up all the resources it has and request even larger number.

-So indefinite postponement can be a problem in a busy system. Also this strategy requires all resources to be preemptible, which is not always the case (e.g. printers should not be preempted while processing a print job).

11.6.3- Denying the Circular-Wait Condition

-In this strategy assigning a unique number to each resource that the system manages, and creating a linear ordering of resources. A process must then request its resources in a strictly ascending order.

-If a process request R_3 (3 is the resource number) then it can request only resources with a number greater than 3. Because all resources are uniquely numbered and because processes must request resources in ascending order, a circular wait cannot develop.



P1 has obtained R3,
R4, R6, R7, R8
No circular wait can
develop because all
arrows must point
upward

-One disadvantage is that resources must be requested in ascending order by resource number. If new resources are added or old ones removed at an installation, existing programs and systems may have to be rewritten.

-Another difficulty is determining the ordering of resources in a system. The resource numbers should be assigned to reflect the order in which most processes actually use the

resources. For processes matching this ordering, more efficient operation may be expected, but for processes that need the resources in a different order resources must be acquired and held possibly for long periods of time, before they are actually used. This can result in poor performance.

-This strategy truly eliminates the possibility of a circular wait, yet it diminishes a programmer's ability to freely and easily write application code that will maximize an application's performance.

11.7- Deadlock Avoidance with Dijkstra's Banker's Algorithm

-It defines how a particular system can prevent deadlock by carefully controlling how resources are distributed to users.

-A system groups all the resources it manages into resource types. Each resource type corresponds to resources that provide identical functionality (Manage only one type of resource). The algorithm prevents deadlock in the operating system that exhibit the following properties:

- The operating system shares a fixed number of resources, t , among a fixed number of processes, n .
- Each process specifies in advance the maximum number of resources that it requires to complete its work.
- The operating system accepts a process's request of that process's maximum need does not exceed the total number of resources available in the system, t , (the process cannot request more than the total number of resources available in the system).
- Sometimes, a process may have to wait to obtain an additional resource, but the operating system guarantees a finite wait time.
- If the operating system is able to satisfy a process's maximum need for resources, the process guarantees

that the resource will be used and released to the operating system within a finite time.

-The system is said to be in a safe state if the operating system can guarantee that all current processes can complete their work within a finite time. If not then the system is said to be in an unsafe state.

-Four terms that describes the distribution of resources among processes:

- Let $\max(P_i)$ be the maximum of resources that P_i requires during its execution ($\max(P_3)=2$).
- Let $\text{loan}(P_i)$ represents P_i 's current loan of a resource, where its loan is the number of resources that process has already obtained from the system $\text{loan}(P_3) = 4$ means the operating system allocate four resources.
- Let $\text{claim}(P_i)$ be the current claim of a process where a process's claim is equal to its maximum need minus its current loan. $\max(P_3)=6$, $\text{loan}(P_3)=4$ then $\text{claim}(P_3)=\max(P_3)-\text{loan}(P_3)=6-4=2$
- Let a be the number of resources still available for allocation. This equivalent to the total number of resources (t) minus the sum of the loans to all the processes in the system i.e.: $a = t - \sum_{i=1}^n \text{loan}(P_i)$

3 processes, 12 resources, where P_1 has 2 resources, P_2 has 1 resource, P_3 has 4 resources then the number of available resources, $a=12-(2+1+4) =5$

11.7.1- Example of a Safe State

A system has 12 resources, 3 processes sharing the resources

Process	max(P_1) maximum need	loan(P_1) current loan	claim(P_1) current claim
P_1	4	1	3
P_2	6	4	2
P_3	8	5	3

$$a=12-10=2$$

-This state is safe because it is possible for all three processes to finish.

- P_2 has 4 and need 2.

-System has 12, 10 are in use and 2 are available. If the system allocates these 2 to P_2 then P_2 can run to completion and release 6 resources, enabling the system to allocate them to P_1 and P_3 enabling both to finish.

11.7.2- Example of Unsafe State

A system has 12 resources, 3 processes sharing the resources

Process	max(P_1) maximum need	loan(P_1) current loan	claim(P_1) current claim
P_1	10	8	2
P_2	5	2	3
P_3	3	1	2

$$a=12-11=1$$

- P_1 requests and is granted the last available resource. A three-way deadlock could occur if indeed each process needs to request at least one more resource before releasing any resources to the pool.

-An unsafe state does not unify the existence of deadlock, nor even that deadlock will eventually occur what an unsafe state does imply is simply that some unfortunate sequence of events might lead to a deadlock.

11.7.3- Example of Safe-State-to-Unsafe-State Transition

-The resource-allocation policy must carefully consider all resource requests before granting them or a process in safe state could enter an unsafe state.

-For example suppose the current state of a system is safe as the following:

A system has 12 resources, 3 processes sharing the resources

Process	max(P_1) maximum need	loan(P_1) current loan	claim(P_1) current claim
P_1	4	1	3
P_2	6	4	2
P_3	8	5	3

The current value of a is 2.

P_3 requests a resource.

If the system grants this request then the new state would be as in the following:

Process	max(P_1) maximum need	loan(P_1) current loan	claim(P_1) current claim
P_1	4	1	3
P_2	6	4	2
P_3	8	6	2

The current value of a is 1 which is not enough to satisfy the current claim of any process, so the state is now unsafe.

11.7.4- Banker's Algorithm Resource Allocation

-The mutual exclusion, wait-for and no-preemption conditions are allowed. Processes are indeed allowed to hold resources while requesting and waiting for additional resources, and resources may not be preempted from a process holding those resources. Processes claim exclusive use of the resources they require.

-Processes ease onto the system by requesting one resource at a time. The system may either grant or deny each request. If a request is denied, then the process holds any allocated resources and waits for a finite time until that request is eventually granted.

-The system grants only requests that result in safe states. Resource requests that would result in unsafe states are repeatedly denied until they can eventually be satisfied.

11.7.5- Weaknesses in the Banker's Algorithm

-It requires that there be a fixed number of resources to allocate, but we cannot count on the number of resources remaining fixed (number of resources vary dynamically).

-It requires that the population of processes remains fixed but this is unreasonable; the process population is constantly changing.

-It requires that the banker (banker) grant all requests within a finite time (much better is needed in real-time system).

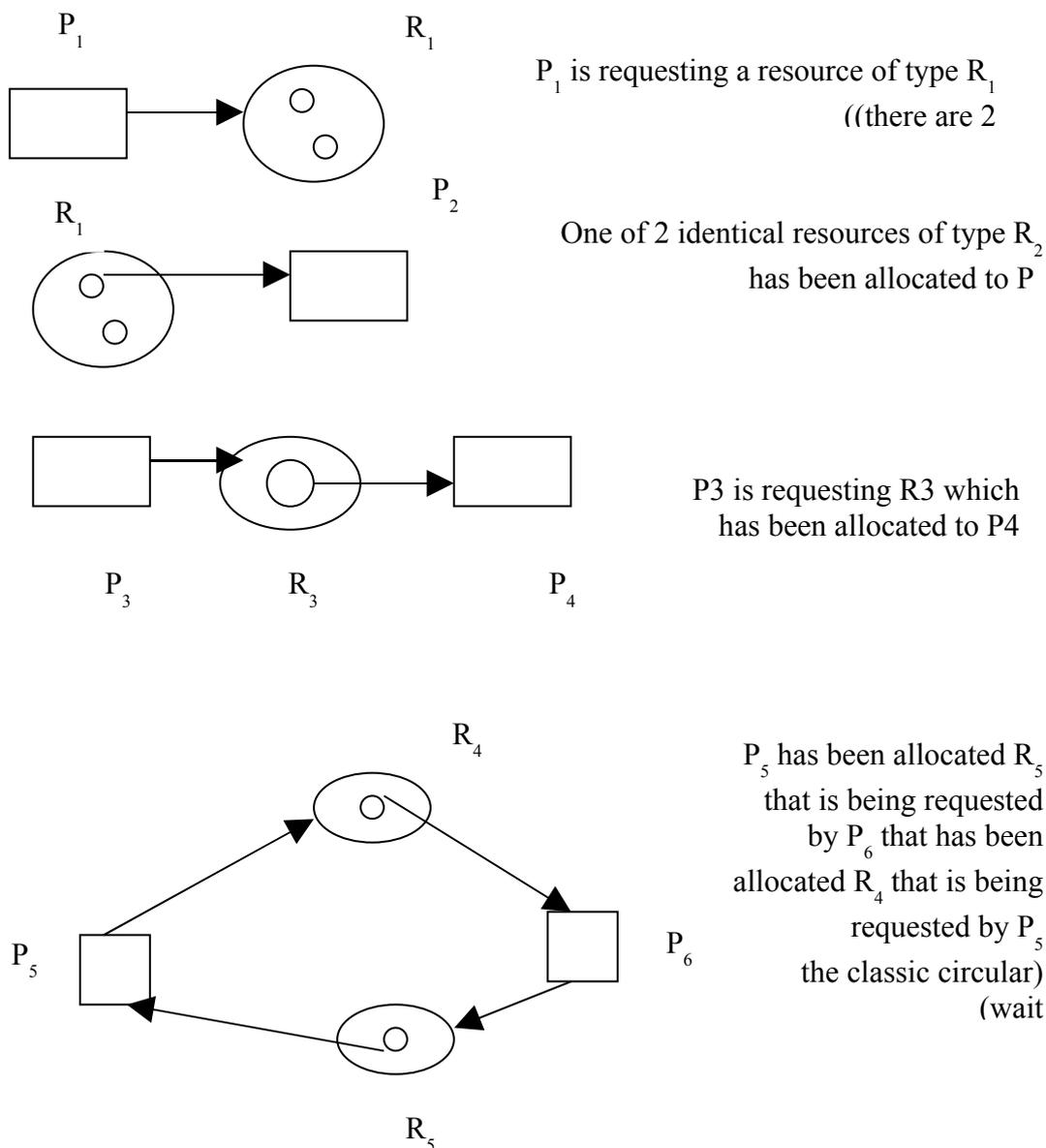
-It requires that processes state their maximum needs in advance-but resource allocation becoming increasingly dynamic, it is difficult to know the maximum needs.

11.8- Deadlock Detection

-Deadlock detection is the process of determining that a deadlock exists and identifying the processes and resources involved in the deadlock.

-Deadlock detection algorithms generally focus on determining if a circular wait exists, given that the other necessary conditions for deadlock are in place.

11.8.1- Resource-Allocation Graphs



A directed graph indicates resource allocations and requests. Resource-allocation and request graphs change as processes

request resources acquire them and eventually release them to the operating system.

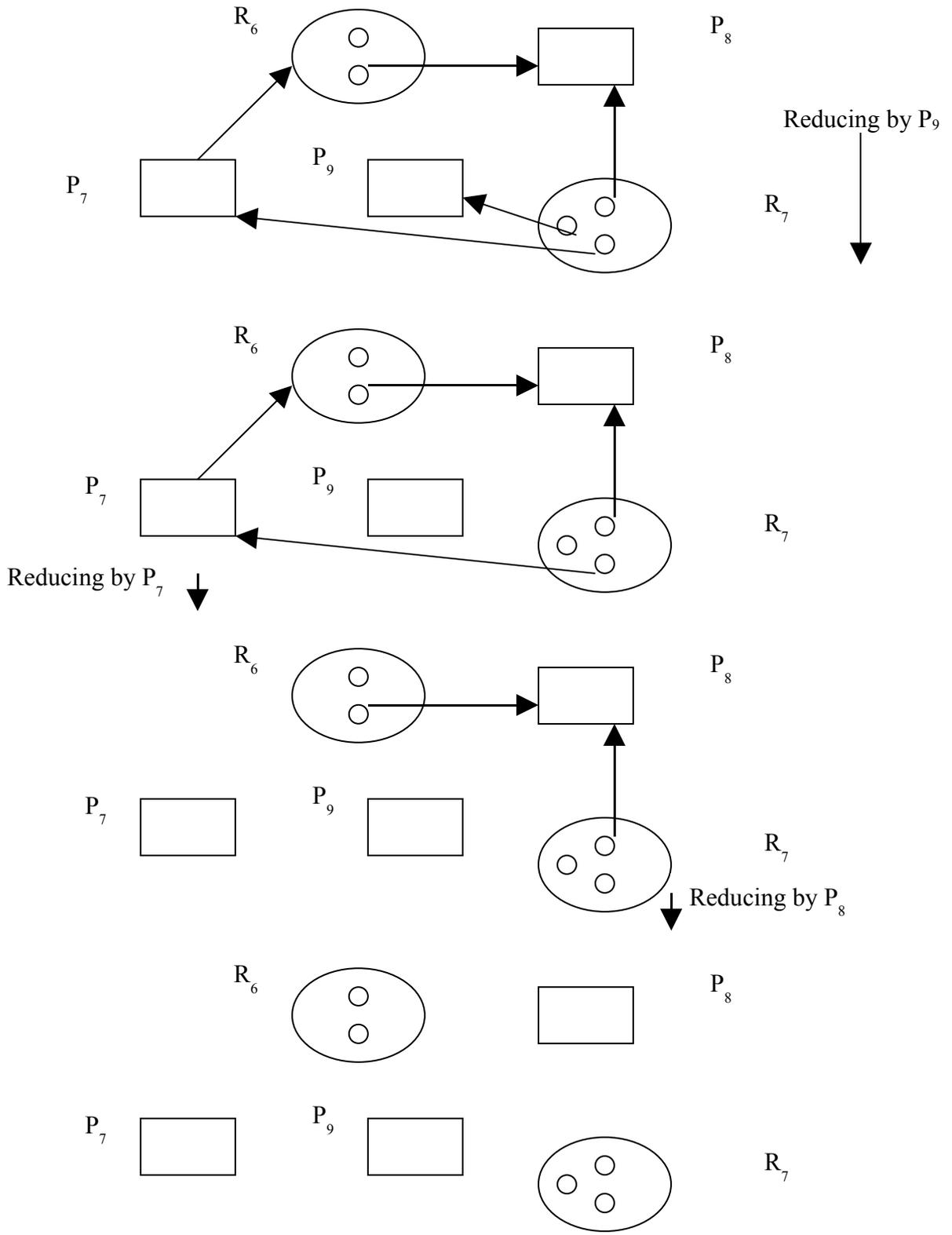
11.8.2- Reduction of Resource-Allocation Graphs

-One technique useful for detecting deadlocks involves graph reductions, in which the processes that may complete their execution. If any processes (and their resources) that will remain deadlocked, if any, are determined.

-If a process's resource requests may be granted, then we say that a graph may be reduced by that process. This reduction showing how the graph would look if the process was allowed to complete its execution and return its resources to the system.

-The graph is reduced by removing the arrows to that process from resources that allocated to that process and from that process to resources requests of that process. If a graph can be reduced by all its processes then there is no deadlock.

-If a graph cannot be reduced by all its processes then the irreducible processes constitute the set of deadlocked processes in the graph. The order in which the graph reductions are performed does not matter. The final result will always be the same.



End of OS_