



Operating System



Published By:



Physics Wallah

ISBN: 978-93-94342-39-2

Mobile App: Physics Wallah (Available on Play Store)



Website: www.pw.live

Email: support@pw.live

Rights

All rights will be reserved by Publisher. No part of this book may be used or reproduced in any manner whatsoever without the written permission from author or publisher.

In the interest of student's community:

Circulation of soft copy of Book(s) in PDF or other equivalent format(s) through any social media channels, emails, etc. or any other channels through mobiles, laptops or desktop is a criminal offence. Anybody circulating, downloading, storing, soft copy of the book on his device(s) is in breach of Copyright Act. Further Photocopying of this book or any of its material is also illegal. Do not download or forward in case you come across any such soft copy material.

Disclaimer

A team of PW experts and faculties with an understanding of the subject has worked hard for the books.

While the author and publisher have used their best efforts in preparing these books. The content has been checked for accuracy. As the book is intended for educational purposes, the author shall not be responsible for any errors contained in the book.

The publication is designed to provide accurate and authoritative information with regard to the subject matter covered.

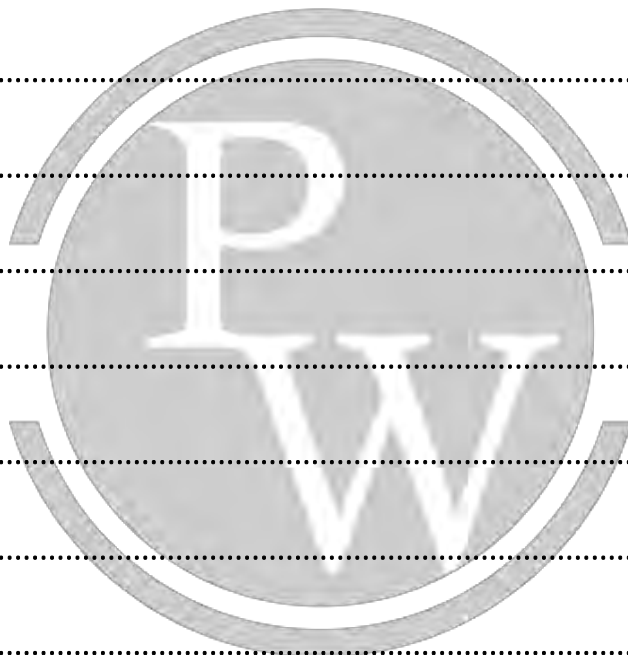
This book and the individual contribution contained in it are protected under copyright by the publisher.

(This Module shall only be Used for Educational Purpose.)

Operating System

INDEX

1.	Introduction and Background of OS	9.1 – 9.3
2.	Process Concepts	9.4 – 9.6
3.	CPU Scheduling	9.7 – 9.10
4.	Multithreading.....	9.11 – 9.12
5.	Synchronization	9.13 – 9.26
6.	Deadlock	9.27 – 9.29
7.	Memory Management	9.30 – 9.35
8.	Virtual Memory	9.36 – 9.38
9.	File Systems	9.39 – 9.42
10.	Disk Scheduling.....	9.43 – 9.44



1



INTRODUCTION AND BACKGROUND OF OS

1.1 What is OS

Operating system is an interface between the user applications and the computer hardware to develop and execute programs.

1.2 Goals of OS

- The primary goal of the operating system is to provide an easy-to-use environment to the user.
- The secondary goal of the operating system is the efficient use of the computer resources. (Operating system is also called as the resource allocator)
- Modularity
- Abstraction
- Ease of debugging

1.3 Functions of OS

- Process Management
- Memory management
- Resource Allocation
- File systems management
- Protection and Security

1.4 Types of OS

1.4.1. Batch Operating System

- Jobs with similar requirements are grouped into batches by the operating system.
- The idea is to execute the jobs onto the CPU one at a time. Another job cannot occupy the CPU time until the previous job has completed execution.
- Increased CPU idleness.



- Decreased throughput of the system.

1.4.2 Multi-programmed/multi-tasking operating system

- Multiple programs/jobs reside in the main memory for execution. The operating system selects and executes one of these jobs on to the CPU.
- If the job in execution requires an I/O operation, another job which is ready for execution is scheduled on the CPU.
- Increased CPU utilization.
- Increased throughput of the system.
- Multi-tasking is a logical extension of multi-programming systems. The jobs are executed on the CPU in time sharing mode.
- The main advantage of multi-tasking systems is good response time.

1.4.3 Real time operating system

- It has well-defined and fixed time constraints.
- Processing of the programs must be done in the defined constraints or the system will fail.
- They are categorized into-hard and soft real time systems.

1.4.4 Distributed operating system

- A distributed operating system handles jobs that are executed by multiple processors networked to each other.
- They are also termed as loosely coupled systems.
- The advantages of distributed systems include resource sharing, computation speed up and reliability.

1.5 Dual Mode Operations

A processor supports two modes of instruction execution:

- User mode/ non-privileged mode
- Kernel mode/ Privileged mode/ Monitor mode

The primary goal of the dual mode operation is to provide protection and security to the user application programs and the operating system from the unauthorized users. The mode bit is used to determine the particular mode in which an instruction is executing.

Mode bit: 0	Kernel mode
Mode bit: 1	User mode

- A process running in kernel mode has direct access to the hardware and full access to the machine instruction set. The operating system always runs in kernel mode.
- Other examples of privileged instructions include I/O operations, context-switching, clearing the memory map etc.



- A process running in user mode has no access to the hardware and limited access to the machine instruction set.
- Other examples of non-privileged instructions include reading the time of the clock, reading the status of the processor, generate trap etc.

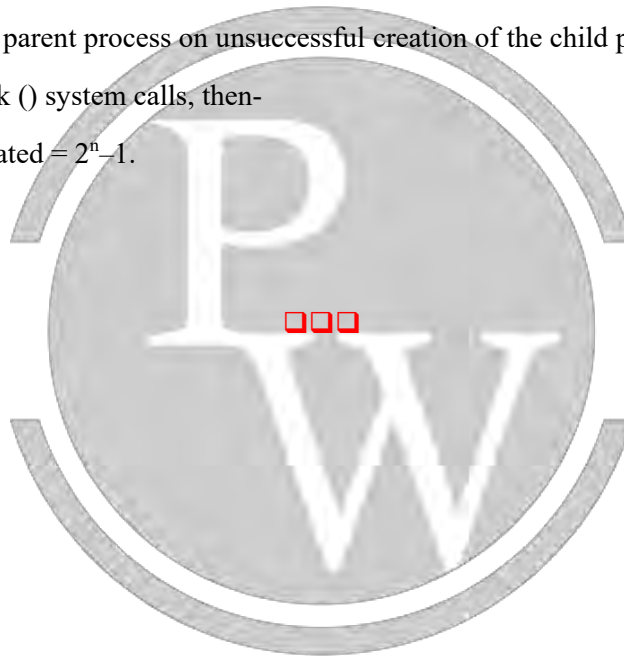
1.6 System Call

- System calls provide the services of the operating system to the user application programs.
- They act as entry points into the kernel system.

1.6.1 Fork () System Call

- The fork () system call is used to create the child processes.
- It returns 0 to the newly created child process.
- It returns the process id of the child process to the parent process on successful creation of the child process.
- It returns negative value to the parent process on unsuccessful creation of the child process.
- If the program contains 'n' fork () system calls, then-

Number of child processes created = $2^n - 1$.



2

PROCESS CONCEPTS

2.1 Program Versus Process

Program	Process
Program is a set of instructions and data.	A program under execution is called a process.
It resides in the secondary memory.	It resides in the main memory.
It is passive.	It is active and dynamic.
It is not allocated any resources.	The operating system allocates resources to the process for its execution.

2.2 Process as ADT

The process image can be described as:

Process		
Process attributes (Process Control Block)	}	Context
Run time Stack		
Dynamic data	}	User address space
Static data		
Code section		

Any process has the following attributes:

- Process identification information
- Priority
- Process state information
- Program counter
- Memory limits
- List of files
- List of open devices
- Protection information

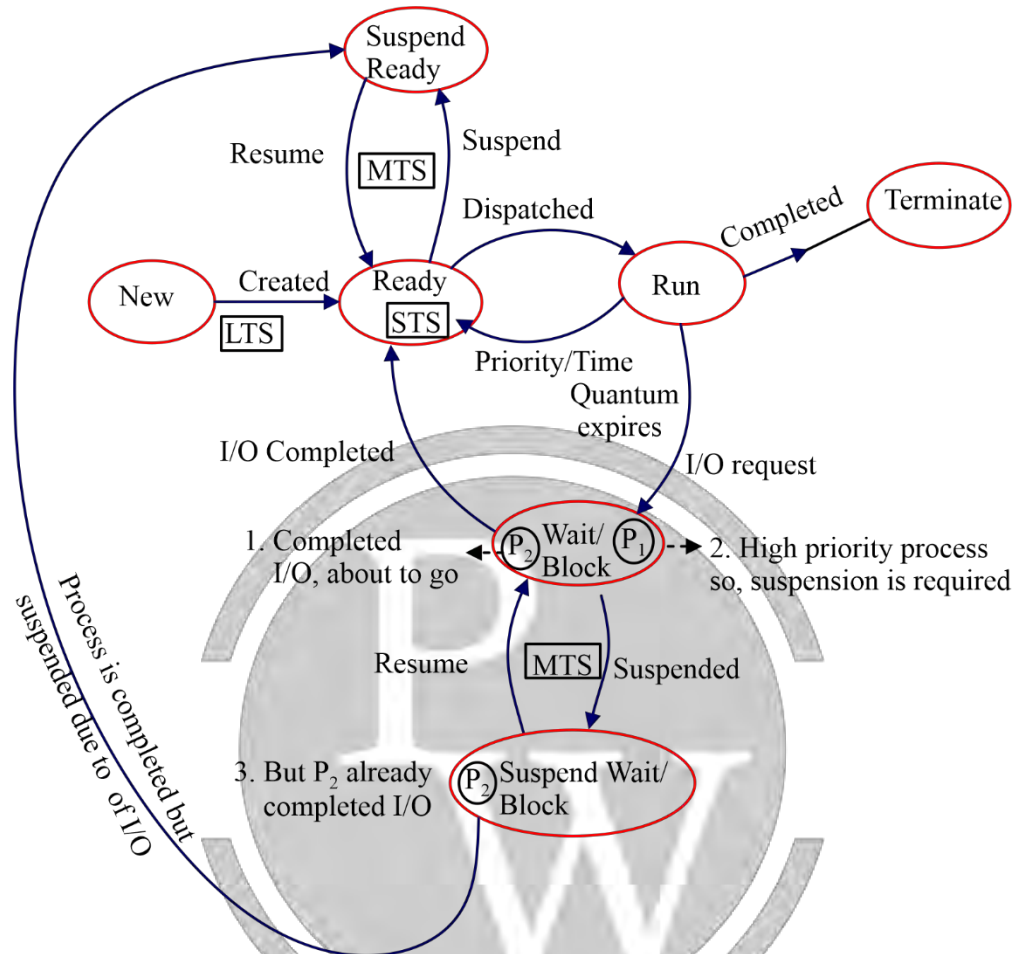
All the attributes of the process are stored in the Process Control Block (PCB). The features of Process Control Block are as follows:

- Every process has its own process control block.
- The process control blocks of all the processes are stored in the main memory.
- They are implemented using **double linked list** data structure.

The **context** of a process includes the process attributes and the stack information.

2.3 Process State Transition Diagram

A process passes through multiple states in its lifetime as follows:



- When the process is present in the Ready, Run or Wait state, it still resides in the main memory.
- When the process is present in the Suspend Ready or Suspend Block state, it resides in the secondary memory.
- There can be multiple processes present simultaneously in the Ready, Wait, Suspend Ready and Suspend Wait states.
- In the Run state, only one process can exist at a time.

2.4 Schedulers

The operating system deploys three kinds of schedulers:

- Long-term scheduler.
- Medium-term scheduler.
- Short-term scheduler.

2.4.1 Long-term scheduler

- It is responsible for the creation and bringing of new processes into the main memory.
- It controls the degree of multi-programming.
- It should generate a good mix of CPU and I/O bound process for efficient resource utilization.
- It is involved in the New → Ready state transition.

2.4.2 Medium-term scheduler

- The medium-term scheduler acts as the swapper by doing swap-out (suspending the process from the main to secondary memory) and swap-in (resuming the process by bringing it from the secondary to main memory) operations.
- It is involved in Ready \Rightarrow Suspend Ready and Block \Rightarrow Suspend Block state transitions.

2.4.3 Short-term scheduler

- The short-term scheduler selects a process from the ready state to be executed.
- It is involved in the Ready \rightarrow Run state transition.

2.5 Dispatcher

- The dispatcher is responsible for loading the job (selected by the short-term scheduler) onto the CPU. It performs context switching. Context switching refers to saving the context of the process which was being executed by the CPU and loading the context of the new process that is being scheduled to be executed by the CPU.



3

CPU SCHEDULING

3.1 Scheduling Criteria

CPU scheduling will occur when:

- A new process arrives into the Ready state.
- A process undergoes Wait \rightarrow Ready state transition.
- A process undergoes Run \rightarrow Wait state transition for an I/O request.
- A process undergoes Run \rightarrow Ready state transition every q second where q is the time slice.
- The priority of a ready process is higher than the priority of a running process.

3.2 Goals of CPU Scheduling

- Maximize CPU utilization.
- Minimize the response time and waiting time of the processes.

3.3 Process times

3.3.1 Arrival Time (AT)

The time at which the process arrives into the Ready state is called arrival time of the process.

3.3.2 Burst Time (BT)

The time required by the process for its complete execution is called burst time/service time of the process.

3.3.3 Completion Time (CT)

The time by which a process completes its execution post arrival is called completion time of the process.

3.3.4 Turnaround Time (TAT)

The time difference between completion and arrival times of a process is called as the turnaround time of the process.

$$TAT = CT - AT$$

3.3.5 Waiting Time (WT)

The time spent waiting for the CPU to complete the execution is called as waiting time of the process.

$$WT = TAT - BT$$

3.3.6 Response Time (RT)

The time difference between the first response and arrival times of a process is called the response time of the process.

3.4 Gantt Chart

Gantt chart shows the execution time period of the processes. For example:



- Process P₁ started execution at time $t = 0$ and finished just before time $t = 10$.
- The CPU was idle from $t = 10$ to $t = 12$.
- Process P₂ started execution at time $t = 12$ and finished at time $t = 17$.
- Process P₃ started execution at time $t = 17$ and finished at time $t = 23$.

3.5 Scheduling Algorithms

3.5.1 FCFS

- It is a non pre-emptive algorithm.
- Processes are assigned to the CPU based on their arrival times. If two or more processes have the same arrival times, then the processes are assigned based on their process ids.
- It is free from starvation.
- It suffers from *Convoy effect*.

3.5.2 Shortest Job First (SJF)

- It is a non-pre-emptive algorithm.
- Processes are assigned to the CPU based on the shortest burst time. If two or more processes have the same burst times, then the processes are assigned to the CPU based on their arrival times.
- If all the processes have the same burst times, SJF behaves as FCFS.
- In SJF, the burst times of all the processes should be known prior execution.
- It minimizes the average response time of the processes.
- There is a chance of starvation.

3.5.3 Shortest Remaining Time First (SRTF)

- It is a pre-emptive algorithm.
- Processes are assigned to the CPU based on their shortest remaining burst times.
- If all the processes have the same arrival times, SRTF behaves as SJF.
- It minimizes the average turnaround time of the processes.
- If shorter processes keep on arriving, then the longer processes may starve.

3.5.4 Longest Remaining Time First (LRTF)

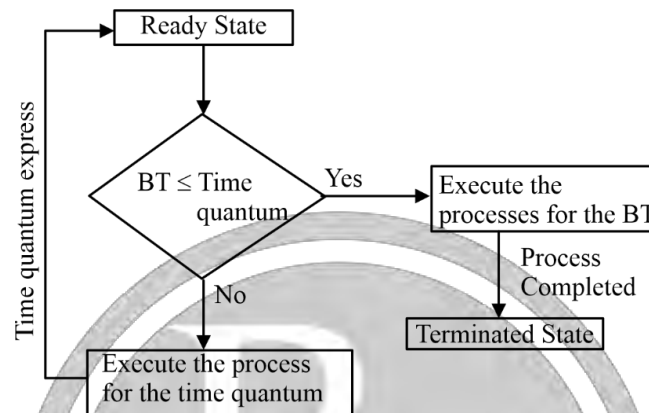
- It is a pre-emptive algorithm.
- Processes are assigned to the CPU based on their longest remaining burst times.
- It minimizes the average response time of the processes.
- It favours CPU bound processes.
- It is free from starvation.

3.5.5 Priority based Scheduling

- Priority scheduling can either be pre-emptive or non-pre-emptive.
- In pre-emptive priority scheduling, a process is voluntarily pre-empted whenever a higher priority process arrives.
- In non-pre-emptive priority scheduling, the scheduler picks up the highest priority process.
- If all the processes have equal priority, then priority scheduling behaves as FCFS.

3.5.6 Round Robin Scheduling

- RR scheduling is a pre-emptive FCFS based on the concept of time quantum or time slice.



- If the time quantum is too small, then the number of context switches (overhead) will increase and the average response time will decrease.
- If the time quantum is too large, then the number of context switches (overhead) will decrease and the average response time will increase.
- If the time quantum is greater than the burst times of all the processes, RR scheduling behaves as FCFS.

3.5.7 Highest Response Ratio Scheduling

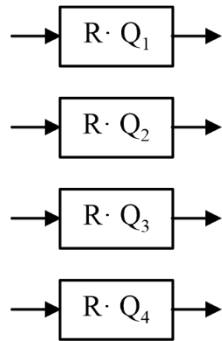
- It is a non pre-emptive algorithm.
- Processes are assigned to the CPU based on their highest response ratio.
- Response ratio is calculated as-

$$\text{Response Ratio} = \frac{WT + BT}{BT}$$

- It is free from starvation.
- It favours the shorter jobs and limits the waiting time of the longer jobs.

3.5.8 Multilevel Queue Scheduling

Multi-Level Queue Scheduling



- Depending on priority of the process, in which particular ready queue, the process has to be placed will be decided.
- High priority processes will be placed in top-level ready queue and low priority process will be placed in bottom level ready queue.
- Only after completion of all the processes, from top level ready queue the further level ready queue processes will be scheduled.
- If this is the strategy followed, then the processes which are placed in bottom level ready queue suffer from starvation. If higher priority processes keep on arriving, the lower priority processes may starve. This problem can be solved with the help of aging. Aging is a technique which automatically increases the priority of the processes that have been waiting in the system for a very long time.

□□□

4

MULTITHREADING

4.1 Threads

- A thread is a light-weight process.
- In multithreading, the threads of the same process share user address space, files, signal and signal handlers etc.
- In multithreading, each thread has its own stack, thread id, CPU state information (control and status registers, stack pointers) and scheduling information (thread state, priority etc.).

4.2 Benefits of Threads

- Improved response time.
- Faster context switches.
- Effective utilization of multiprocessor systems
- Enhanced throughput of the system.
- Economical.
- Effective resource sharing and utilization.

4.3 Types of Threads

4.3.1 Based on the number of threads

There are two types of threads:

- (i) Single thread process
- (ii) Multi thread process

4.3.2 Based on level

There are two types of threads:

- (i) User-level threads
- (ii) Kernel-level threads

4.3.3 User level threads versus kernel level threads

User level threads	Kernel level threads
These threads are managed at user level.	These threads are managed at kernel level.
These threads are not recognized by the kernel.	These threads are recognized by the kernel.
They are implemented as dependent threads.	They are implemented as independent threads.
All user-level threads of a process can run on one processor only and only one thread runs at a time.	The kernel-level threads of a process can run on different processors concurrently in a multi-processor environment.
Blocking one user-level thread of a process blocks the entire process.	Blocking one kernel-level thread of a process does not affect the other threads of the process.
These threads have less context.	These threads have more context.
Scheduling of user-level threads is done by the thread libraries.	Scheduling of kernel-level threads is done by the operating system.
No hardware support is required.	Hardware support is required.
Implementation is easy and simple.	Implementation is complicated and difficult.

4.4 Multithreading Models

4.4.1 Many-to-One

- Many user level threads are mapped to single kernel-level thread.

4.4.2 One-to-One

- Each user level thread is mapped to single kernel-level thread.

4.4.3 Many-to-Many

- Many user level threads are mapped to multiple kernel-level threads.
- It allows the OS to create a sufficient number of kernel threads

4.5 Thread Libraries

- Thread libraries provide programmer with API for creating and managing threads.
- There are two primary ways of implementing thread libraries:
 - Library entirely in user space
 - Kernel-level library supported by the OS
- Examples include pthread libraries, java threads, green threads etc.

4.6 Disadvantages of Multithreading

- Blocking:** Blocking one user-level thread of a process blocks the entire process. If the kernel thread is single threaded, then blocking the kernel thread will block the whole process. Consequently, CPU may remain idle during this period.
- Security:** Since, there is an extensive sharing among threads, there is a potential problem of security.
- Maintaining Thread Control Block (TCB) is considered as overhead for the system.



5

SYNCHRONIZATION

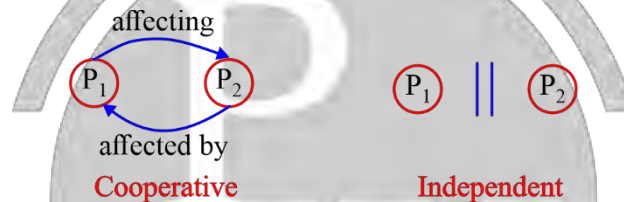
5.1 Synchronization

Processes are categorized into two types

Process



Execution of one process affects or affected by other process then those processes are said to be cooperative process. Otherwise, they are said to be Independent Process.



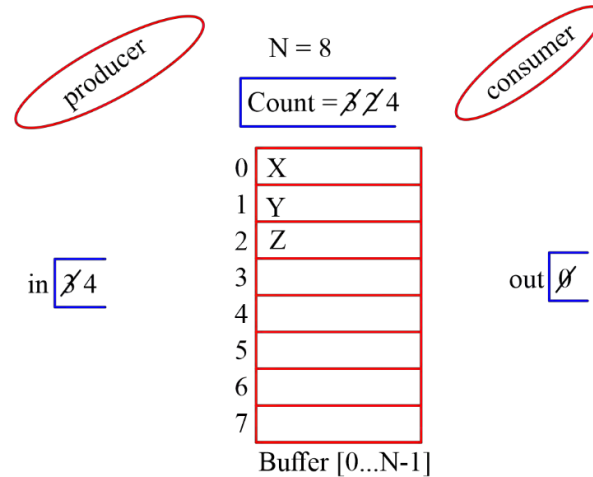
Processes must be cooperative to be synchronized.

5.2 Understanding Synchronization

- (1) The problems arise not having synchronization between the processes.
- (2) The conditions to be followed to achieve synchronization.
- (3) The solutions (wrong and right).

5.3 Problems arises for not having synchronization between the processes

5.3.1 Producer Consumer



int count = 0; // Initially buffer will be empty and only producer will be allowed to execute first.

void producer (void)

```
{
    int itemp;
    while(true)
    {
        produce_item(itemp);
        while (count == N); // Buffer Full
        Buffer [in] = itemp;
        in = (in + 1) mod N; // ← to repeat in value from 0 - 7
        count = count + 1;
    }
}
```

Register of producer memory

```
I. Load  $R_p.m[count]$ 
II. INCR  $R_p$ 
III. Store  $m[count], R_p$ 
```

void consumer(void)

```
{
    int itemc;
    while(true)
    {
        while (count == 0);
        itemc = Buffer [out];
        out = (out + 1) mod N;
        count = count - 1;
        process_item(itemc);
    }
}
```

```
I. Load  $R_c.m[count]$ 
II. DECR  $R_c$ 
III. Store  $m[count], R_c$ 
```

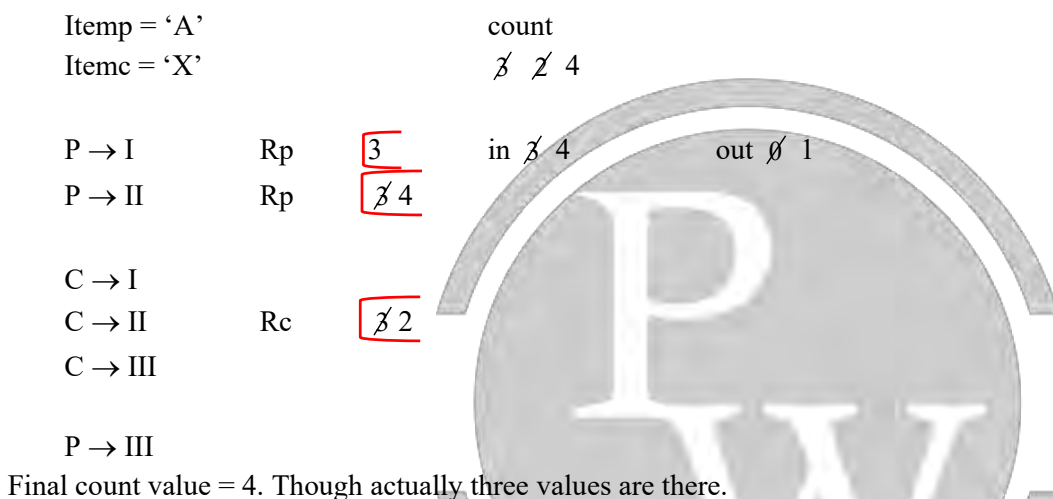
- IN is a variable used by the producer to identify the next empty slot in the buffer.
- OUT is a variable used by the consumer to identify from where it has to consume the item.

- Count is a variable used both producer and consumer to identify no of items present in the buffer at any point of time.
- Shared Resources:**
 - Count Variable
 - Buffer
- Two conditions to be followed:**
 - If the buffer is full, producer is not allowed to produce the item into the buffer.
 - If the buffer is empty, consumer is not allowed to consume the item from the buffer.

5.3.2 Universal Assumption

The running process can get pre-empted at any point of time after completion of current instruction.

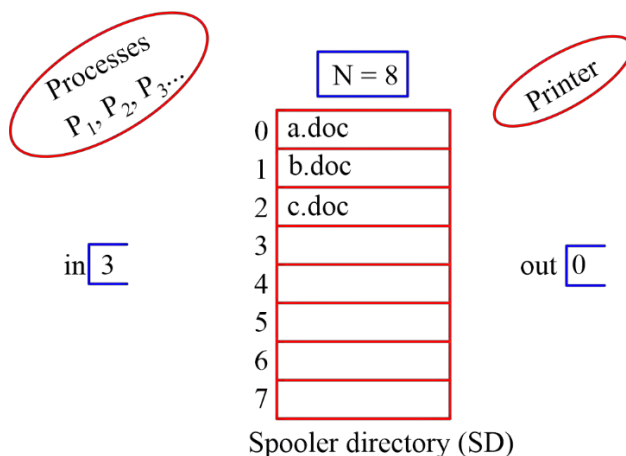
Analysis:



(1) Inconsistency:

The producer and consumer are not properly synchronized while sharing the common variable “count”. Hence it is leading to the problem of inconsistency.

5.3.4 Printer Spooler Daemon (Daemon → Background process)



5.3.5 Definitions



(1) Critical Section:

The portion of program text where shared variables or shared resources will be placed.

Example:

Count = Count + 1
OR
Count = Count – 1

(2) Non-Critical Section:

The portion of program text where the independent code of the processes will be placed.

Example:

$T_n = (in + 1) \bmod N$

(3) RACE condition:

The final value of any variable depends on execution sequence of the processes. This type of condition is called as RACE condition.

- To avoid the RACE condition, only one process is allowed to enter into critical section.

5.3.6 Conditions to be followed to achieve Synchronization

(1) Mutual Exclusion (M.E):

- No two processes may be simultaneously present inside the critical section at any point of time.
- Only one process is allowed to enter into critical section at any point of time.

(2) Progress:

- No process running outside the critical section should block the other intersected process from entering into critical section when critical section is free.
- If there is only one process trying to enter into critical section then it should be definitely allowed to enter into critical section.
- If two or more processes are trying to enter into critical section then one process should be definitely allowed to enter critical section.

(3) Bounded Waiting:

- No process should have to wait forever to enter into critical section.
- There should be a bound on getting chance to enter into critical section.
- Some process is indefinitely waiting to enter into critical section because critical section is always busy by some other processes. This situation should not arise.
- If the bounded waiting is not satisfied then it is possible for starvation.

(4) No assumptions related to hardware and the processor speed:

- Number of processes.

Solutions:

I. Software Type:

- (a) Lock Variables
- (b) Strict alteration or Decker's Algorithm
- (c) Petersons Algorithm

II. Hardware Type:

- (a) TSL Instruction Set
- (b) Test and Set Lock

III. OS Type:

- (a) Counting semaphore
- (b) Binary Semaphore

IV. Programming Language Compiler Support Type:

- (a) Monitors

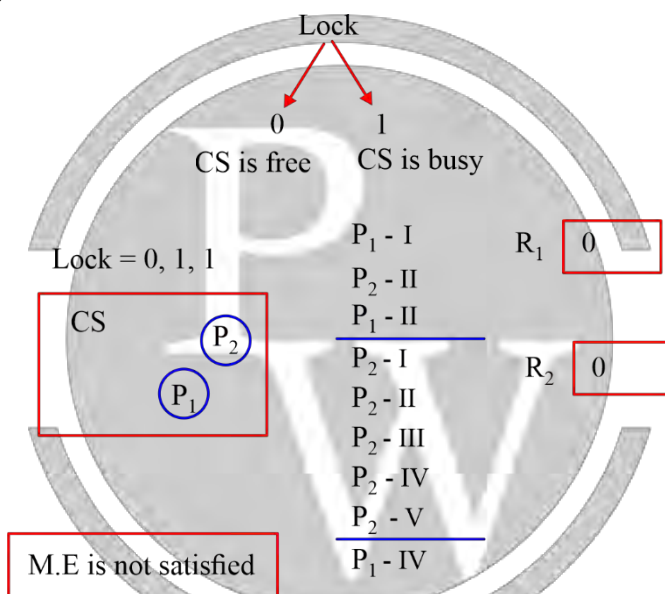
I. Software Types:

(a) Lock Variables:

Entry Section:

- Load $R_i, m[\text{Lock}]$ ($R_i \rightarrow$ Respective Process Register)
- Cmp $R_i, \#0$
- JNZ to Step (I)
- Store $m[\text{Lock}], \#1$
- C.S
- Store $m[\text{Lock}], \#0$

Analysis:



We have proved that both the processes P_1 and P_2 are entering into the critical section at the same time, Hence mutual exclusion is not satisfied and the solution is bound to be incorrect.

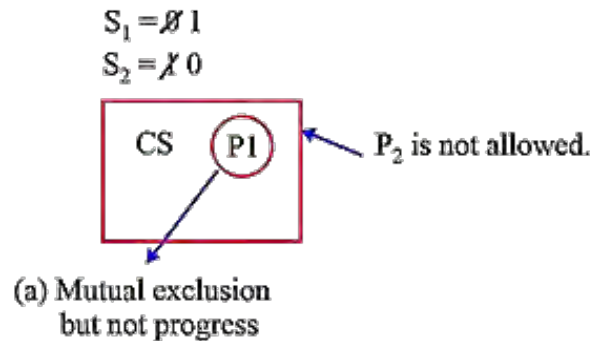
(b) Strict Alteration or Decker's Algorithm:

(Process takes 'Turn' to enter into C.S)

Process ' P_0 ' code	Process ' P_1 ' code
<pre>while (true) { non_cs (); while (turn! = 0); c.s turn = 1; }</pre>	<pre>while (true) { non_cs(); while(turn! = 1); c.s turn = 0; }</pre>

Important Points:

- The pre-emption is just a temporary stop and the process will come back and continue the remaining execution.
- If there is any possibility of solution becoming wrong by taking the pre-emption then consider the pre-emption.
- If any solution is having deadlock the progress is not satisfied.



(c) Peterson's Algorithm:

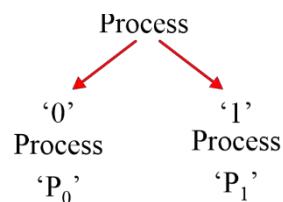
(Two Process Solution)

```

#define N 2
#define TRUE 1
#define FALSE 0
int turn;
int interested [N];
void enter_Region (int process)
{
    1. int other
    2. other = 1 - process;
    3. interested [process] = TRUE;
    4. turn = process;
    5. while (turn == process && interested [other] == TRUE);
}
C.S
void leave_Region(int process)
{
    interested [process] = FALSE;
}

```

initially
 interested [0] = FALSE;
 interested [1] = FALSE



TURN is a shared variable used by both the processes P_0 and P_1 , interested [N] is also shared by both the processes.

5.3.7 Hardware Type

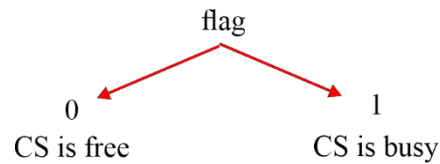
(a) TSL Instruction Set: (Test and Set Lock):

- **TSL Register Flag:**

Copies the current value of flag into register and stores the value of '1' into flag in a single atomic cycle without any pre-emption.

- **Entry Selection:**

1. TSL Ri, m[flag]
2. Cmp Ri, ≠ 0
3. JMP to step (1)
4. c.s
5. Store m[flag], ≠ 0



Algorithm	M.E.	Progress	Bounded Waiting
1. Lock Variable	×	✓	×
2. Strict alteration or Decker's Algorithm	✓	×	✓
3. Peterson's Algorithm	✓	✓	✓
4. TSL Instruction	✓	✓	×

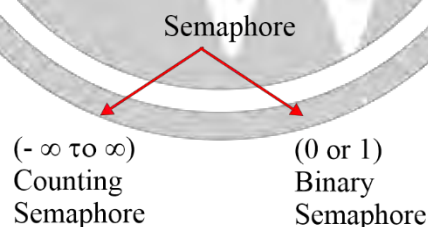
5.3.8 OS Type

Semaphore:

Semaphore is an integer variable which is used by the various processes in a mutual exclusive manner to achieve synchronization.

Improper usage of semaphore will also give the wrong results.

Semaphore is categorised into 2 types:



The two different operations will be performed on the semaphore variable.

- (1) Down, or wait (); or p ()
- (2) up (); or signal (); or v (); or release ();

(a) Counting Semaphore:

```

Down (Semaphore s)
{
    s.value = s.value - 1;
    if (s.value < 0)
    {
        Block the process and
        place its PCB in the
  
```



```

        suspended list ();
    }
}
Up (Semaphore s)
{
    s.value = s.value + 1;
    if (s.value ≤ 0)
    {
        Select a process from
        suspended list and
        wakeup ();
    }
}

```

(b) Binary Semaphore:

```

Down (Semaphore S)
{
    if (S.value == 1)
        S.value = 0;
    else
    {
        block the process
        and place its PCB
        in the suspended list ()
    }
}

Up (Semaphore S)
{
    if (Suspended list is empty)
        S.value = 1;
    else
    {
        select a process
        from suspended list
        and wakeup ();
    }
}

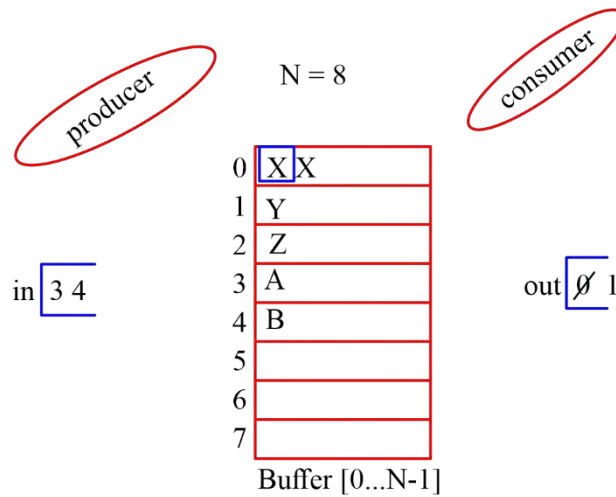
```

Notes: (Applicable for both counting and binary semaphore).

- ⇒ Every semaphore variable will have its own suspended list.
- ⇒ The down and up operations are atomic [OS will prevent the interrupts]
- ⇒ If more than one process is in the suspended list then every time when we perform one up operation one process will wakeup from the suspended list and this will be based on (FIFO → to ensure bounded waiting)
- ⇒ If two or more processes are in the suspended list and there is no other process to wakeup these processes then those processes are said to be involved in the deadlock.

5.4 Classical problems of IPC (Inter Process Communication)

- Producer consumer with semaphore:



```
semaphore mutex = 1;
```

```
semaphore empty = N;
```

```
semaphore full = 0;
```

```
void producer(void)
```

```
{
```

```
    int item p;
```

```
    while(true)
```

```
    {
```

```
        produce_item (item p);
```

```
        down(empty);
```

```
        down(mutex);
```

```
        buffer [in] = item p;
```

```
        in = (in + 1) mod N
```

```
        up(mutex);
```

```
        up(full);
```

```
    }
```

```
}
```

```
void consumer (void)
```

```
{
```

```
    int item c;
```

```
    while (true)
```

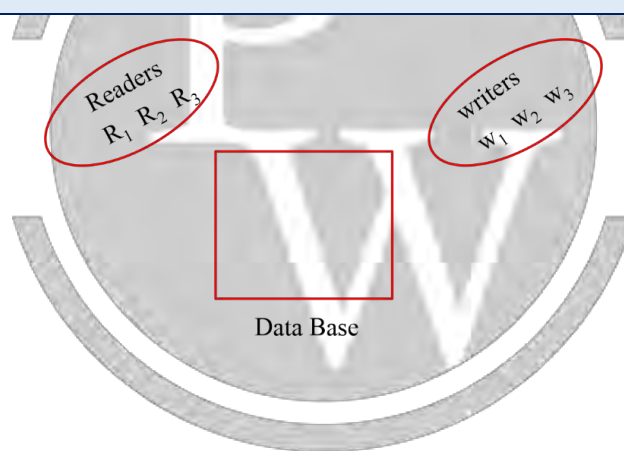
```

{
    down (full);
    down (mutex);
    item c = buffer [out];
    out = (out + 1)mod N;
    up (mutex);
    up (empty);
    process_item (item c);
}
}

```

- mutex is a binary semaphore used by the producer and consumer to access the buffer in a mutual exclusive manner.
- empty is counting semaphore variable represents number of empty slots in the buffer at any point of time.
- full is counting semaphore variable represents number of slots full in the buffer at any point of time.

5.5 READERS WRITERS PROBLEM



```

int rc = 0;
semaphore mutex = 1;
semaphore db = 1;
void reader (void)
{
    while (true)
    {
        down (mutex);
        rc = rc + 1;
        if (rc == 1) down (db);
        up (mutex);
    }
}

```

```

}

void writer (void)
{
    while (true){
        down (db);

        D.B

        up (db);
    }
}


```

5.5.1 conditions to be followed

- | | |
|------------|------------|
| 1) R - W ✗ | 2) R - R ✓ |
| 3) W - R ✗ | 4) W - W ✗ |

- rc is a integer variable represents readers count i.e number of readers present in the data base at any point of time.
- mutex is a binary semaphore used by the readers in mutual exclusive manner.
- db is a binary semaphore variable used by readers and writers in a mutual exclusive manner.

5.5.2 Dining Philosophers' problem

	<p><u>Dining Philosophers problem</u></p> <ul style="list-style-type: none"> • N philosophers and N forks • Philosophers eat/think <ul style="list-style-type: none"> ▪ Eating needs 2 forks ▪ Pick one fork at a time
---	---

5.5.3 Solution 1 for Dining Philosophers problem

<pre> #define N5 /* number of philosophers */ Void philosophers (int i) /* i: philosophers' number from 0 to 4 */ { while (TRUE) { think (); /* philosopher is thinking */ take_fork (i); /* take left fork */ take_fork ((i+1)%N) /* take right fork; % is modulo operator */ eat (); /* yum-yum spaghetti */ put_fork (i); /* put left fork back on the table */ put_fork ((i+1)%N) /* put right fork back on the table */ } } </pre>	<ul style="list-style-type: none"> • This solution to dining philosopher problem suffers from Deadlock. • Everyone picks up their left fork first, then waits for right fork... \Rightarrow this leads to Starvation!
--	--

- This solution to dining philosophers' problem suffers from Deadlock everyone picks up their left fork first, then waits for right fork \Rightarrow This leads to Starvation!

5.5.4 Solution 2 – state based

```
# define N                5                /* number of philosophers */
# define LEFT            (i+N-1)%N        /* number of I's left neighbours */
# define RIGHT           (i+1)%N         /* number of I's right neighbours */
# define THINKING        0                /* philosopher is thinking */
# define HUNGRY           1                /* philosopher is trying to get forks */
# define EATING           2                /* philosopher is eating */
typedef int semaphore     /*semaphores are a special kind of int */
int state [N];            /* array to keep track of everyone's state */
semaphore mutex = 1;      /* mutual exclusion for critical regions */
semaphore s[N];           /* one semaphore per philosopher */

void philosopher (int i)    /* i: philosopher number, from 0 to N -1 */
{
    while (TRUE) {          /* repeat forever */
        Think();           /* philosopher is thinking */
        Take_forks(i);     /* acquire two forks or block */
        Eat();             /* yum-yum spaghetti */
        Put_forks(i);      /* put both forks back on table */
    }
}

void take_forks(int i)      /* i: philosopher number, from 0 to N -1 */
{
    down(&mutex);           /* enter critical region */
    state [i] = HUNGRY;    /* record fact that philosopher is hungry */
    test(i);               /* try to acquire two forks */
    up(&mutex);             /* exit critical region */
    down(&s[i]);            /* block if forks were not acquired */
}

void put_forks(i)          /* i: philosopher number, from 0 to N -1 */
{
    down(&mutex);           /* enter critical region */
    state [i] = THINKING; /* philosopher has finished eating */
    test(LEFT);            /* see if left neighbour can now eat */
    test(RIGHT);           /* see if right neighbour can now eat */
    up(&mutex);            /* exit critical region */
}

void test(i)               /* i: philosopher number, from 0 to N -1 */
{
    if (state[i] == HUNGRY && state [LEFT] == EATING && state [RIGHT] == EATING) {
        state [i] = EATING;
        up(&s[i]);
    }
}
```

Made picking up left and right chopsticks an atomic operation ξ or, $N - 1$ philosophers but N chopsticks ξ ...both changes prevent deadlock.

5.6 Monitors:

- Monitors is a programming language compiler support type of solution to achieve synchronization.
- Monitors is collection of variables and procedures combined together in a special kind of module or package.
- The process running outside the monitor cannot directly access the internal variables of the monitor but however they can call the procedures of the monitor.
- Monitors has an important property that only one process can be active inside the monitor at any point of time.

5.6.1 Syntax

Monitor example

```
{
    Variables;
    Condition variables;
    Procedure P1
    {
    }
    Procedure P2
    {
    }
}
```

5.6.2 Condition variables

Condition x,y;

The two different operations performed on the condition variables of the monitor.

1. Wait operation: wait ()
2. Signal operation: signal ()

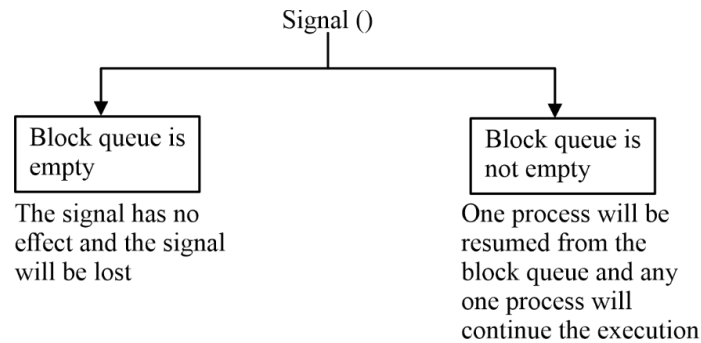
- **Wait ()**

Ex – x.wait (); or wait (x)

The process performing wait operation on any condition variable will be suspended and suspended process will be placed in the “block queue” of respective condition variable.

- **Signal ()**

Ex – x.signal (); or signal(x)



5.6.3 Concurrent Programming

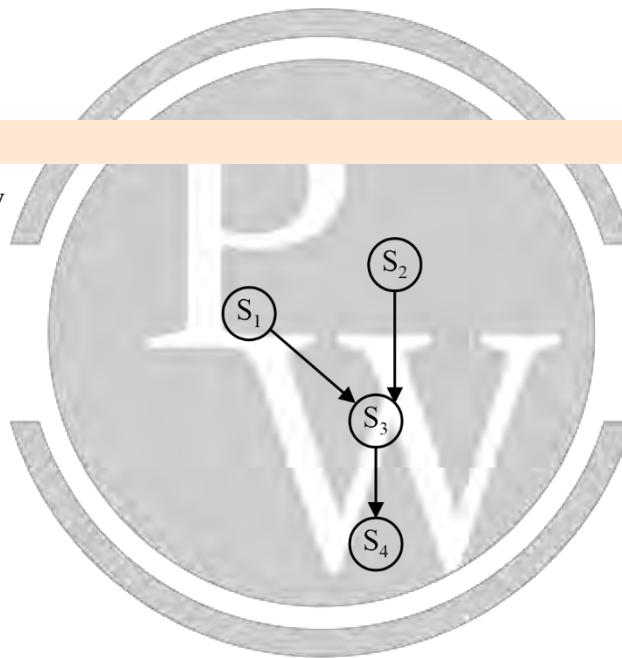
$S_1: a = b + c ;$
 $S_2: d = e * f ;$
 $S_3: g = a / d ;$
 $S_4: h = g * i ;$

Read set = {b, c, e, f, a, d, g, i}

Write set = {a, d, g, h}

5.6.4 Precedence graph

S_1, S_2 can execute concurrently



- Any two statements S_i and S_j can be executed concurrently or parallel if they are following the conditions.

- (1) $R(S_i) \cap W(S_j) = \varnothing$
 - (2) $W(S_i) \cap R(S_j) = \varnothing$
 - (3) $W(S_i) \cap W(S_j) = \varnothing$

- The real concurrent programming is possible only on the multiprocessor system.
 - Concurrent has 3 different meanings
 - They can execute concurrently or parallel
 - They don't have any dependency
- Anyone can start first [for single processor this will be applicable]



6

DEADLOCK

6.1 Concept of Deadlock

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example:
 - System has 2 disk drives.
 - P_1 and P_2 each hold one disk drive and each needs another one.

6.2 System Model

- Resource types R_1, R_2, \dots, R_m
- Resources include CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
 - request
 - use
 - release

6.3 Deadlock Characteristics

6.3.1 Mutual Exclusion

There should be a one-to-one relationship between a resource and a process.

6.3.2 Hold and Wait

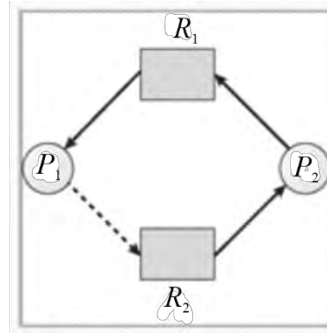
This condition arises when a process is requesting another resource while holding on to a resource.

6.3.3 No Pre-emption

The resource has to be voluntarily released by the process after its execution. It is not allowed to forcibly pre-empt a process to release its held resources.

6.3.4 Circular Wait

The processes are circularly waiting on each other for the resources.



Note:

If all the deadlock characteristics simultaneously exist in the system, then the system is in deadlock.

6.4 Deadlock Prevention

It ensures that the system will *never* enter into a deadlock state. Deadlock can be prevented by unsatisfying one of the mentioned deadlock characteristics.

- Mutual exclusion cannot be unsatisfied because of the concept of sharable and non-sharable resources.
- Hold and Wait characteristic can be unsatisfied through any of the following strategies:
 - (i) A process should be assigned all the required resources before the start of its execution. This may lead to low device utilization.
 - (ii) The process should release all the existing resources before making a new request. This may eventually lead to starvation.
- Pre-emption can be achieved as:
- Suppose a process P_1 is requesting for a resource R which is held by another process P_2 . If P_2 is already in execution then P_1 has to wait; else, the resource R will be pre-empted from P_2 and allocated to P_1 .
- Circular wait can be avoided by the following algorithm:
 - (i) Assign unique numbers to resources. (assume there exists single instance of a resource).
 - (ii) A process can request for a resource in either increasing or decreasing order of enumeration.

6.5 Deadlock Avoidance

Deadlock avoidance is achieved by implementing Banker's algorithm. Banker's algorithm ensures that the system never enters in unsafe state.

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- System is in safe state if there exists a sequence $\langle P_1, P_2, \dots, P_n \rangle$ of ALL the processes in the systems such that for each P_i , the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j .
- If a system is in safe state \Rightarrow no deadlocks.
- If a system is in unsafe state \Rightarrow possibility of deadlock.

6.5.1 Data Structures for Banker's algorithm

Let n = number of processes and m = number of resources, then-

- **Available:** Vector of length m . If available $[j] = k$, there are k instances of resource type R_j available.
- **Max:** $n \times m$ matrix. If $Max[i, j] = k$, then process P_i may request at most k instances of resource type R_j .
- **Allocation:** $n \times m$ matrix. If $Allocation[i, j] = k$ then P_i is currently allocated k instances of R_j .
- **Need:** $n \times m$ matrix. If $Need[i, j] = k$, then P_i may need k more instances of R_j to complete its task.
- $Need[i, j] = Max[i, j] - Allocation[i, j]$.

6.5.2 Banker's Algorithm

```

Algo Bankers(i, Request, Available, Allocation, Need){
    If( $Need_i \leq Max_i$ )
    {
        If( $Need_i \leq Available$ )
        {
            Available = Available -  $Need_i$  ;
            Allocation = Allocation +  $Need_i$  ;
             $Max_i = Max_i - Need_i$  ;
            Run safety Algorithm;
            If the system is in safe state, then grant the  $Need_i$ ;
            Else block the process;
        }
    }
}

```

6.6 Deadlock Detection

- A cycle in the resource allocation graph represents deadlock only when resources are of single-instance type.
- If the resources are of multiple instance type, then the safety algorithm is used to detect deadlock.

6.7 Deadlock Recovery

A system can recover from deadlock through adoption of the following mechanisms:

- Process termination
- Resource Pre-emption
- Ostrich Algorithm:
- Ignore the problem and pretend that deadlocks never occurred in the system; used by most operating systems, including UNIX.



7

MEMORY MANAGEMENT

7.1 Introduction

- In multiprogramming system, the task of subdividing the memory among the various processes is called memory management.
- The task of the memory management unit is the efficient utilization of memory and minimize the internal and external fragmentation.

7.2 Logical versus Physical Address

- The address generated by CPU is called the logical address.
- The address perceived by the memory unit is called physical address.

7.3 Memory Management Unit

- Hardware device that maps virtual to physical address.
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses.

7.3.1 Loading

It is defined as bringing the program from the secondary to the main memory.

It is classified into three types:

(i) Absolute Loading

- A given program is always loaded into the same memory location whenever loaded for execution.

(ii) Relocatable Loading

- A given program is loaded into any desired memory location whenever loaded for execution.
- The compiler must generate relative address for the program.

(iii) Dynamic Loading

- Routine is not loaded until it is called better memory-space utilization (unused routine is never loaded, postponed until execution time).
- Useful when large amounts of code are needed to handle in frequently occurring cases.
- No special support from the operating system is required. It is implemented through program design
- Address translation is performed through the hardware.

7.3.2 Linking

Linking is the process of collecting and combining various pieces of code and data into a single file that can be loaded (copied) into memory and executed. It can be performed at compile time, load time or run time.

It is classified into two types:

- **Static Linking**
Static linkers take as input a collection of relocatable object files and command-line arguments and generate as output a fully linked executable object file that can be loaded and run.
- **Dynamic Linking**
A shared library is an object module that, at run time, can be loaded at an arbitrary memory address and linked with a program in memory. This process is dynamic linking.

7.3.3 Address Binding

Association of the program instructions and data into the actual physical memory locations is called as address binding. Address binding of instructions and data to memory addresses can happen at three different stages

- **Compile time:** If memory location known a priori, **absolute code** can be generated. It must recompile code if starting location changes.
- **Load time:** It must generate **relocatable code** if memory location is not known at compile time.
- **Execution time:** Address binding is delayed until run time if the process can be moved during its execution from one memory segment to another. It needs hardware support for address mapping (e.g., base and limit registers).

7.4 Memory Management Techniques

7.4.1 Contiguous Memory Allocation

It comprises of – Fixed Partition Scheme and Variable Partition Scheme

Fixed Partition Scheme / Static Partitioning

- The memory is divided into a fixed number of partitions of equal/unequal sizes.
- Every partition is associated with limit registers.
- The degree of multiprogramming is restricted by the number of partitions.
- Partition size may not be large enough for any waiting process.
- Internal fragmentation may be present.

Variable Partition Scheme / Dynamic Partitioning

- Initially, memory is available as a single continuous free block. When a process arrives, a hole large enough to accommodate it is created in the memory.
- There is no internal fragmentation.
- It suffers from external fragmentation.
- It is associated with the overhead of compaction.

Note

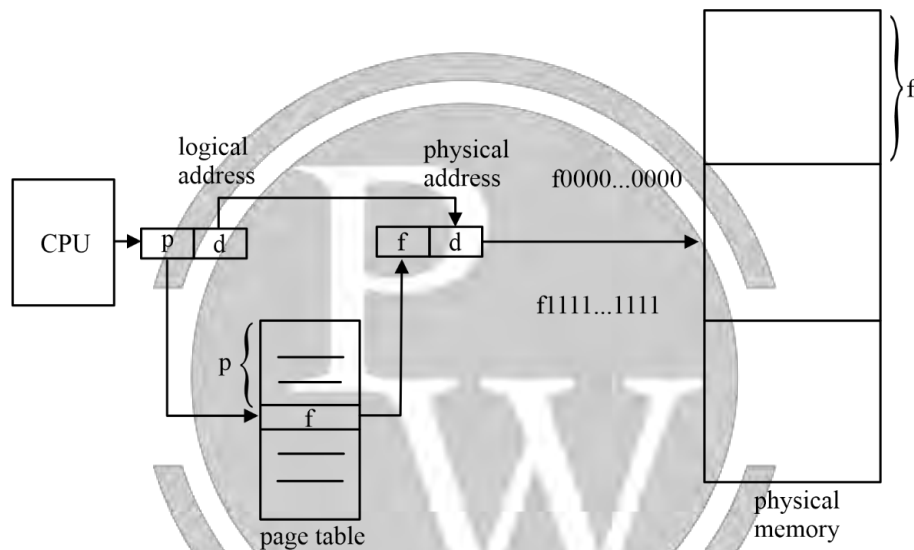
If more than one partition is sufficient to accommodate a process, then any of the following dynamic allocation methods can be adopted:

- **First Fit:** Allocate the first partition that is big enough starting from the beginning of the memory.
- **Next Fit:** It behaves similar to first fit except that it scans the memory after the 'last allocation point'.
- **Best Fit:** It scans the entire memory to find the smallest sufficient partition to accommodate the process.
- **Worst Fit:** It scans the entire memory to find the largest sufficient partition to accommodate the process.

7.4.2 Non-Contiguous Memory Allocation

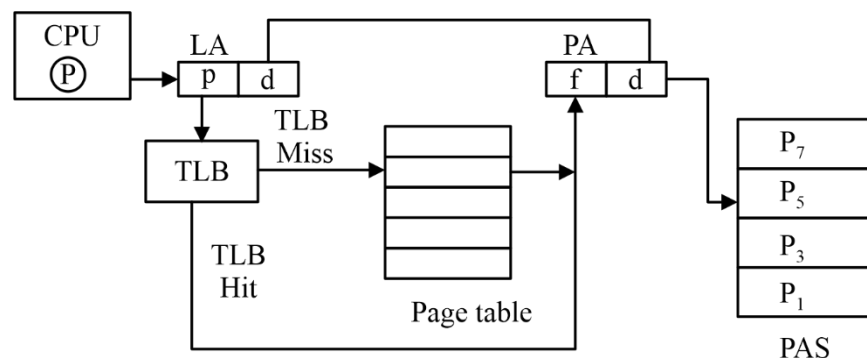
Paging

- The technique of mapping CPU generated logical address to physical address is called paging.
- Logical address space is divided into equal size pages. Physical address space is divided into equal size frames. In paging, the frame size is equal to the page size.
- When a process is created, paging will be applied on the process. A page table will be maintained in the main memory for a process. The base address of the page table will be stored in the process control block.
- The number of entries in the page table is equal to the number of the pages in the logical address space. Each page table entry contains the frame number. Hence, the page table is also known as the **address translation table**.
- There is no external fragmentation in paging. Internal fragmentation may exist in the last page and is formulated as $\left\lfloor \frac{p}{2} \right\rfloor$ where p is the page size.



Paging with TLB

- Translation Look Aside Buffer** is a hardware device implemented using associative registers.
- TLB** is added to improve the performance of paging. The TLB access time will be very less compared to the main memory access time.
- TLB** contains frequently referred page numbers and their corresponding frame numbers.



- TLB access time = c
- TLB hit ratio = x

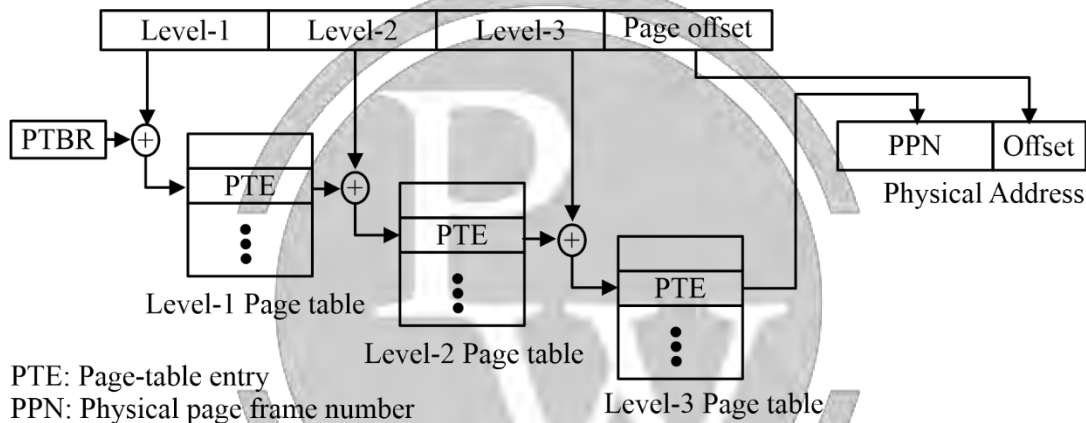
Then the formula for effective memory access time:

$$E.M.A.T = x(c + m) + (1 - x) \cdot 1 \cdot (c + 2m)$$

1 TLB access
1 MR for actual page
1 MR for PT
1 MR for actual page

Multilevel Paging

- To avoid the overhead of maintaining large size page tables, multilevel paging will be applied.
- In multilevel paging, pages of the page table are brought into the main memory.
- The page tables of all the levels of multi-level paging are kept in the memory.
- The entries of the level-1 page table are pointers to the level-2 page table.
- The entries of the level-2 page table are pointers to the level-3 page table.
- The entries of the last level page table will have frame numbers of the actual page.
- All levels page table entry must contain frame number.
- Address translation is done as:

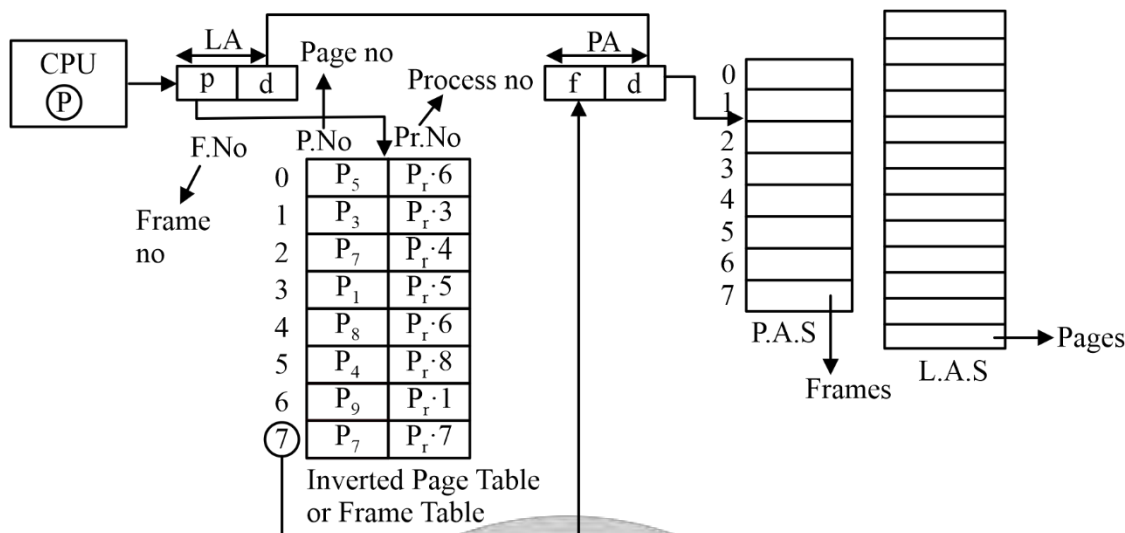


- Performance of multi-level paging with TLB:
Assume, TLB hit ratio = p , TLB access time = c , Main Memory Access time = m
If the system uses n -level paging, then effective memory access time is given as-
$$EMAT = p(c + m) + (1 - p)(c + (n + 1)m)$$

Inverted Paging

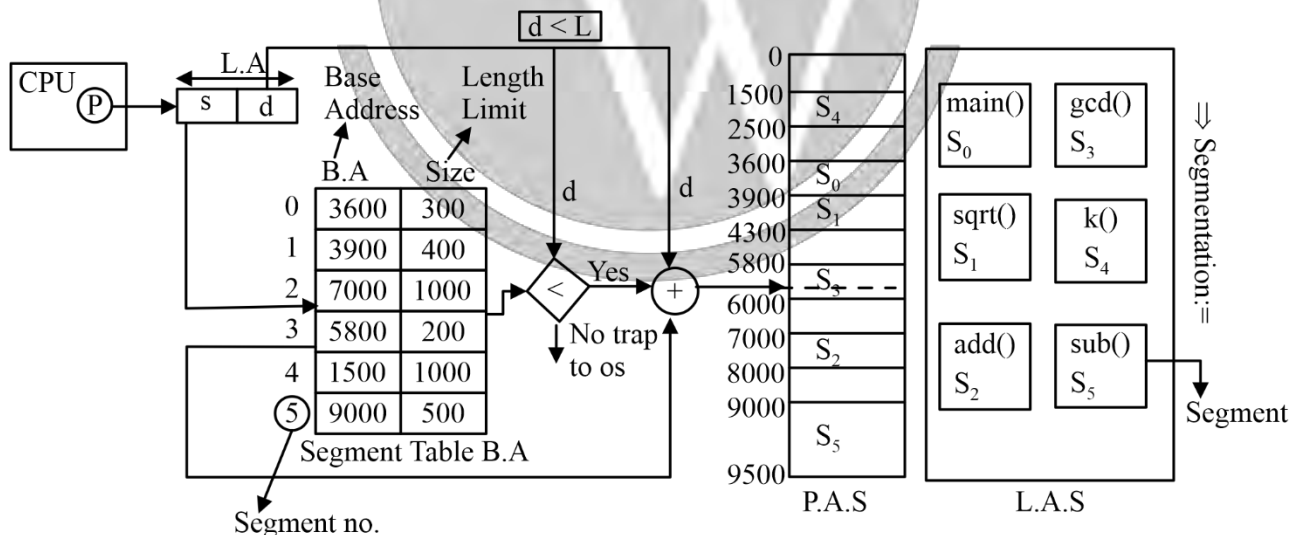
- Inverted paging maps physical frames to virtual pages.
- Instead of maintaining multiple page tables for different processes, an inverted page table can be maintained.
- The number of entries in the inverted page table is equal to the number of the frames in the physical address space.
- Each inverted page table entry contains a page number and a process identifier. It gives an idea about 'which page of which process is allocated in which frame'.
- The disadvantage of inverted paging is the increased lookup time and hard to implement.

- Address translation is done as:



Segmentation

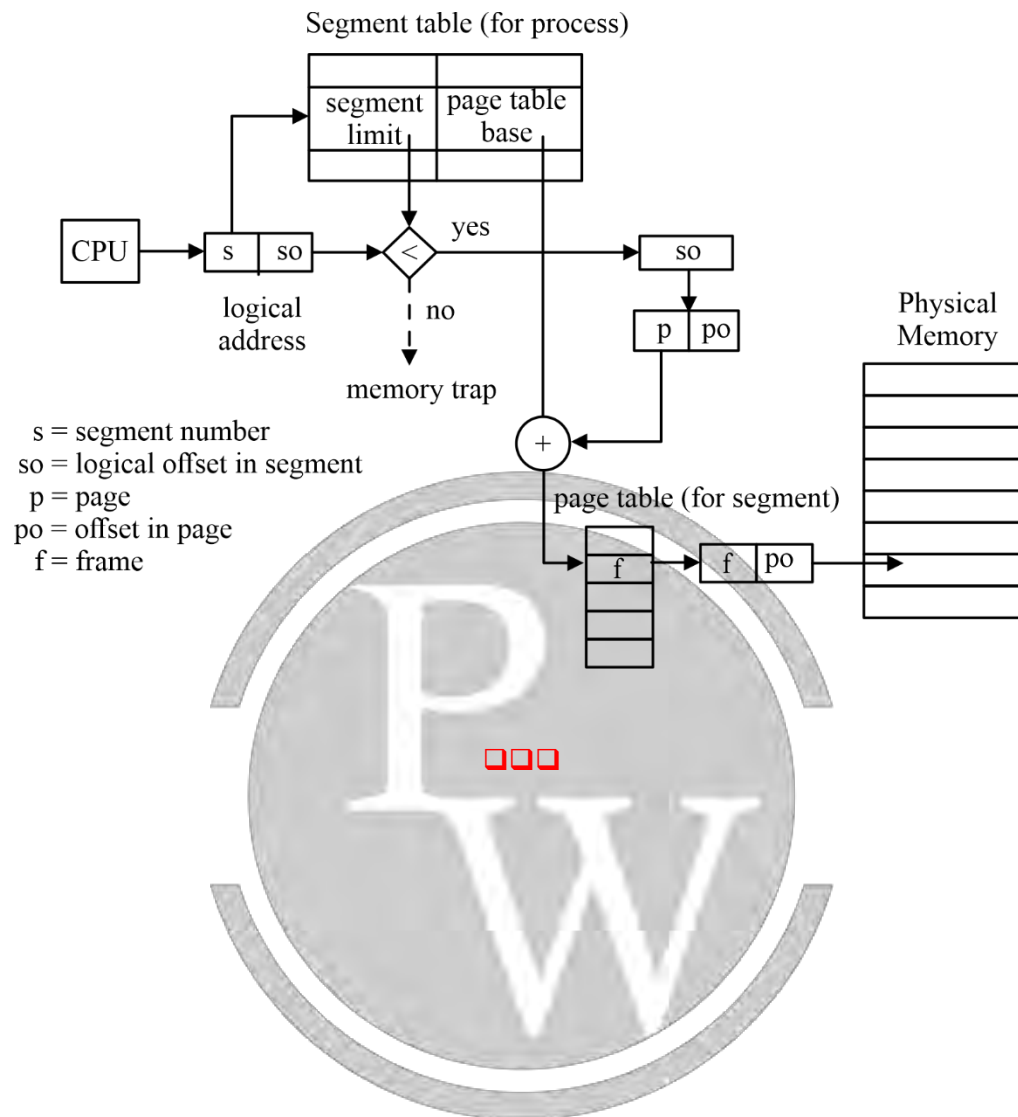
- Paging does not follow user's view of memory allocation. Instead of dividing the process into equal sized pages, it can be divided into variable length segments.
- A segment table is maintained for each process. The number of entries in the segment table is equal to the number of segments of a process.
- Each segment table entry contains the base (starting address) and limit(length) of the segment.
- Segmentation does not suffer from internal or external fragmentation.
- Address translation is done as:



Segmented Paging:

- To avoid the overhead of bringing large sized segments of a process into the main memory, paging will be applied on the segments.
- Pages of the segment will be loaded into the main memory.
- A page table will be maintained for each segment of the process.
- The number of entries in the page table is equal to the number of the pages of the segment in the logical address space.
- Lookup time increases.

- It suffers from internal fragmentation only.
- Address translation is done as:



08

VIRTUAL MEMORY

8.1 VM Concept

- It gives an illusion to the programmer that programs having larger size than the physical memory can be executed.
- It allows address spaces to be shared by several processes.

8.2 VM Implementation

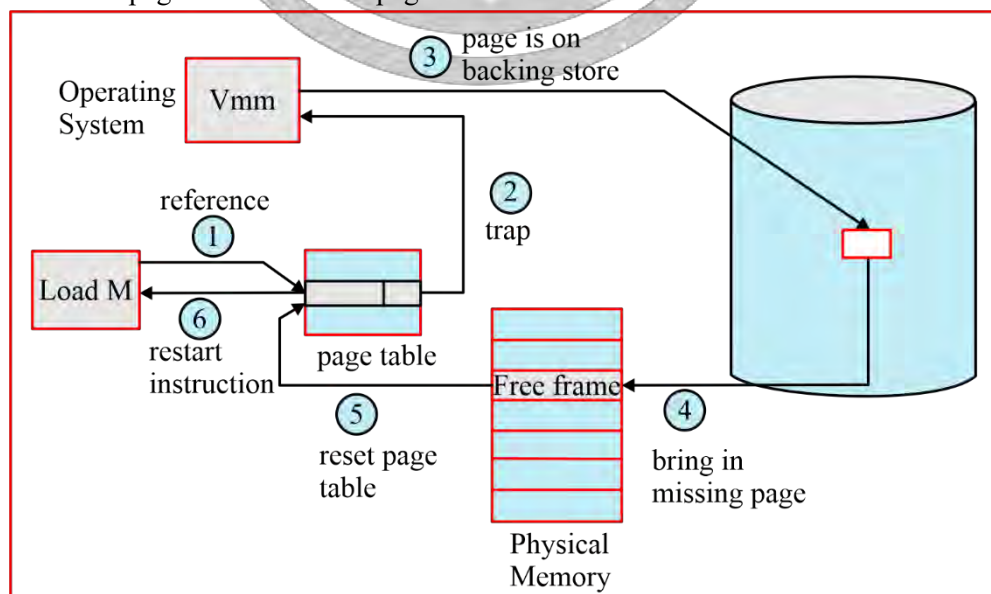
Virtual memory can be implemented through:

- Demand Paging
- Demand Segmentation

8.2.1 Demand Paging

Loading the pages from the secondary memory to the main memory on demand is called 'demand paging'.

- When the CPU tries to refer a page which is not available in the main memory, 'page fault' occurs. A signal is sent to the OS regarding the page fault. The OS locates the page into the Logical address space and loads it into the main memory frame. If the memory frames are full, appropriate page replacement algorithm is used. The respective page table entries are also updated accordingly.
- The time taken to service a 'page fault' is called 'page fault service time'.



8.3 Performance of virtual Memory

8.3.1 Temporal Issues

Let,

Page fault service time = 'S'

Main memory access time = 'M'

Page fault rate = 'P' where $0 \leq P \leq 1$

The effective memory access time is formulated as-

$$EMAT = P \times S + (1 - P) \times M$$

(Assume, page table access time is neglected)

8.3.2 Demand Paging with TLB

Assume, TLB hit ratio = 'h'

TLB access time = 'c'

Page fault service time = 'S'

Main memory access time = 'M'

Page fault rate = 'P' where $0 \leq P \leq 1$

The effective memory access time is formulated as-

$$EMAT = h(c + m) + (1 - h)(c + (P \times S + (1 - P) \times M))$$

8.3.3 Page Replacement Algorithms

FIFO

- When a page fault occurs and all the memory frames are full, FIFO algorithm replaces the oldest page to allocate the page referred by the CPU.
- It is implemented using a queue or time-stamp on pages.
- Sometimes, even on increasing the number of frames, the page fault rate increases. This situation is called Belady's Anomaly.

LRU

- LRU algorithm replaces the page that has not been used for the longest period (least recently used page).
- It is implemented using a stack or counter.

Optimal

- Optimal algorithm replaces the page that will not be used for the longest period in future.
- For a fixed number of frames, optimal algorithm gives the least page fault rate.
- It cannot be implemented in real time as it requires future references.

Note

Reference String is a set of successively unique pages referred in a given list of virtual addresses.

8.3.4 Thrashing

When CPU utilization is low, the OS may increase the degree of multiprogramming. After a certain point, the throughput of the system will gradually decrease as the system spends more time in page replacements owing to the lack of frames. This phenomenon is called 'thrashing'.

8.3.5 Working Set Model

It is defined as the set of the unique pages referred during the past ' Δ ' references.

- If $\Delta < (\text{total number of frames})$, the OS can bring in more processes.
- If $\Delta > (\text{total number of frames})$, the OS should instruct the mid-term scheduler to suspend some of the processes to avoid thrashing.



9

FILE SYSTEMS

9.1 File

- Collection of logically related entities/records

9.1.1 Attributes

- File Name
- File Type
- File Size
- Location of file
- Generation date
- Last modified date
- Permission
- Owner/Author
- Password

9.2 File Context

File context is stored in file control block (FCB)

File will have various types–

- .doc
- .txt
- .pdf
- .exe
- .obj
- .dll
- .png
- .apk
- .xls/.xlsx
- .jpg
- .mp3
- .mp4
- .avi/.flv/.mkv/.3gp
- .c/.cpp/.java/.xml/.html

9.3 Operations performed on file

- create
- open
- write
- read
- hide
- save
- save as
- close
- copy
- paste
- cut
- delete
- rename
- send/share
- print

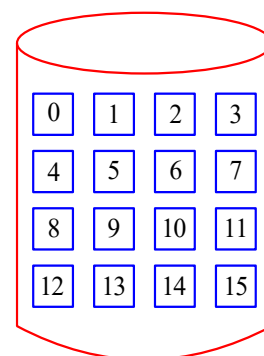
9.4 Access methods

- Sequential Access
- Random Access
- For better classification of files, files will be stored in directory.

9.5 Disk Space Allocation Methods

9.5.1 Contiguous Allocation

File	Starting Disk Block Address	Size w.r.t no. of DB
Abc.docx	2	4
Xyz.docx	9	5



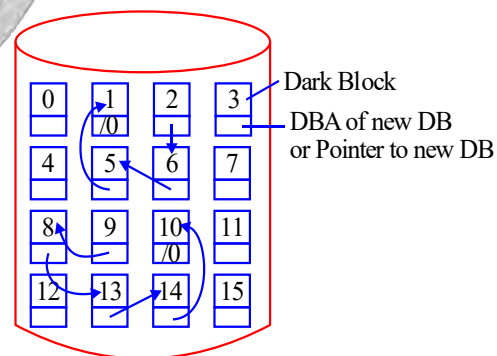
- In contiguous allocation, whenever file is created, disk blocks are allocated in a continuous manner.
- Every file is associated with two parameters
 - (1) Starting DBA
 - (2) Size
- Increasing the file size may not be possible always.
- It suffers from external fragmentation.
- Internal fragmentation may exist in the last disk block of the file.
- It supports both sequential and random access of file.

Starting disk block Address + Offset Value \Rightarrow Random location

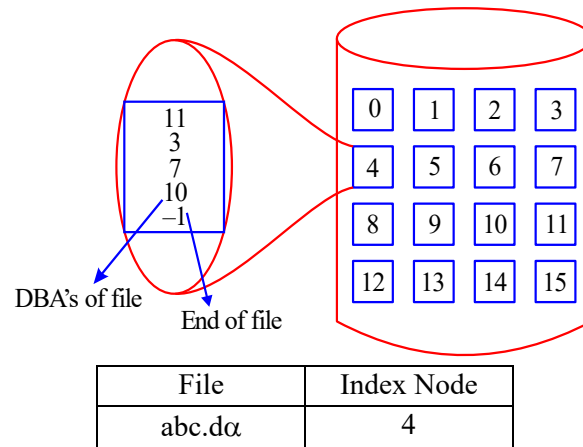
9.5.2 Linked Allocation/Non-contiguous allocation

File	Starting DBA	Ending DBA
abc.docx	2	1
xyz.docx	9	10

- In the linked allocation disk blocks are allocated in a non-continuous manner.
- Every file is associated with 2 parameters–
 - (1) Starting DBA
 - (2) Ending DBA
- Increasing file size is always possible if free disk block is available.
- There is no external fragmentation.
- Internal fragmentation may exist in last disk block of the file.
- There is an overhead of maintaining pointer in every disk blocks.
- If the pointer of any disk block is lost, the file will be truncated.
- It supports only sequential access of the file.



9.5.3 Indexed Allocation



- In the indexed allocation, every file is associated with its own index node.
- Index node contains all the disk block address of the file.
- There is no external fragmentation but internal fragmentation may exist in the last disk block of file.
- If the file is very large, then one disk block may not be sufficient to store all the disk block addresses of the file.
- If the file is very small, then it is waste of using one entire disk blocks. Just to store, the addresses.

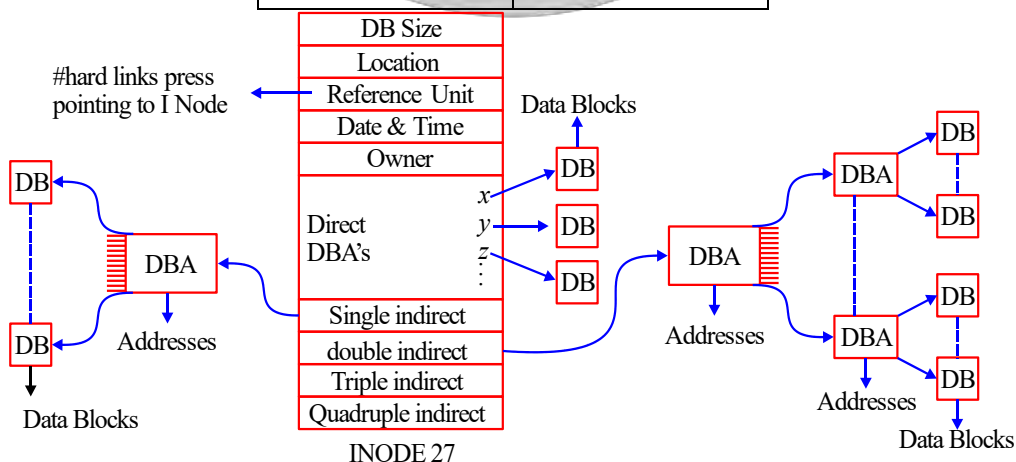
9.7 UNIX I-NODE IMPLEMENTATION

9.7.1 An extension of indexed allocation

$$\text{Total size of file system} = \left[\# \text{Direct DBA's} + \left(\frac{\text{DB Size}}{\text{DBA}} \right) + \left(\frac{\text{DB Size}}{\text{DBA}} \right)^2 + \left(\frac{\text{DB Size}}{\text{DBA}} \right)^3 + \dots \right] \times \text{DB Size}$$

\Downarrow \Downarrow \Downarrow
 #Single indrect. #Double Index #Triple Index
 DBA DBA DBA

File	I-NODE
abc.docx	27



9.8 DISK FREE SPACE MANAGEMENT

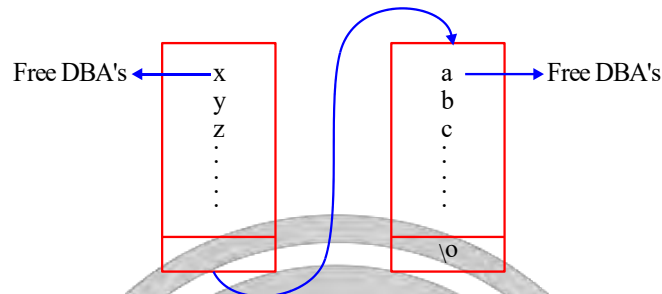
Size of Disk = 20 MB

DB Size = 1 KB

DBA = 16 Bits = 2 B

$$\# \text{ Disk blocks available on given disk} = \frac{\text{Size of disk}}{\text{DB size}} = \frac{20 \times 2^{20} B}{2^{10} B} = 20 K$$

9.8.1 Free List Approach



- In the free list approach, some disk blocks are used just to store the free disk block addresses.

$$\# \text{ Disk block addresses possible to store in one disk block} = \frac{\text{DB Size}}{\text{DBA}} = \frac{2^{10}}{2} = 2^9$$

2^9 addresses \rightarrow 1 disk block

$$20 K \text{ addresses} \rightarrow \frac{1}{2^9} \times 2^{11} \times 10 \rightarrow 2^2 \times 10 = 40 \text{ disk block}$$

9.8.2 Bit Map Approach

- In Bit map approach, every disk block were be mapped with 1 binary bit

Free Disk Blocks

0, 2, 4, 6, 9, 11,

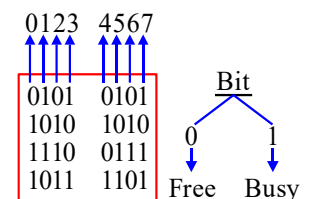
Busy Disk Blocks

1, 3, 5, 7, 8, 10,

Disk blocks we can map in 1 disk block = 1 K \times 8 bits = 8 K bits.

8 K bits \rightarrow 1

$$20 K \text{ bits} \rightarrow \frac{1}{8K} \times 20K = 2.4 = 3$$



10

DISK SCHEDULING

10.1 Goal of Disk Scheduling

- Enhanced throughput.
- Minimize the average seek time of the disk.
- Fair chance to all the I/O requests.

10.2 Disk scheduling Algorithms

10.2.1 FCFS

It serves the first request in the queue.

10.2.2 SSTF

It selects the request with minimum seek time for the current head position.

10.2.3 SCAN or Elevator Algorithm

The disk arm starts from the current head position and moves towards the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.

10.2.4 C-SCAN

The disk arm starts from the current head position and moves towards the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and immediately returns to the beginning of the disk without servicing any requests at the return trip. The servicing then continues from the beginning of the disk again.

10.2.5 C-LOOK

It is similar to C-SCAN where the arm goes as far as the last request in each direction, then reverses the direction immediately, without first going all the way to the end of the disk.

Note:

- Performance depends on the number and types of requests.
- I/O requests can be influenced by the file allocation methods.
- Either SSTF or LOOK is a reasonable choice for the default algorithm.

10.2.6 Comparison Among the Disk Scheduling Algorithms

Algorithm	Advantages	Disadvantages
First Come First Served	<ul style="list-style-type: none"> • Easy to implement • Fair chance to all the I/O requests • It doesn't suffer from starvation 	<ul style="list-style-type: none"> • Seek time is not optimized. • It doesn't maximize throughput.
Shortest Seek Time First	<ul style="list-style-type: none"> • It tends to minimize the arm movement. • It has better throughput than FCFS. 	<ul style="list-style-type: none"> • It may suffer from starvation.
SCAN/LOOK	<ul style="list-style-type: none"> • It eliminates starvation. • It works well with light to heavy loads. 	<ul style="list-style-type: none"> • It is complex to implement. • Increased overhead. • It needs a directional bit to keep track of the head direction.
C-SCAN/C-LOOK	<ul style="list-style-type: none"> • No directional bit is required. • It works well with light to heavy loads. 	<ul style="list-style-type: none"> • It is complex to implement. • Increased overhead.

□□□



For more questions, kindly visit the library section: Link for web: <https://smart.link/sdfez8ejd80if>



PW Mobile APP: <https://smart.link/7wwosivoicgd4>