OS is resource management responsible for allocation, Protection, reclamation

virtualization – illusion. Virtual memory

Finite resources with computing demands.

Allocating memories

Protection – you cant hurt me I wont hurt you. Memory should be alocated but protected also.

Os Is also responsible for that.

CODE that sits between different programs and hardware- hardware management, and different users-multiprogramming.

Services –

Abstraction

Simplification

Convenience

Standardization

PROCESS MANAGEMENT:

Process

- Process is a program in execution.
- To understand the importance of this definition, let's imagine that we have written a program called **my_prog.c** in C.
- On execution, this program may read in some data and output some data. When a **program** is written and a file is prepared, it is still a script. It **has no dynamics of its own** i.e, it cannot cause any input processing or output to happen.
- Once we compile, and still later when we run this program, the intended operations take place. In other words, a program is a text script with no dynamic behavior. When a **program is in execution**, the **script is acted upon**. It can result in engaging a processor for some processing and it can also engage in I/O operations. It is for this reason a process is differentiated from program.
- While the program icution is a process

Process is active entity but program has passive entity.

Program has longer life span. But process has shorter life span.

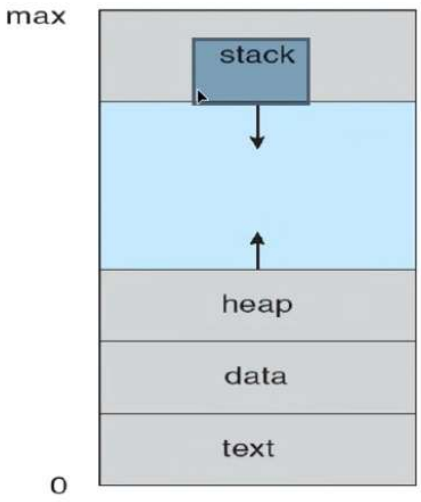
Process needs many resources(CPU TIME, MEM FILE, IO DEVICES) but program needs just space.

The resources are allocated when it is created or when it is executed.

Process

- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
- Program becomes process when executable file loaded into memory
- One program can be several processes (Consider multiple users executing the same program)
- **Multiple parts**
  - The program code, also called **text section,** also includes the current activity represented by the value of the **Program Counter**.
  - **Stack** containing temporary data (Function parameters, return addresses, local variables)
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

Process in Memory



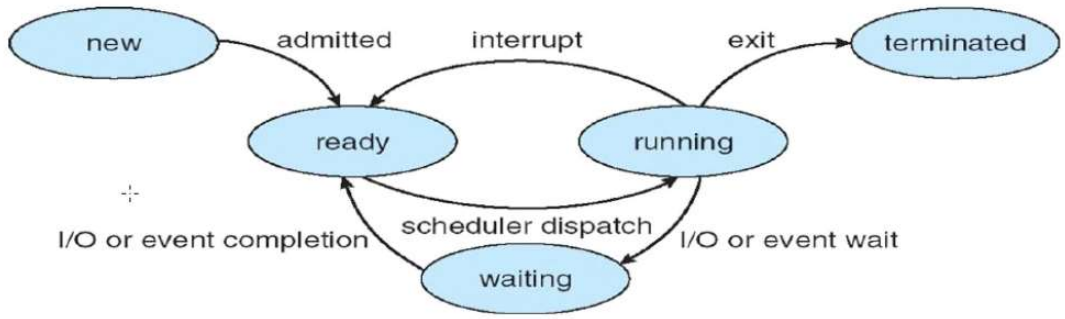
Need of Process Management

- Process needs some resources at the time of execution.
- When more than two processes needs same resources at the same time that time system became inconsistence or deadlock occur.
- To prevent this situation there is need of Operating System.
- Subtask of OS for process management
  - Process Synchronization
  - Process Communication
  - Process Scheduling
  - Suspend and resume process
  - Creating and deleting both user and system processes.

Process State

- A process goes through a series of process states for performing its task.
- As a process executes, it changes state.
- Various events can cause a process to change state.
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

Diagram of Process State



There are scheduling algo to decide which process to be chosen first.

Suspended ready state: while the processes are waiting due to lower priority. During this main memory might be full if there are a lot of processes are waiting in ready state. If so, to make space in memory, transfers processes w low priority from ready state to suspended ready state. So from main memory to secondary mem. And whenever memory is available, OS transfers it back.

Suspended block/waiting state: similar.

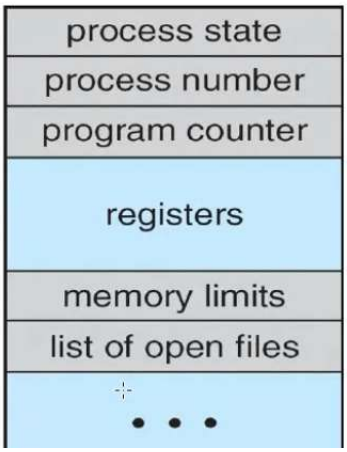
No ready state(for uniprogramming)

Process Control Block

- Process Control Block (PCB) is a data structure used by operating system to store all the information about a process.
- It is also known as Task Control Block.
- When a process is created, the operating system creates a corresponding PCB.
- Information in a PCB is updated during the transition of process states.
- When a process terminates, its PCB is released.
- Each process has a single PCB.

Process Control Block

- The PCB of a process contains the following information:



Process Control Block

- Attributes which are stored in the PCB are described below.
- **Process ID**
  - When a process is created, a unique id is assigned to the process which is used for unique identification of the process in the system.
- **Program counter**
  - A program counter stores the address of the last instruction of the process on which the process was suspended. The CPU uses this address when the execution of this process is resumed.
- **Process State**
  - The Process, from its creation to the completion, goes through various states which are new, ready, running and waiting.

Process Control Block

- **Priority**
  - Every process has its own priority. The process with the highest priority among the processes gets the CPU first. This is also stored on the process control block.
- **General Purpose Registers**
  - Every process has its own set of registers which are used to hold the data which is generated during the execution of the process.
- **List of open files**
  - During the Execution, Every process uses some files which need to be present in the main memory. OS also maintains a list of open files in the PCB.

Process Scheduling

- In multiprogramming, several processes are kept in main memory so that when one process is busy in I/O operation, other processes are available to CPU.
- In this way, CPU is busy in executing processes at all times.
- This method of selecting a process to be allocated to CPU is called Process Scheduling.
- Types of Schedulers
  - Long Term Scheduler
  - Short Term Scheduler
  - Medium Term Scheduler

Long Term Scheduler

- Long term scheduler is also known as **job scheduler**.
- It chooses the processes from the pool (secondary memory) and keeps them in the ready queue maintained in the primary memory.
- Long Term scheduler mainly controls the **degree of Multiprogramming**.
- The purpose of long term scheduler is to choose a perfect mix of IO bound and CPU bound processes among the jobs present in the pool.
- If the job scheduler chooses more IO bound processes then all of the jobs may reside in the blocked state all the time and the CPU will remain idle most of the time. This will reduce the degree of Multiprogramming.
- Therefore, the Job of long term scheduler is very critical and may affect the system for a very long time.

Long Term Scheduler

- Creating or moving process from secondary memory to system memor (RAM).
- Controls the degree of multiprogramming.
  - No of active processes in RAM.
  - How many processes will be transferred to RAM at any point of time.

Short Term Scheduler

- Short term scheduler is also known as **CPU scheduler**.
- It selects one of the Jobs from the ready queue and dispatch to the CPU for the execution.
- A scheduling algorithm is used to select which job is going to be dispatched for the execution.
- The Job of the short term scheduler can be very critical in the sense that if it selects job whose CPU burst time is very high then all the jobs after that, will have to wait in the ready queue for a very long time.
- This problem is called **starvation** which may arise if the short term scheduler makes some mistakes while selecting the job.

Short Term Scheduler

- Selecting one process from ready state and schedule it to running state. (On the basis of CPU Scheduling Algorithm).
- With the help of dispatcher.

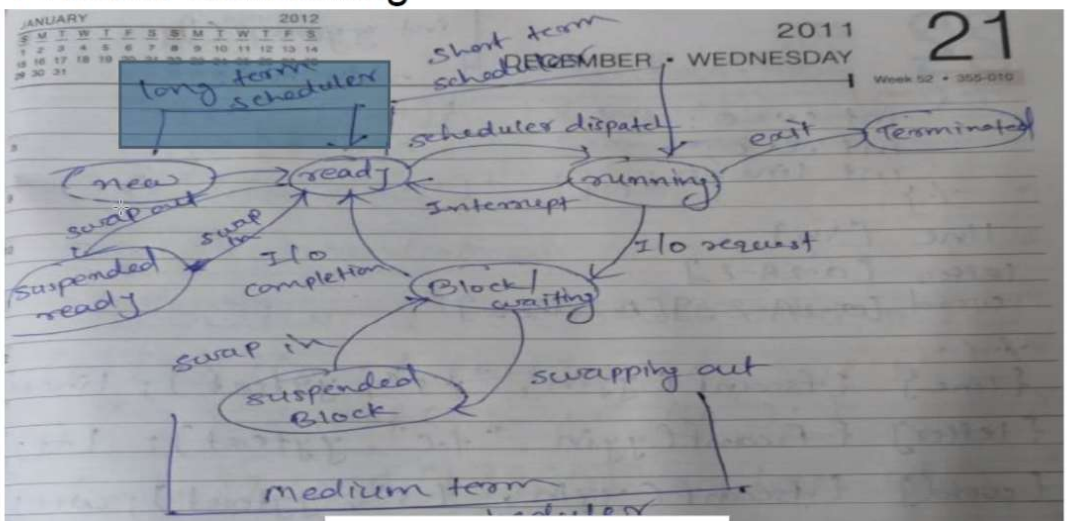
Medium Term Scheduler

- Medium term scheduler takes care of the swapped out processes.
- If the running state processes needs some IO time for the completion then there is a need to change its state from running to waiting.
- Medium term scheduler is used for this purpose. It removes the process from the running state to make room for the other processes. Such processes are the **swapped out** processes and this procedure is called **swapping.**
- The medium term scheduler is responsible for suspending and resuming the processes.
- It **reduces the degree of multiprogramming**. The swapping is necessary to have a perfect mix of processes in the ready queue.

Medium Term Scheduler

- Decreases degree of multiprogramming.
- Swapping processes from RAM to secondary and vice versa.
  - i.e. From block to suspended block and back.
    and from ready to suspended ready and back.

Process Scheduling



Process Queues

- There are many scheduling queues that are used in process scheduling.
  - **Job queue –** set of all processes in the system
  - **Ready queue –** set of all processes residing in main memory, ready and waiting to execute
  - **Device queues –** set of processes waiting for an I/O device
- Processes migrate among the various queues
- When the processes enter the system, they are put into the job queue. The processes that are ready to execute in the main memory are kept in the ready queue. The processes that are waiting for the I/O device are kept in the I/O device queue.

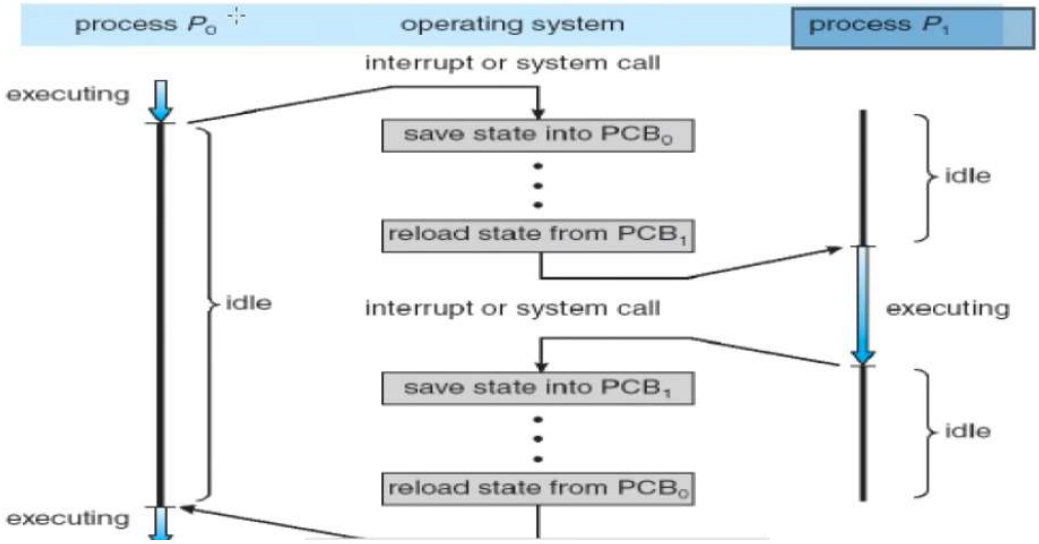
Jobs by OS is:

- Dispatching (Ready to Run State)
- I/O Event handling (Run to Block State)
- I/O Completion (Block to Ready Queue)

Context Switching

- Switching the CPU from one process to another process requires saving the state of old process and loading the saved state of new process.
- This task is known as **Context Switch.**
- When context switch occurs, operating system saves the context of old process in its PCB and loads the saved context of the new process.

Context Switching



Context Switching

- Saving of data of one process (deallocating CPU) and loading of data of an another process (allocating CPU) is called context switch.
- **It is an overhead (CPU ideal)**
- Depends upon hardware.
- Time required for this loading and saving is **context switching time**.
- **Dispatcher** is responsible for context switch.

➢ **Context switching depends upon:**

- Degree of multiprogramming:
  Degree ↑ context switch ↑
  Therefore CPU Throughput ↓
- Burst Time ↑ context switch ↑
- Time quantum is inversely proposonal to context switch.

System Calls

- Generally, system calls are made by the user level programs in the following situations:
  - Creating, opening, closing and deleting files in the file system.
    Example: open(), read(), write(), close()
  - Creating and managing new processes.
    Example: fork(), exit(), wait()
  - Creating a connection in the network, sending and receiving packets.
    Example: pipe(), shmget(), mmap()
  - Requesting access to a hardware device, like a mouse or a printer.

Operational Dual Mode

- CPU spot two modes of operations
- **Kernel Mode (mode bit 0)**
  - Privileged mode
  - Supervisor mode
  - Monitor mode
  - System mode
- **User Mode (mode bit 1)**
  - Non-privileged mode

Monolithic kernel:

Micro kernel:

Adv: if to add any process no need to change entire space, User and kernal services become independent, if program crashes system will not halt.

Dis:

Hybrid kernel:

Multiprogramming Environment of CPU

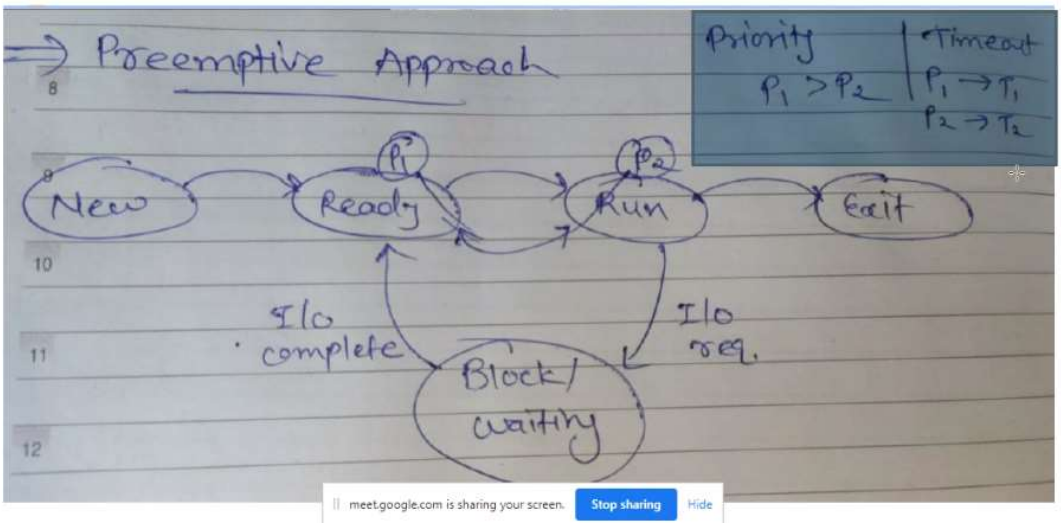
- **Non-Preemption Multiprogramming**
  - Process releases CPU:-
    i) After completion
    ii) I/o event request occur
  - Starvation occur and No priority set here
- **Preemption (Preemtive) Multiprogramming**
  - Used in Multitasking (Time sharing) and Priority present
  - Process releases CPU
    i) Completion
    ii) I/o request occur
    iii) Time out (time quantum Tq)
  - Preemption is for CPU not for I/o devices

Non-preemptive State Model

- Even important process has to wait and again go to ready queue because no priority.

Process releases cppu only when 1) process is finished or any ip/op is needed.

Preemptive State Model

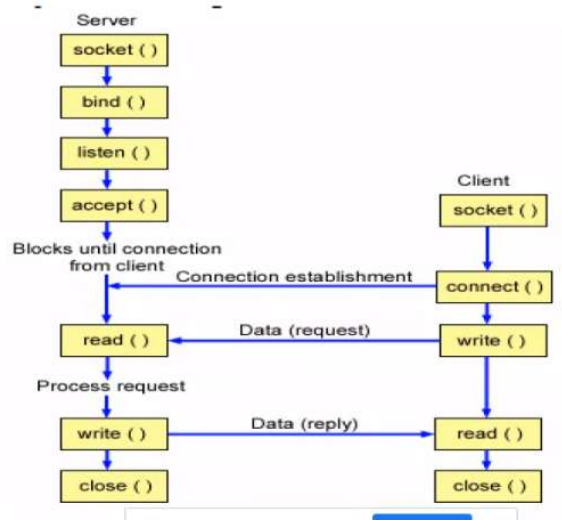


fork() System Call

- System call **fork()** is used to create processes. It takes no arguments and returns a process ID.
- The purpose of **fork()** is to create a ***new*** process, which becomes the *child* process of the caller. After a new child process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**:

- If **fork()** returns a **negative** value, the creation of a child process was unsuccessful.
- **fork()** returns a **zero** to the newly created child process.
- **fork()** returns a **positive** value, the ***process ID*** of the child process, to the parent.
- Normally, the process ID is an integer. Moreover, a process can use function **getpid()** to to this process.

System Call for Connection Oriented communication



System Call for Connection Oriented communication

```
#include <sys/types.h>
#include <sys/socket.h>
```

socket Function

```
int socket (int family, int type, int protocol);
```

family: specifies the protocol family {AF_INET for TCP/IP}

type: indicates communications semantics

SOCK_STREAM	stream socket	TCP
SOCK_DGRAM	datagram socket	UDP
SOCK_RAW	raw socket	

protocol: set to 0 except for raw sockets

returns on success: socket descriptor {a small nonnegative integer}

on error: -1

Example:

```
if (( sd = socket (AF_INET, SOCK_STREAM, 0)) < 0)
err_sys ("soci
```

System Call for Connection Oriented communication

bind Function

```
int   bind (int sockfd, const struct sockaddr  *myaddr,
socklen_t  addrlen);
```

bind assigns a local protocol address to a socket.

protocol address: a 32 bit IPv4 address and a 16 bit TCP or UDP port number.

sockfd: a socket descriptor returned by the socket function.

**myaddr*: a pointer to a protocol-specific address.

addrlen: the size of the socket address structure.

Servers bind their "well-known port" when they start.

returns on success: 0

on error: -1

Example:

```
if (bind (sd, (struct sockaddr *) &servaddr, sizeof (servaddr)) != 0)
    errsys ("bind
```


System Call for Connection Oriented communication

listen Function

```
int  listen (int  sockfd, int backlog);
```

listen is called **only** by a TCP server and performs two actions:

1. Converts an unconnected socket (*sockfd*) into a passive socket.
2. Specifies the maximum number of connections (*backlog*) that the kernel should queue for this socket.

listen is normally called before the accept function.

returns on success: 0

on error: -1

Example:

```
if (listen (sd, 2) != 0)
    errsys ("l
```

System Call for Connection Oriented communication

accept Function

```
int accept (int sockfd, struct sockaddr *cliaddr, socklen_t
*addrlen);
```

accept is called by the TCP server to return the next completed connection from the front of the completed connection queue.

sockfd: This is the same socket descriptor as in listen call.

**cliaddr*: used to return the protocol address of the connected peer process (i.e., the client process).

**addrlen*: {this is a value-result argument}

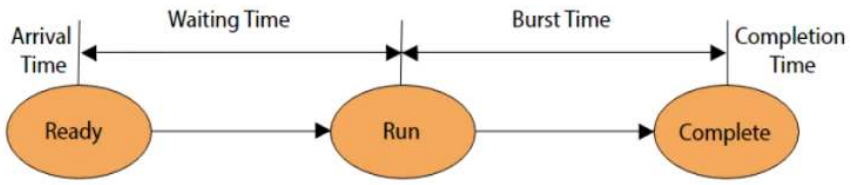
before the accept call: We set the integer value pointed to by *addrlen to the size of the socket address structure pointed to by *cliaddr;

on return from the accept call: This integer value contains the actual number of bytes stored in the socket address structure.

returns on success: a **new** socket descriptor

on error: -

CPU Scheduling



- **Arrival time**: The time at which the process enters into the ready queue is called the arrival time.
- **Completion time**: The Time at which the process enters into the completion state or the time at which the process completes its execution, is called completion time.

CPU Scheduling

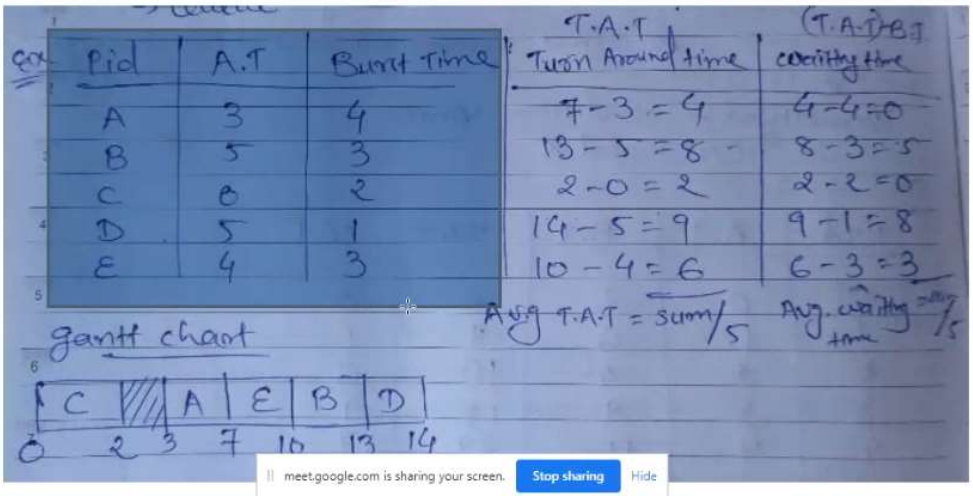
- **Waiting time**: Process waiting for CPU in its whole life **OR** The Total amount of time for which the process waits for the CPU to be assigned is called waiting time.
- **Response time**: First time CPU interaction **OR** The difference between the arrival time and the time at which the process first gets the CPU is called Response Time.
- **Burst time (Execution time)**: The total amount of time required by the CPU to execute the whole process is called the Burst Time. This does not include the waiting time.
- **Turn Around time**: The total amount of time spent by the process from its arrival to its completion, is called Turnaround time.

  TAT = Completion time – Arrival time OR

  TAT = Burst time + Waiting time

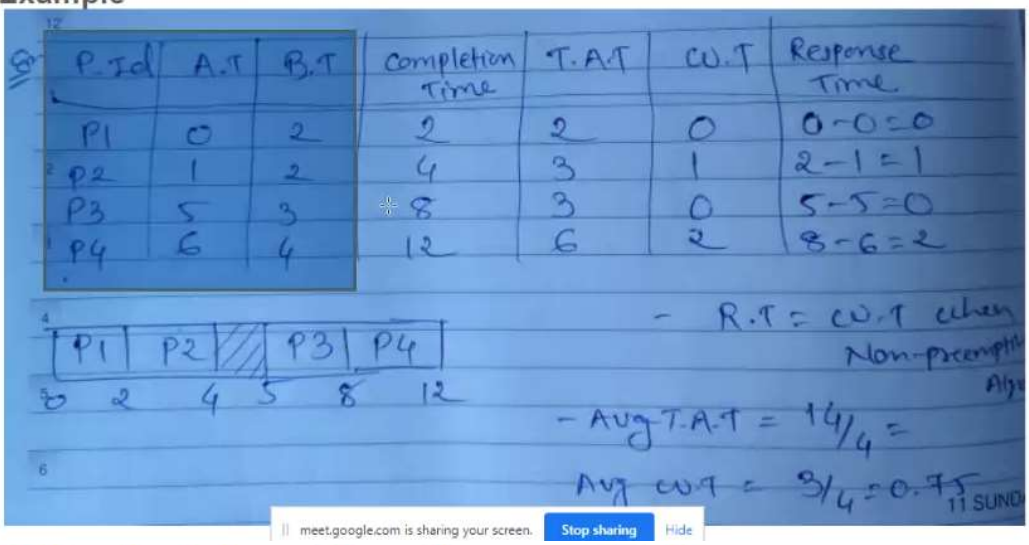
CPU Scheduling Algorithm - FCFS

- **Example**



CPU Scheduling Algorithm - FCFS

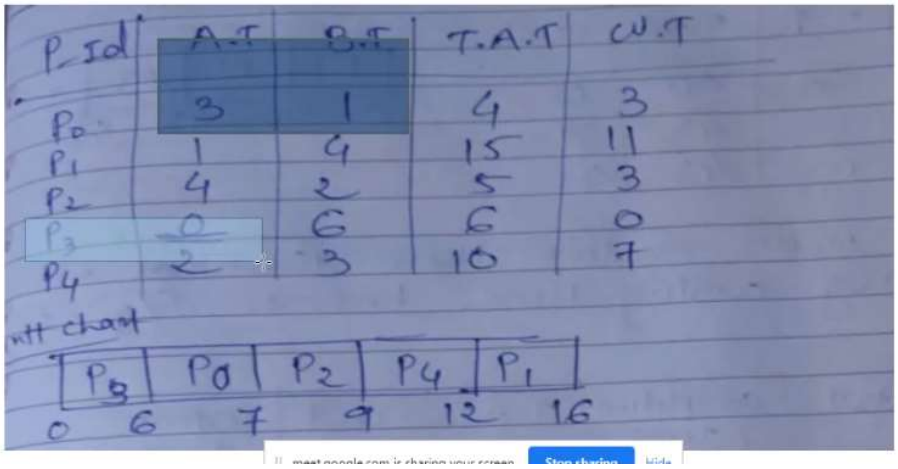
- **Example**



P.Id	A.T	B.T	Completion Time	T.A.T	W.T	Response Time
P1	0	2	2	2	0	0-0=0
P2	1	2	4	3	1	2-1=1
P3	5	3	8	3	0	5-5=0
P4	6	4	12	6	2	8-6=2

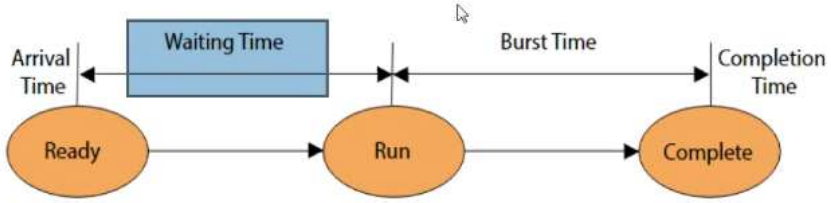
CPU Scheduling Algorithm – SJF (Non-preemptive)

- **Example:**



P.Id	A.T	B.T	T.A.T	W.T
P0	3	1	4	3
P1	1	4	15	11
P2	4	2	5	3
P3	0	6	6	0
P4	2	3	10	7

CPU Scheduling



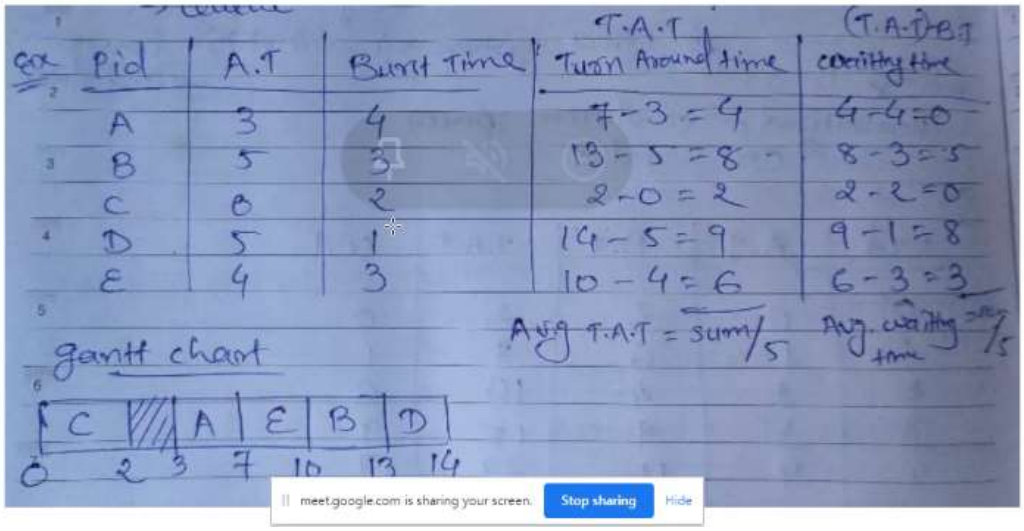
- **Arrival time**: The time at which the process enters into the ready queue is called the arrival time.
- **Completion time**: The Time at which the process enters into the completion state or the time at which the process completes its execution, is called completion time.

CPU Scheduling Algorithm

- **FCFS**- First Come First Serve (Non primitive)
  - FCFS scheduling algorithm simply schedules the jobs according to their **arrival time**.
  - The job which comes first in the ready queue will get the CPU first. The lesser the arrival time of the job, the sooner will the job get the CPU.
  - Always **Non-preemptive**
  - FIFO- First in First out (QUEUE)
  - Used in background process like date and time (Not urgent processes)

CPU Scheduling Algorithm - FCFS

- **Example**



Pid	A.T	Burst Time	T.A.T (Turn Around time)	(T.A.T - B.T) (waiting time)
A	3	4	7 - 3 = 4	4 - 4 = 0
B	5	3	13 - 5 = 8	8 - 3 = 5
C	0	2	2 - 0 = 2	2 - 2 = 0
D	5	1	14 - 5 = 9	9 - 1 = 8
E	4	3	10 - 4 = 6	6 - 3 = 3

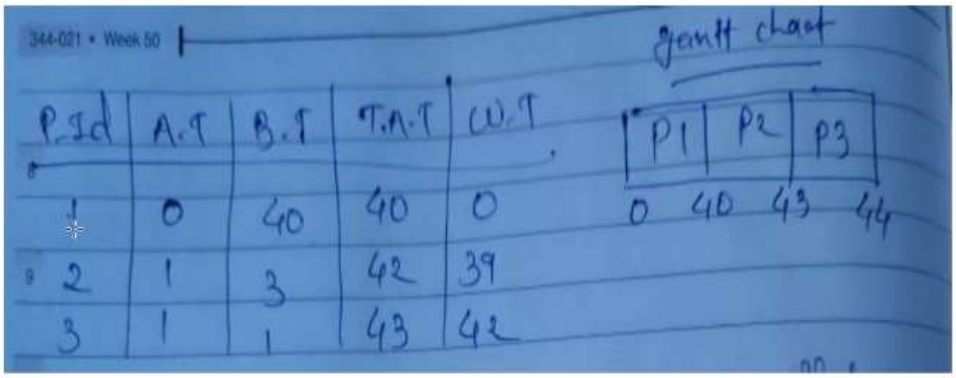
Avg T.A.T = sum/5

Avg. waiting time = sum/5

gantt chart

CPU Scheduling Algorithm - FCFS

- **Example**

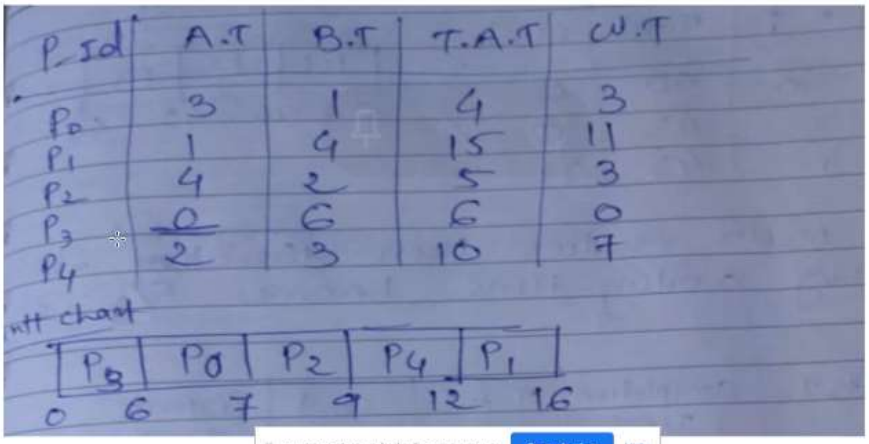


CPU Scheduling Algorithm - SJF

- **SJF- Shortest Job First**
  - Basis – Burst time
  - Non-preemptive
  - Better Average waiting time than FCFS
  - Starvation occur in SJF.

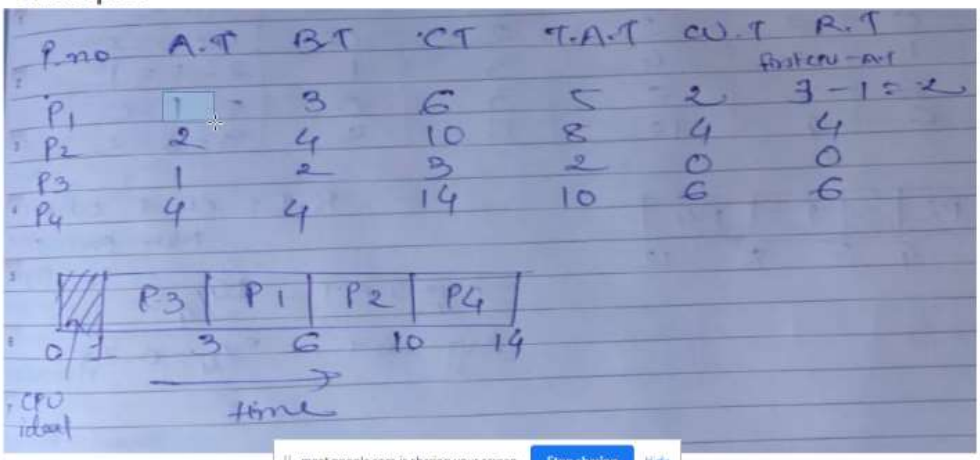
CPU Scheduling Algorithm – SJF (Non-preemptive)

- **Example:**



CPU Scheduling Algorithm – SJF (Non-preemptive)

- **Example:**

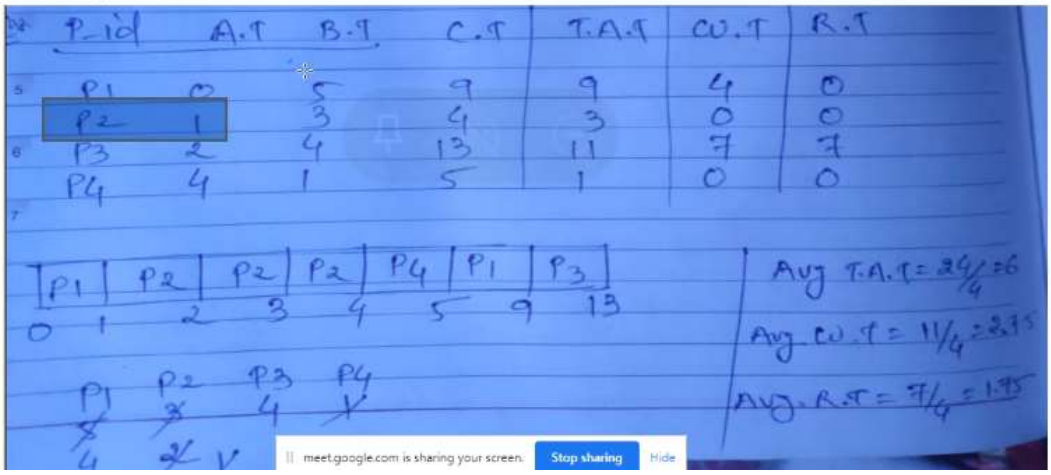


CPU Scheduling Algorithm – SJF Preemptive

- **SRTF- Shortest Remaining Time First (Preemptive)**
- Basis – Burst time
- Purely greedy approach
- Average waiting time is optimal (minimum).

CPU Scheduling Algorithm – SJF (Preemptive)

- **Example:**

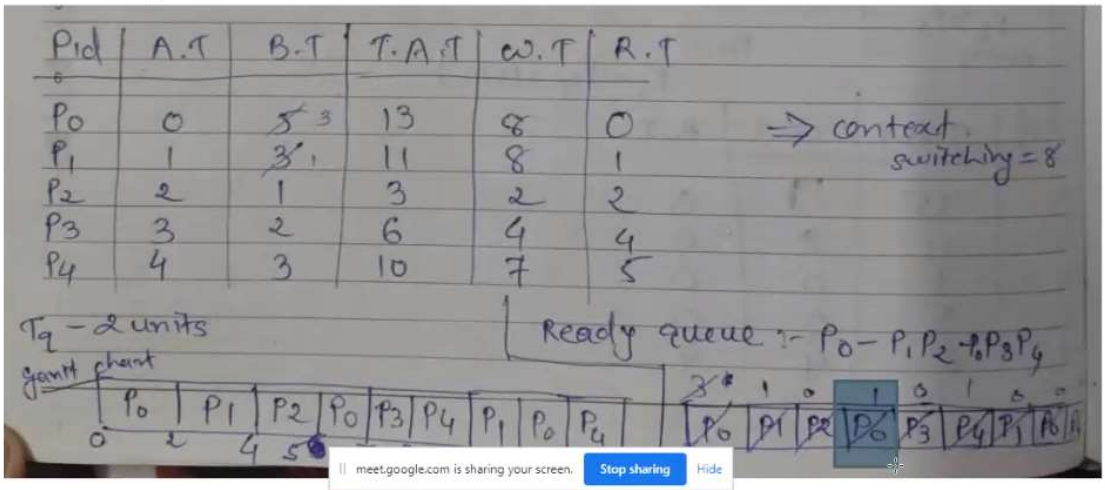


CPU Scheduling Algorithm – Round Robin

- **Round Robin Algorithm (Time Sharing system)**
- Always Pre-emptive
- Best Average Response time
- Based on time quantum, circular queue to divide the time between all process in ready state to share CPU
- Best for interactive processes
- If the process is executing before time quantum then it will not take full time quantum
- Tq = unit of time
- Ready queue is circular queue

CPU Scheduling Algorithm – Round Robin

- **Example 2**



CPU Scheduling Algorithm – Round Robin

- **Disadvantages**
- Depends heavily on time quantum
- Time quantum is inversely proportional to context switching
  If time quantum is high then context switching is less and throughput increase but In this case RR nearly equal to FCFS algorithm
- The higher the time quantum, the higher the response time in the system.
- The lower the time quantum, the higher the context switching overhead in the system.
- Deciding a perfect time quantum is really a very difficult task in the system.

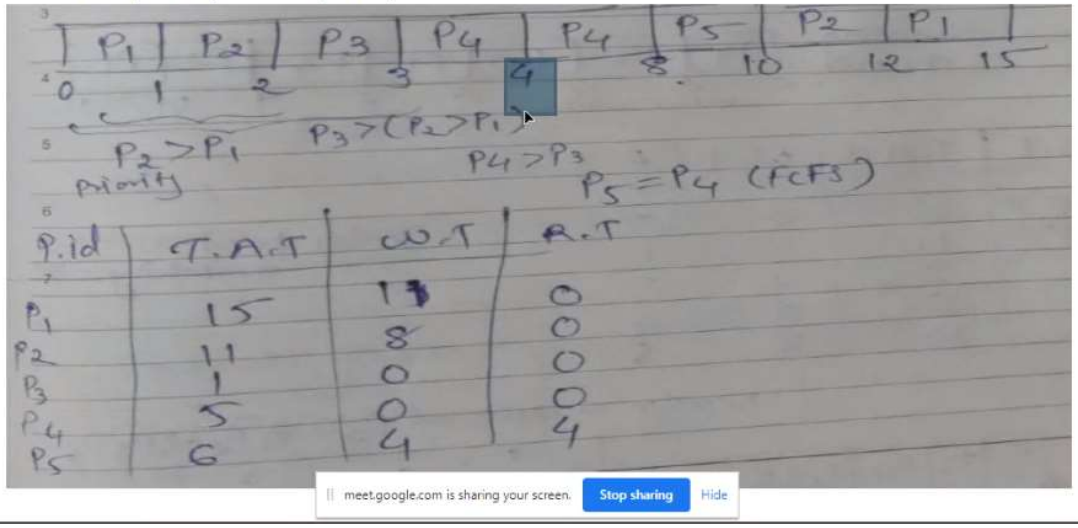
CPU Scheduling Algorithm – Priority Scheduling

- **Priority Scheduling Algorithm**
  - High value – High priority or vice versa
  - No importance given to Arrival time or Burst time
  - Tie then use FCFS

- **Disadvantage**
  - Small priority processes may suffer from starvation
  - Solution to starvation - **Ageing** is a scheduling technique used to avoid starvation. Aging is used to gradually increase the priority of a task, based on its waiting time in the ready queue.

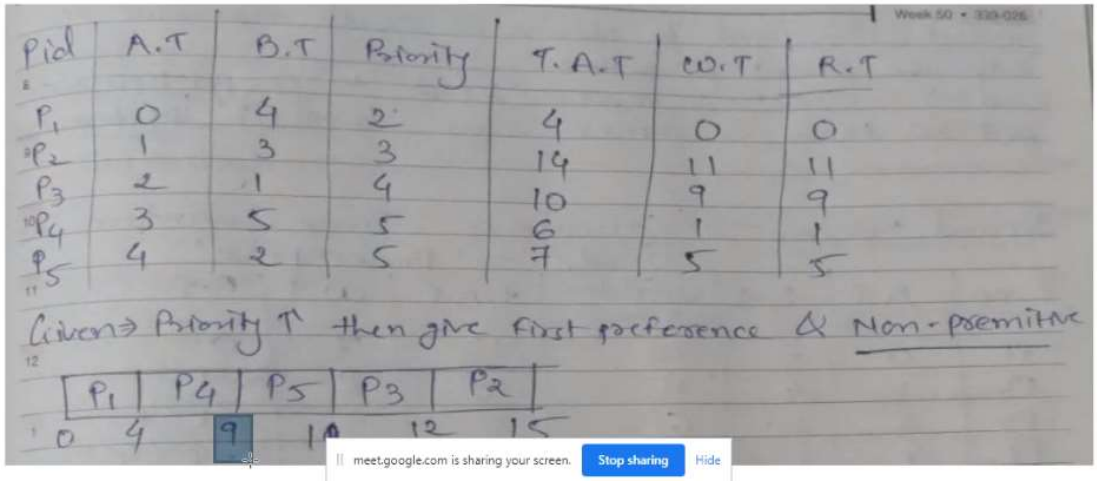
CPU Scheduling Algorithm – Priority Scheduling

- **Example 2 (Pre emptive)**



CPU Scheduling Algorithm – Priority Scheduling

- **Example 2 (Non-pre emptive)**



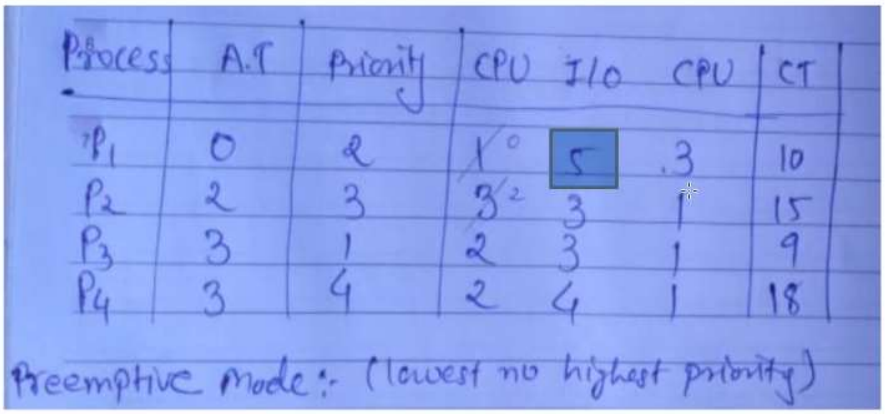
Pid	A.T	B.T	Priority	T.A.T	W.T	R.T
P_1	0	4	2	4	0	0
P_2	1	3	3	14	11	11
P_3	2	1	4	10	9	9
P_4	3	5	5	6	1	1
P_5	4	2	5	7	5	5

Given ⇒ Priority ↑ then give first preference & Non-premitive

P_1	P_4	P_5	P_3	P_2
0	4	9	12	15

CPU Scheduling Algorithm – Priority Scheduling

- **Example of Mix Burst Time (CPU and I/O both) in CPU Scheduling**

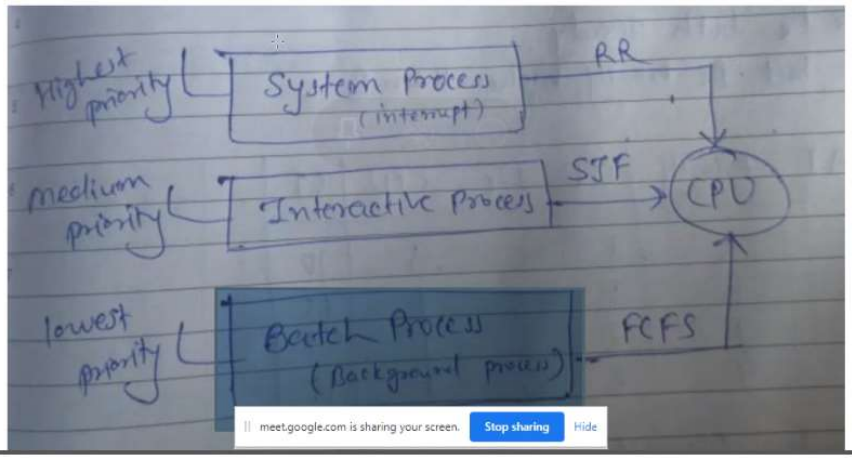


Process	A.T	Priority	CPU	I/O	CPU	CT
P_1	0	2	~~1~~ 0	5	3	10
P_2	2	3	~~3~~ 2	3	1	15
P_3	3	1	2	3	1	9
P_4	3	4	2	4	1	18

Preemptive Mode :- (lowest no highest priority)

CPU Scheduling – Multilevel Queue Scheduling

- Batching process acc. to there nature and executing acc. Different algorithms

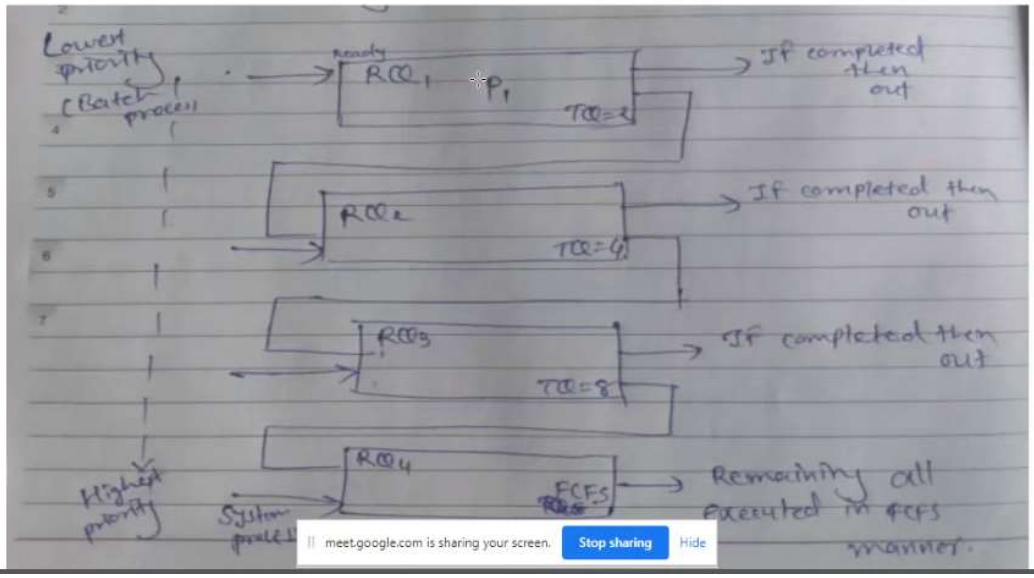


CPU Scheduling – Multilevel Queue Scheduling

- **Major Disadvantage**
  - Saturation may occur for low priority process because CPU will be busy executing system processes.

- To remove starvation we use **multilevel feedback queue** scheduling, where **aging** is used, queue of a low priority process keeps on changing to lead to a high priority queue and get execute.
- T(n+1) = x * tn + (1-x) * Tn
  where x = aging factor
  tn = burst Time
  Tn = value of previous stage

CPU Scheduling – Multilevel Feedback Queue



Process Vs Threads

Process	Threads (User Level)
System calls involved in process. i.e. fork()	There is no system calls involved
OS treats different process differently.	All user level thread treated as single task for OS.
Different process have different copies of data, files, code.	Threads share the same copies of code and data
Context switching is slower	Context switching is faster
Blocking a process will not block another process	Blocking a thread will block entire process
Independent	Interdependent

Types of Threads

- Basically there are two types of threads which are given as follows:
  - **User Level threads**
  - **Kernel Level threads**

- **User Level thread (ULT)**
  - Is implemented in the user level library, they are **not created using the system calls**. Thread switching does not need to call OS and to cause interrupt to Kernel.
  - Kernel does not know about the user level thread and manages them as if they were single threaded processes.

Types of Threads

- **Advantages of ULT -**
  - Simple representation since thread has only program counter, register set, stack space.
  - Simple to create since no intervention of kernel.
  - Thread switching is fast since no OS calls need to be made.
  - Can be implemented on an OS that does not support multithreading.

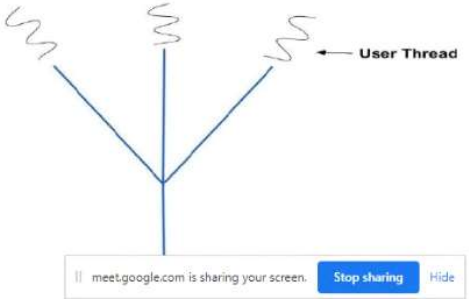
- **Disadvantages of ULT –**
  - No or less co-ordination among the threads and kernel.
  - If one thread causes a page fault, the entire process blocks
  - Since kernel has full knowledge about the threads in the system, scheduler may decide to give more time to processes having large number of threads.
  - Good for applicatior

Types of Threads

- **Kernel Level threads**
  - Kernel knows and manages the threads. Instead of thread table in each process, the kernel itself has thread table (a master one) that keeps track of all the treads in the system.
  - In addition kernel also maintains the traditional process table to keep track of the processes. OS kernel provides system call to create and manage threads.
- **Advantages of KLT –**
  - Since kernel has full knowledge about the threads in the system, scheduler may decide to give more time to processes having larger no of threads.
  - Good for applications that are frequently block.
- **Disadvantages of KLT –**
  - Slow and inefficient
  - It requires thread co

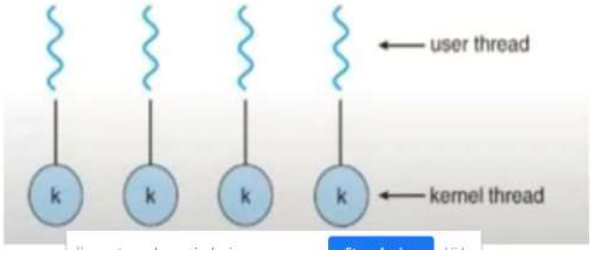
Multithreading Models

- **Many-To-One Model**
  - In the many-to-one model, many user level threads are mapped onto a single kernel thread.
  - Thread management is handled by the thread library in user space, which is very efficient.
  - However, if a blocking system call is made, then the entire process blocks, even if the other user threads would otherwise be able to continue.



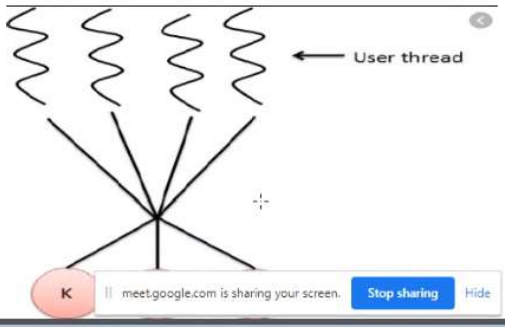
Multithreading Models

- **One-To-One Model**
  - The one-to-one model creates a separate kernel thread to handle each user thread.
  - One-to-one model overcomes the problems listed above involving blocking system calls and splitting of processes across multiple CPUs.
  - However the overhead of managing the one-to-one model is more significant, involving more overhead and slowing down the system.



Multithreading Models

- **Many-To-Many Model**
  - The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models
  - User have no restrictions on umber of threads created.
  - Blocking kernel system calls do not lock the entire process.



Multithreading Models

- **Note:**
  - Process are created using system calls called fork()
    If a parent process and child process are present O.S will treat it separate. It is heavy weighted. Context switching is slow.
  - Kernel Level thread will not block whole process on these blocking.
  - Context switching is slow in compare slow to user level thread.
- **Context switching time**
  - Process context switching time > Kernel level thread > User level thread

MEMORY MANAGEMENT

Memory Management

- Is the task carried out by the OS and hardware to accommodate several programs in main memory
- The more programs can be kept ready for execution in main memory, the more the CPU will be busy
  - resulting in less waiting time for applications
- Hence, memory needs to be allocated efficiently

9 Chapter 7

- Relocation
- Protection
- sharing
- Logical Org?
- Physical "

Memory Management Requirements

- **Relocation** Uni. Vs. Multi
  - programmer cannot know where the program will be loaded in memory when it is executed
  - a program may be (often) ***relocated*** in main memory due to swapping
  - swapping enables the OS to have a larger pool of ready-to-execute processes
  - memory references in code (for both instructions and data) must be translated to actual physical memory addresses

11 Chapter 7

Memory Management Requirements

- **Protection**
  - processes share memory
  - processes should not be able to reference memory locations in another process without permission
  - impossible to check addresses at compile time in programs since the program could be relocated
  - → address references must be checked at run time by hardware

* Relⁿ ⟷ Protection

* Protection must be satisfied by the processor (H|W) rather than OS (S|W).

Memory Management Requirements

- **Sharing**
  - must allow several processes to access a common portion of data or program without compromising protection
    - cooperating processes may need to share access to the same data structure
    - better to allow each process to access the same copy of the program rather than have their own separate copy

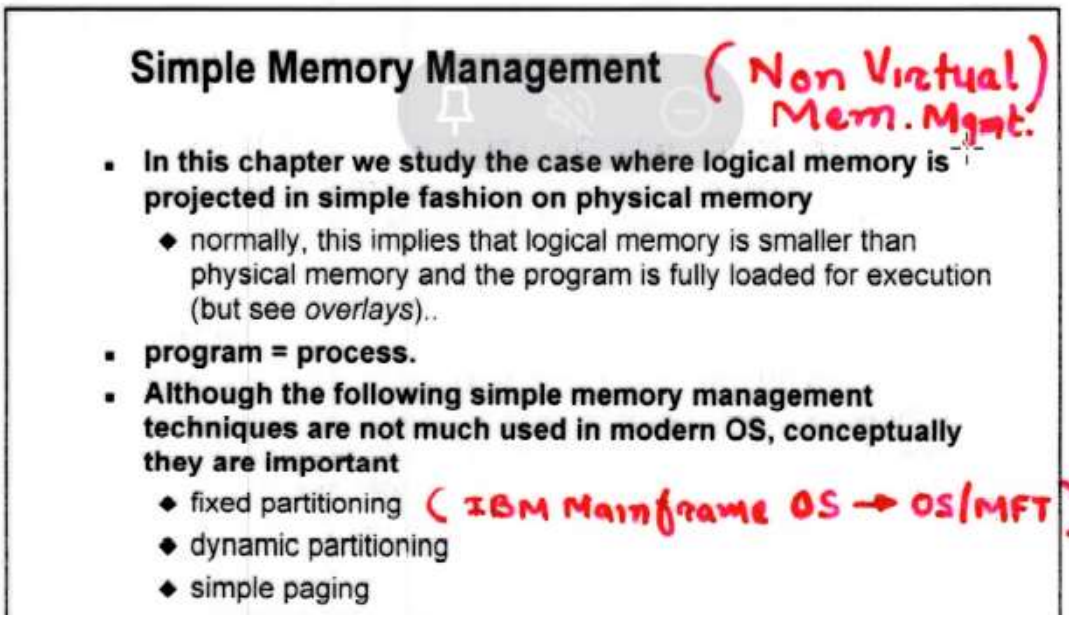
Thus, Prog. is seen as a set of Modules which gives above adv.

Memory Management Requirements

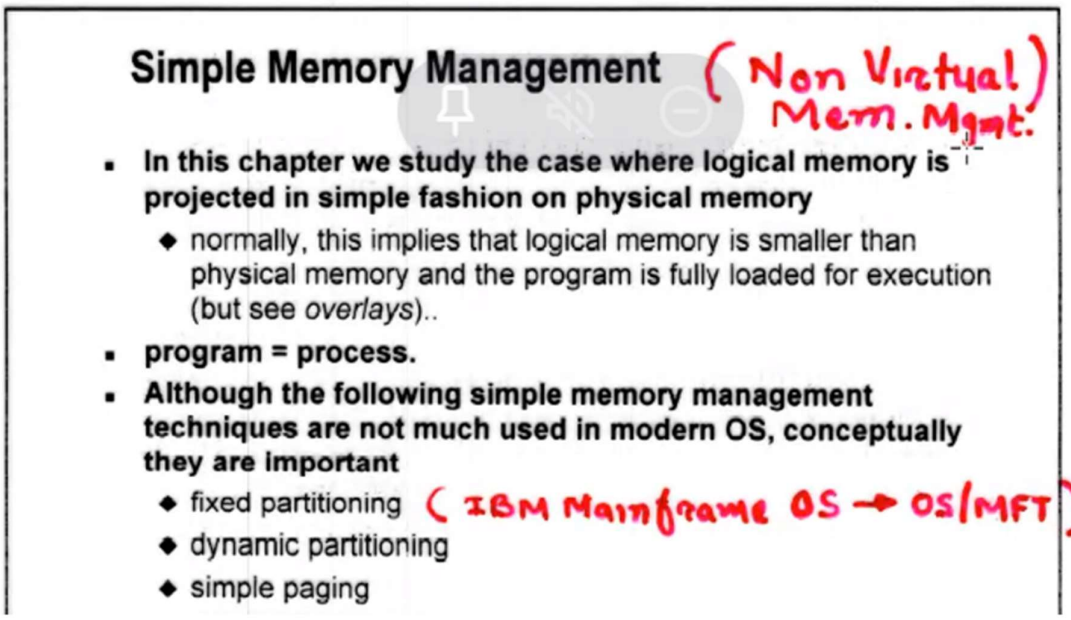
- **Physical Organization**
  - Memory hierarchy: there are several types of secondary memory: from slow and large to small and fast.
  - secondary memory (disk, etc) is the long term store for programs and data while main memory (RAM) holds program and data currently in use
  - moving information between levels of memory is a major concern of memory and file management (OS)
    - it's inefficient and risky to leave this responsibility to the programmer

Address translation mechanisms

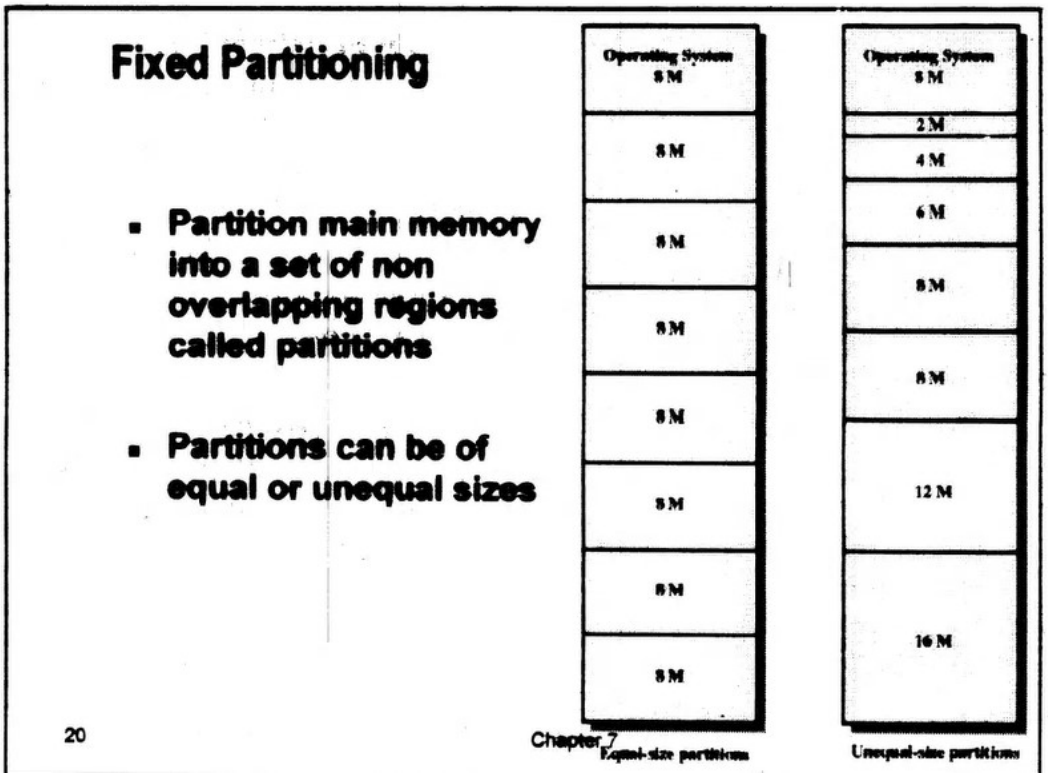
- **A number of address translation mechanisms are at work:**
  - compilation translates programmer names into addresses in modules
  - linkage editing, loading, perform other translations
  - there are also dynamic translation mechanisms implemented in hardware...



Simple segmentation



Simple segmentation



INTERNAL FRAGMENTATION: A program is assigned to one partition but not full mem is needed. The remaining mem is called internal fragmentation.

Inefficient use of mem.

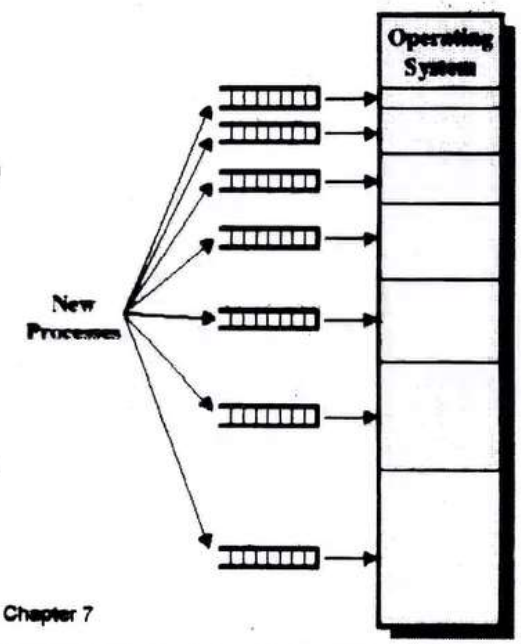
Program will work only if complete mem of program size is available in the mem. This is without virtual mem space.

Fixed Partitioning

- **any program whose size is less than or equal to a partition size can be loaded into the partition**
- **if all partitions are occupied, the operating system can swap a process out of a partition**
- **a program may be too large to fit in a partition. The programmer must then design the program with *overlays***
  - when the module needed is not present the *user program* loads that module into the program's partition, overlaying whatever program or data are there
- **If all partitions are occupied and a priority program arrives, then one of the programs in memory must be suspended to free a partition**

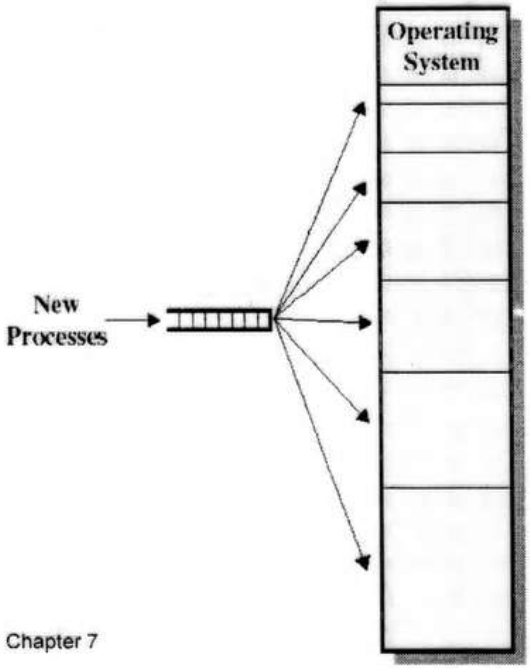
Placement Algorithm with Partitions

- **Unequal-size partitions: use of multiple queues**
  - assign each program to the smallest partition within which it will fit
  - A queue for each partition size
  - tries to minimize internal fragmentation
  - Problem: some queues will be empty if no program within a size range is present



Placement Algorithm with Partitions

- **Unequal-size partitions: use of a single queue**
  - When its time to load a process into main memory the *smallest available partition* that will hold the program is selected
  - increases the level of multiprogramming possibly at the expense of internal fragmentation



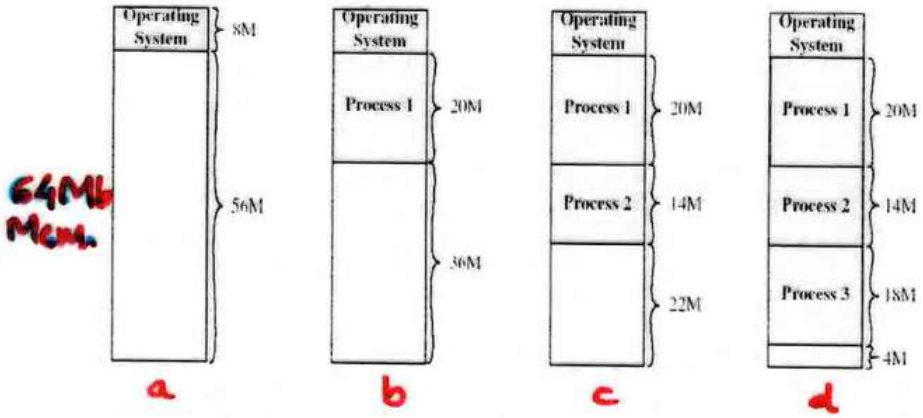
Fixed Partitioning

- **Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This is called *internal fragmentation.***
  - *internal fragmentation:* unused space in a partition
- **Unequal-size partitions lessens these problems but they still remain...**
- The No. of partitions specified at syst. gen. time limits the No. of active processes in the system.

Dynamic Partitioning :

- Each program is allocated exactly as much memory as it requires
- Partitions are of variable length and number
- Eventually holes are formed in main memory. This is called *external fragmentation*

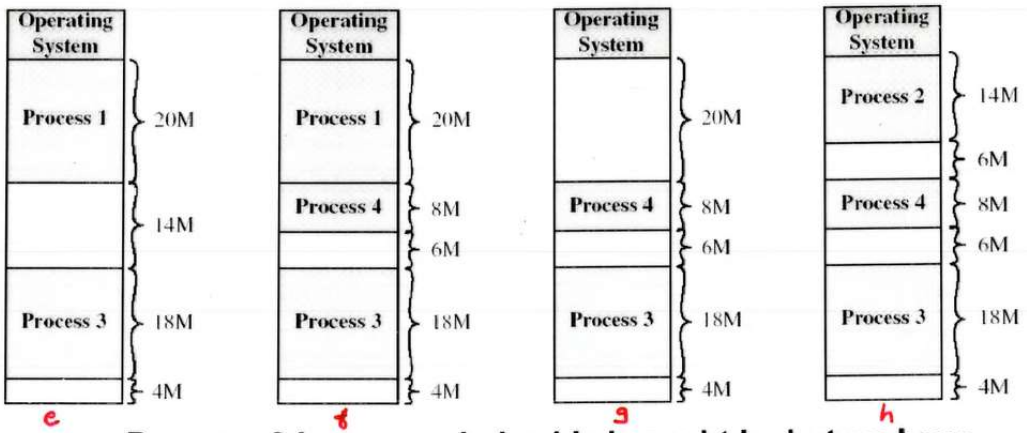
Dynamic Partitioning: an example



- A hole of 4M is left after loading 3 programs: not enough room for program 4 of 8M.
  - external fragmentation of 4M
- If all 3 processes become blocked, the OS can swap out program 2 (closest fit) to bring in program 4 = 8M

- Eventually holes are formed in main memory. This is called *external fragmentation*
- Periodically, must use *compaction* to shift programs so they are contiguous and all free memory is in one block

2.9. OS/MVT for IBM Mainframe



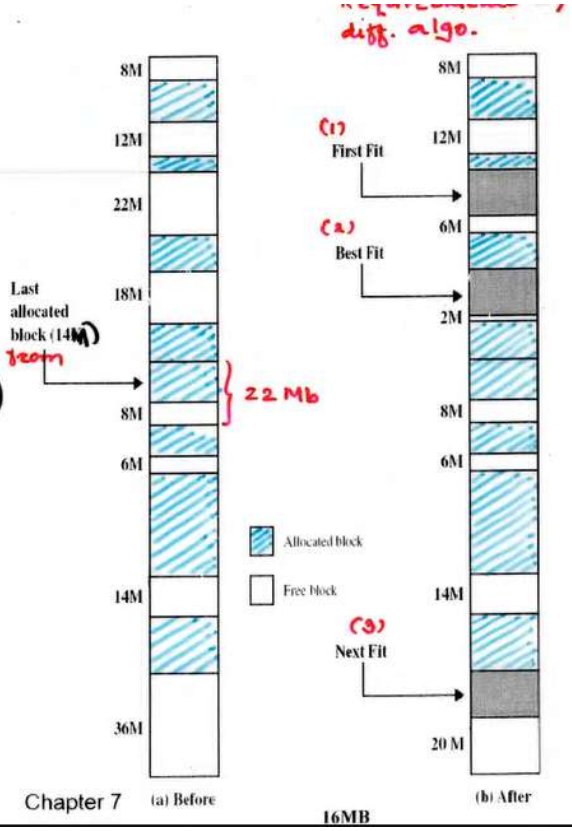
- Program 2 is suspended, p4 is brought in, but we have external fragmentation of 4+6 (f)
- Suppose that all programs become blocked. The OS may suspend program 1 (g) to bring in again program 2 (h) and another hole of 6M is created...

Compaction performed and 6+6+4-16mb space made.

Compaction is a very costly affair.

Placement Algorithm

- Used to decide which free block to allocate to a program
- Goal: to reduce usage of compaction (time consuming)
- Possible algorithms:
  - Best-fit: choose smallest hole
  - First-fit: choose first hole from beginning
  - Next-fit: choose first hole from *last placement*



Placement Algorithm: comments

- What is the best algorithm? Simulation showed that *first fit* usually is (see book)!
  - Best fit may seem to be optimal, but it creates small holes hence it exacerbates fragmentation!
  - First fit is also the most efficient and simplest to implement
- Next fit tends to generate holes towards the end of the memory

- **Next fit tends to generate holes towards the end of the memory**

- too small holes, so compaction is still req^d.

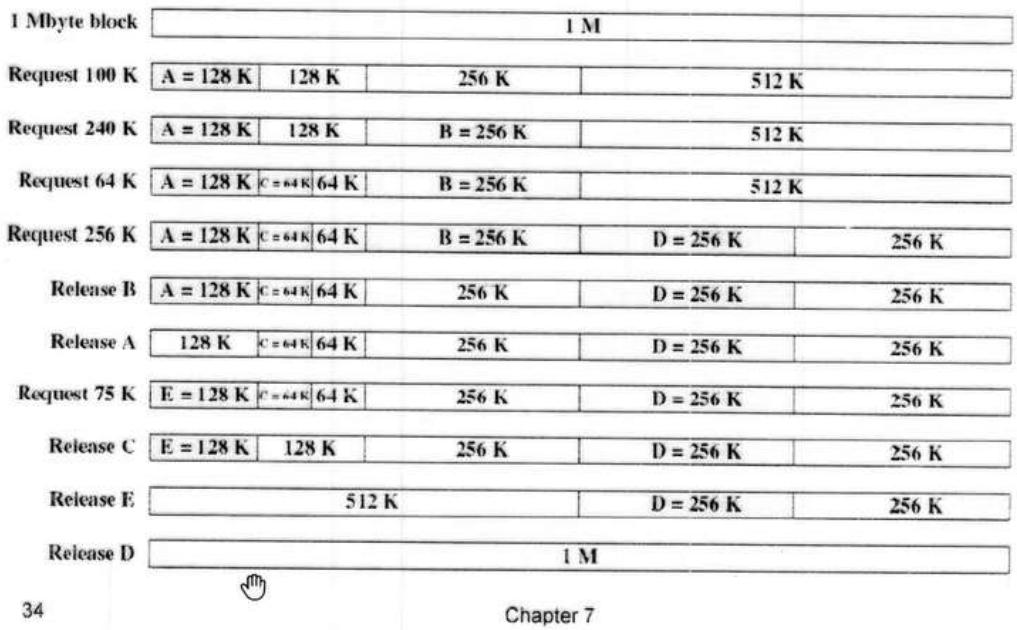
* Comb^n of static & Dynamic decisions.

We decide statically which sizes of blocks to allow (i.e. 2^n), and we decide dynamically how many of each size to have.

Buddy System

- **A reasonable *compromise* to overcome disadvantages of both fixed and variable partitioning schemes** → *
- **Modified forms are used in Unix and Linux** → Kernel Mem. Alloc^n.
- **Memory blocks are available in size of 2^{K} where L <= K <= U and where**
  - 2^{L} = smallest size of free block
  - 2^{U} = largest size of free block
  - (generally, the entire memory available)

Example of Buddy System



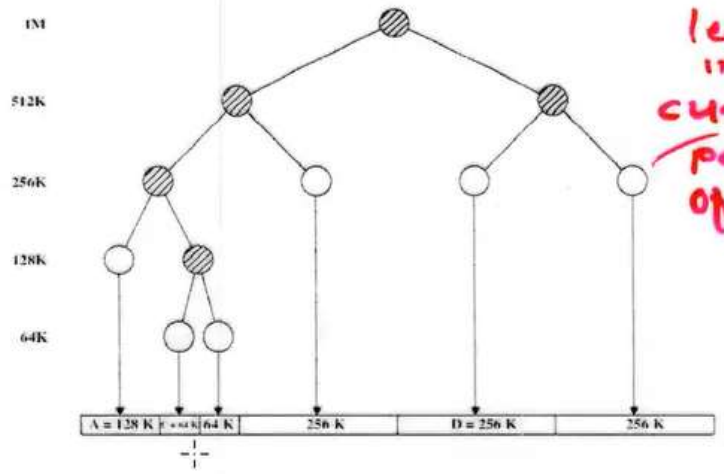
Buddy System

- **We start with the entire block of size 2^{U}**
- **When a request of size S is made:**
  - If 2^{U-1} < S <= 2^{U} then allocate the entire block of size 2^{U}
  - Else, split this block into two *buddies*, each of size 2^{U-1}
  - If 2^{U-2} < S <= 2^{U-1} then allocate one of the 2 buddies
  - Otherwise one of the 2 buddies is split again
- **This process is repeated until the smallest block greater or equal to S is generated**
- **Two buddies are merged whenever both of them become free**

Buddy System

- **The OS maintains several lists of holes**
  - the i-list is the list of holes of size 2^{i}
  - whenever a pair of buddies in the i-list occur, they are removed from that list and merged into a single hole in the (i+1)-list
- **Presented with a request for an allocation of size k such that 2^{i-1} < k <= 2^{i}:**
  - the i-list is first examined
  - if the i-list is empty, the (i+1)-list is then examined...

Tree representation



leaf nodes indicates current partitioning of the Mem.

Figure 7.7 Tree Representation of Buddy System

(after Release B request)

Advantages of buddy systems

- **Average memory waste (internal fragmentation) is 25%**
  - on the avg for each program there will be one full power of two partition, plus one half-used
  - plus of course wasted blocks if there aren't any programs waiting small enough for them.
- **Programs are not moved in memory**
  - this simplifies memory management and address translation.

• Application in 11^{el} syst.s as an efficient means of all^{n} & release for 11^{el} progs.

- **So far, we have only discussed *loading* concepts: how programs are loaded in physical memory.**
- **We have assumed that programs are loaded contiguously in memory.**
- **But programs can consist of several logical parts. These are the segments, or load modules, e.g.:**
  - one or more executable segments
  - one or more data segments
  - stack segment

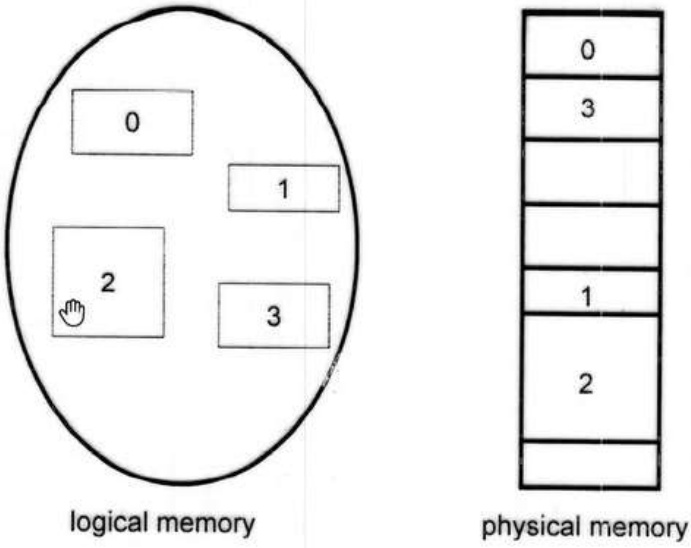
☝ **Idea: make allocation more flexible by allocating segments independently.**

Simple Segmentation

(Vs Dynamic Part 149)

- Each program is subdivided into blocks of non-equal size called segments
- these are program modules visible to programmer
- When a process gets loaded into main memory, its different segments can be located anywhere
- The methods for allocating memory to segments are those we have seen so far: just replace *program* by *segment*
- However in this chapter we consider only the case where memory is allocated to segments by using *dynamic partitioning.*
- We can use the methods already studied: e.g. buddy systems

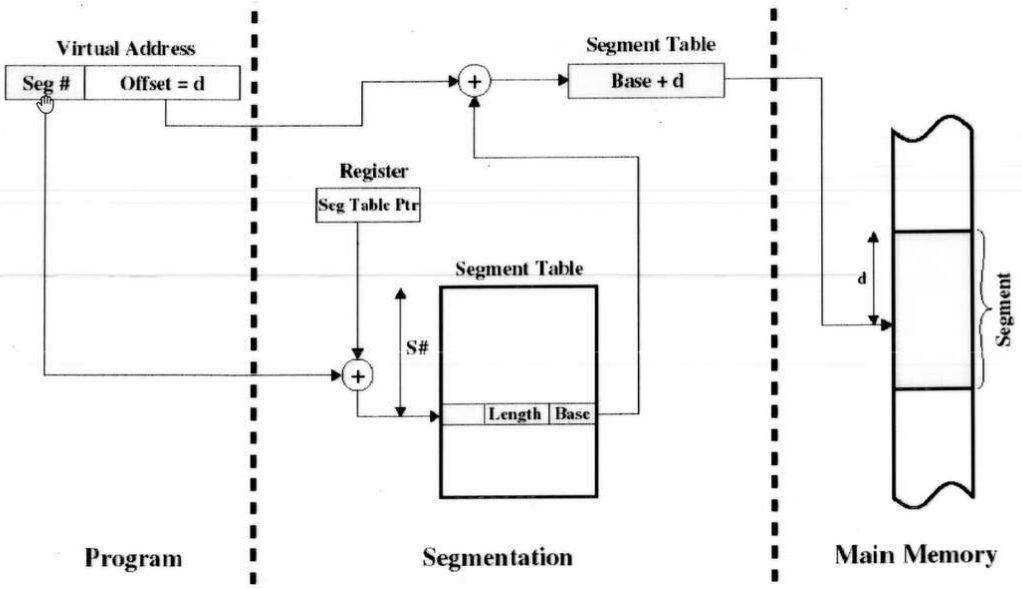
Segments as memory allocation units



Given that segments are smaller than whole programs, this technique implies less external fragmentation (small ice cubes will waste less space in a glass than large ones).

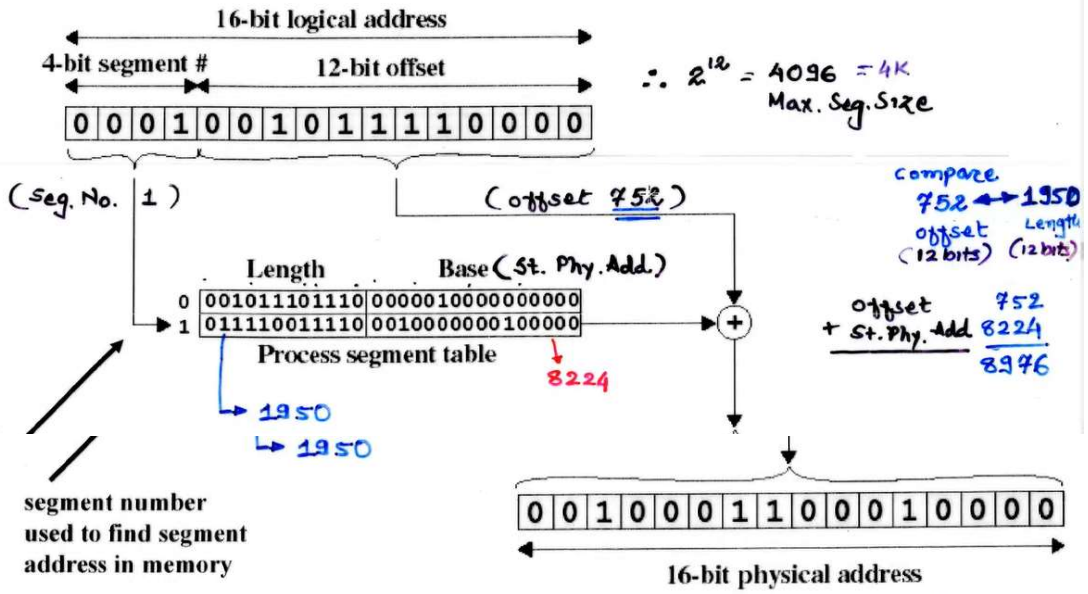
Simple Segmentation

- **The OS maintains a segment table for each program. Each entry contains:**
  - the starting physical addresses of that segment.
  - the length of that segment (for protection)



Logical-to-Physical Address translation in segmentation

with dynamic partitioning ([redacted])



(using addition)

Logical address used in simple segmentation with dynamic partitioning

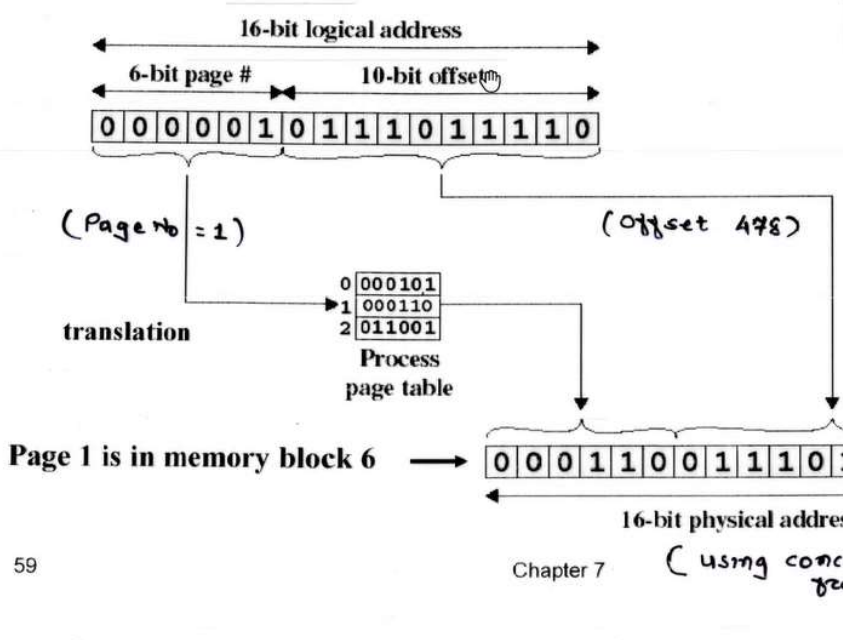
- **A CPU register holds the starting address of the segment table of the program which the CPU executes.**
- **Presented with a logical address (segment number, offset) = (n,m), the CPU indexes (with n) the segment table to obtain the starting physical address k and the length l of that segment**
- **The physical address is obtained by adding m to k**
  - the hardware also compares the offset m with the length l of that segment to determine if the address is valid

Evaluation of Simple Segmentation

- **Advantage: memory allocation unit is a logically independent portion of program**
  - Segments can be loaded individually on demand (*dynamic linking*).
- **Disadvantage: the problems of dynamic partitioning**
  - external fragmentation
  - compaction
- **The next step is to try and simplify mechanisms by using *equally sized* memory allocation units.**

Simple Paging (Vs Fixed Partitioning)

- Main memory is partitioned into equal fixed-sized chunks called *frames*
- Each program is also ideally divided into chunks of the same size called *pages*
- The pages of a program can thus be assigned to the available frames in main memory
- Consequences:
  - a program can be scattered all over the physical memory
  - external fragmentation eliminated



|m| => No of bits used to represent m

n+m

P.A. $= k \times 2^{|m|} + m$

6×1024

6144

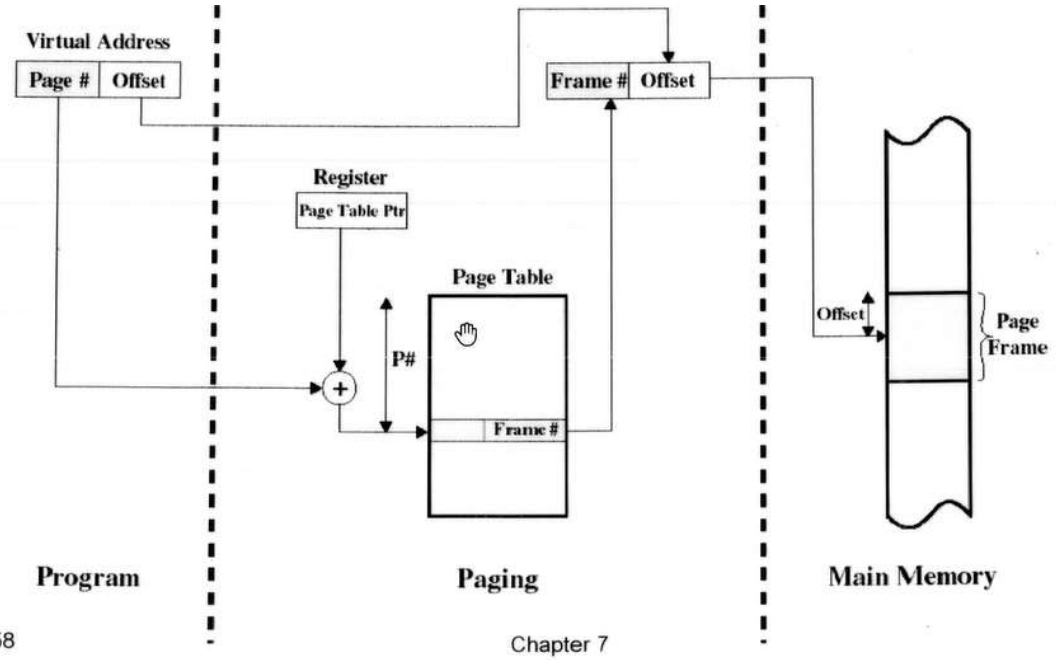
478

6622

(using concatenation)

frame : offset

P.A. $= 6622_{10}$


Virtual Address
Page #
Offset
Register
Page Table Ptr
Page Table
P#
Frame #
Frame #
Offset
Offset
Page Frame
Program
Paging
Main Memory

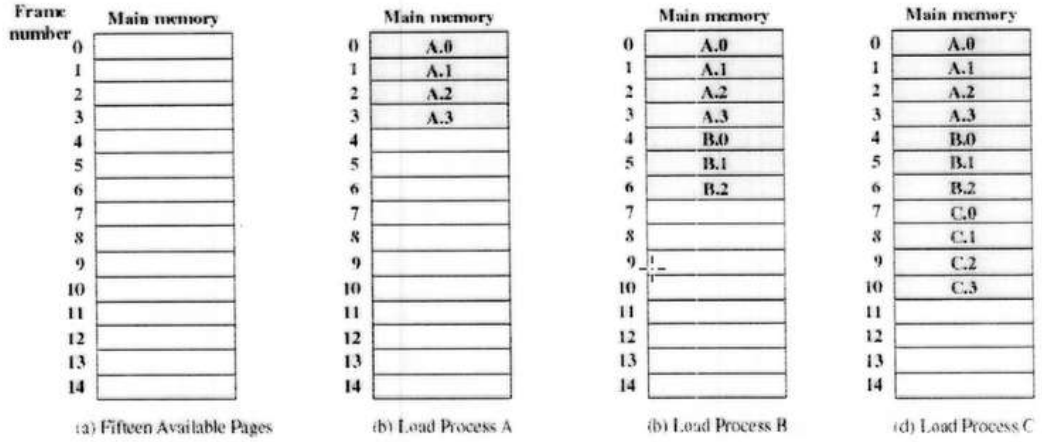
Evaluation of Simple Segmentation

- **Advantage: memory allocation unit is a logically independent portion of program**
  - Segments can be loaded individually on demand (*dynamic linking*).
- **Disadvantage: the problems of dynamic partitioning**
  - external fragmentation
  - compaction
- **The next step is to try and simplify mechanisms by using *equally sized* memory allocation units.**

Simple Paging (Vs Fixed Partitioning)

- **Main memory is partitioned into equal fixed-sized chunks called *frames***
- **Each program is also ideally divided into chunks of the same size called *pages***
- **The pages of a program can thus be assigned to the available frames in main memory**
- **Consequences:**
  - a program can be scattered all over the physical memory
  - external fragmentation eliminated

Example of program loading

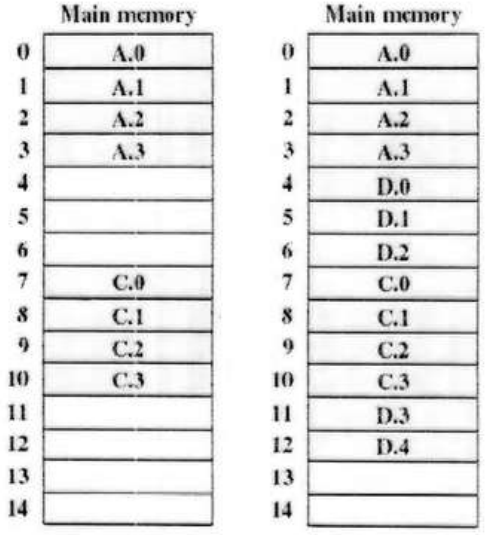


(a) Fifteen Available Pages (b) Load Process A (b) Load Process B (d) Load Process C

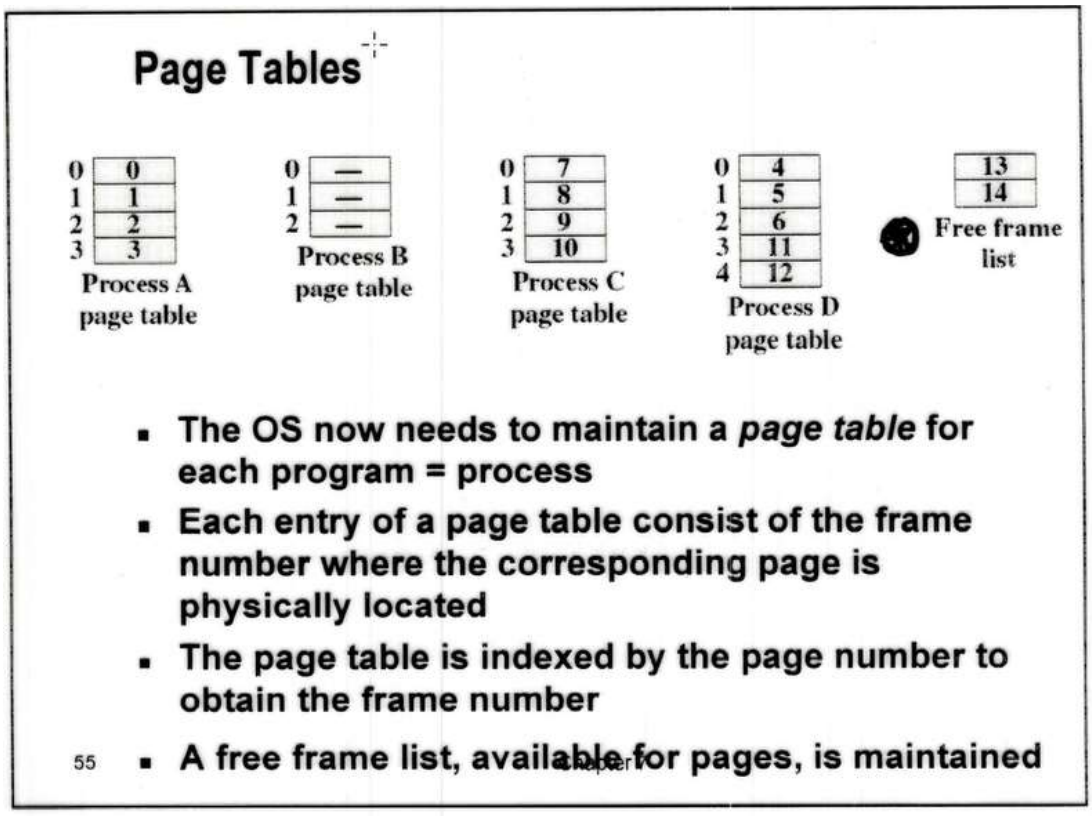
- **Now suppose that program B is suspended or terminates**

Example of process loading (cont.)

- **The OS then loads a new program D consisting of 5 pages**
- **Program D does not occupy a contiguous section of memory**
- **There is no external fragmentation**
- **Internal fragmentation consists only of part of the last page of each program (avg 1/2 page/program).**
- **Programs did not have to be moved.**



(e) Swap out B (f) Load Process D

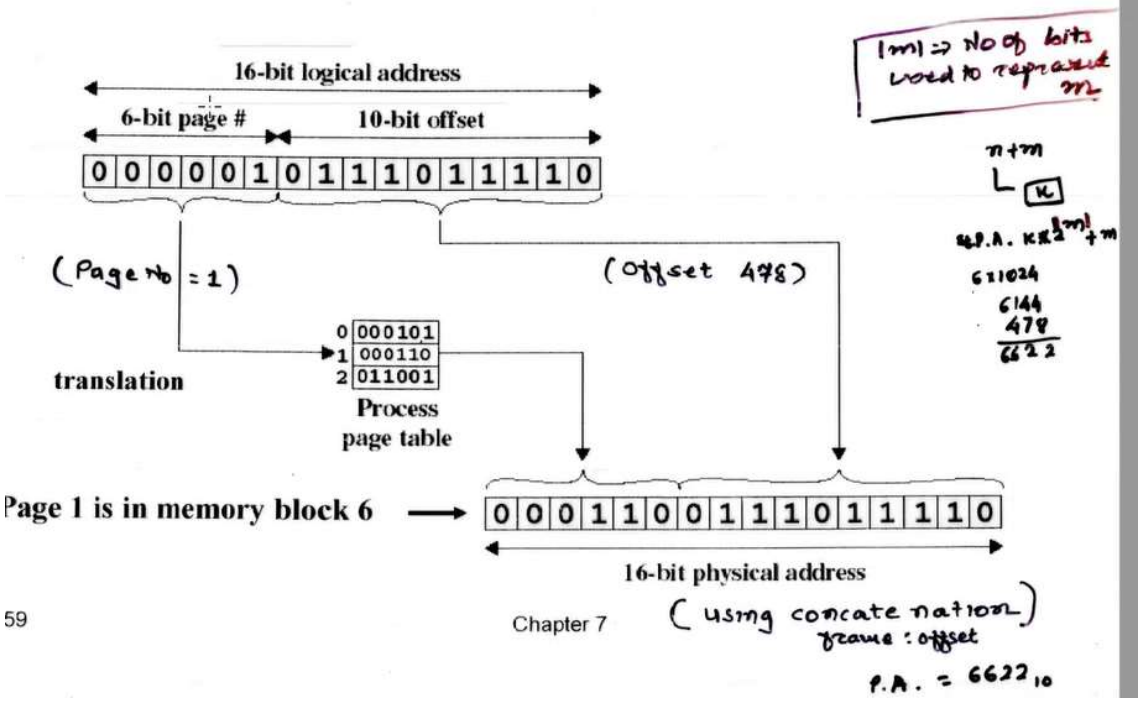

Page Tables
0 0
1 1
2 2
3 3
Process A page table
0 —
1 —
2 —
Process B page table
0 7
1 8
2 9
3 10
Process C page table
0 4
1 5
2 6
3 11
4 12
Process D page table
13
14
Free frame list
The OS now needs to maintain a page table for each program = process
Each entry of a page table consist of the frame number where the corresponding page is physically located
The page table is indexed by the page number to obtain the frame number
A free frame list, available for pages, is maintained
55
Chapter 7

Page fault.

Logical address used in paging

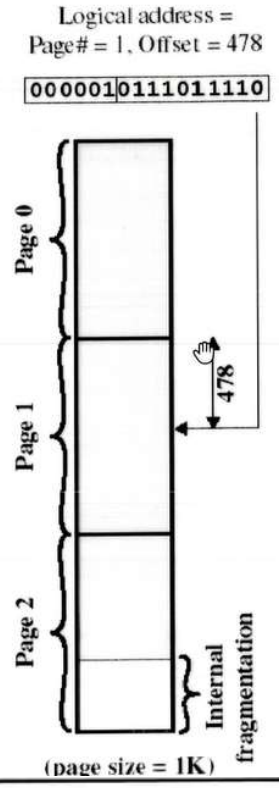
- Within each program, each logical address must consist of a *page number* and an *offset* within the page
- A CPU register always holds the starting physical address of the page table of the program/process which is running on that CPU
- Presented with a logical address (page number, offset) the CPU accesses the page table to obtain the physical address (frame number, offset)

Logical-to-Physical Address Translation in Paging (see also Fig. 8.3)



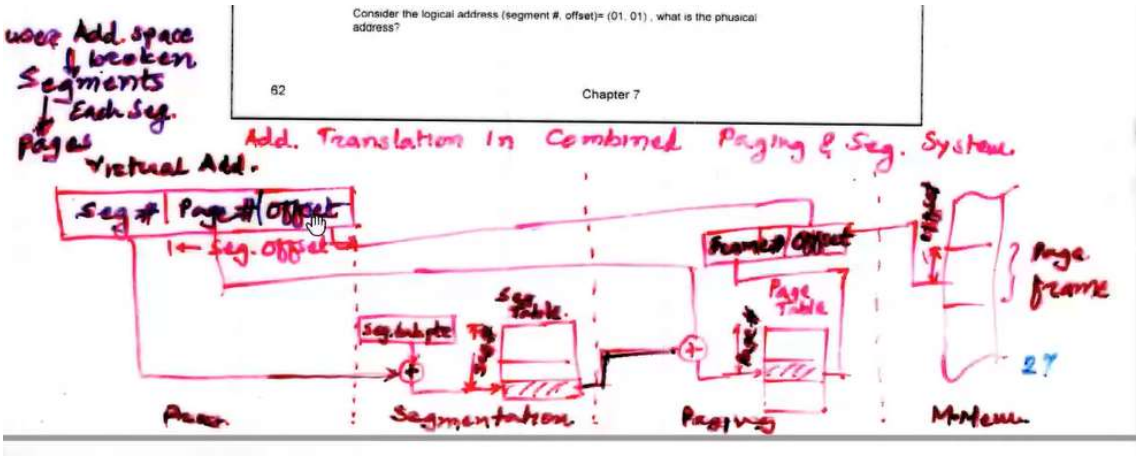
Logical address in paging

- Page size is always chosen to be a power of 2.
- Ex: if 16 bits addresses are used and page size = 1K, we need 10 bits for offset and have 6 bits available for page number * Max $2^6 = 64$ Pages
- Logical address (n,m) gets translated to physical address (k,m) by indexing the page table and appending the page offset m to the frame number k

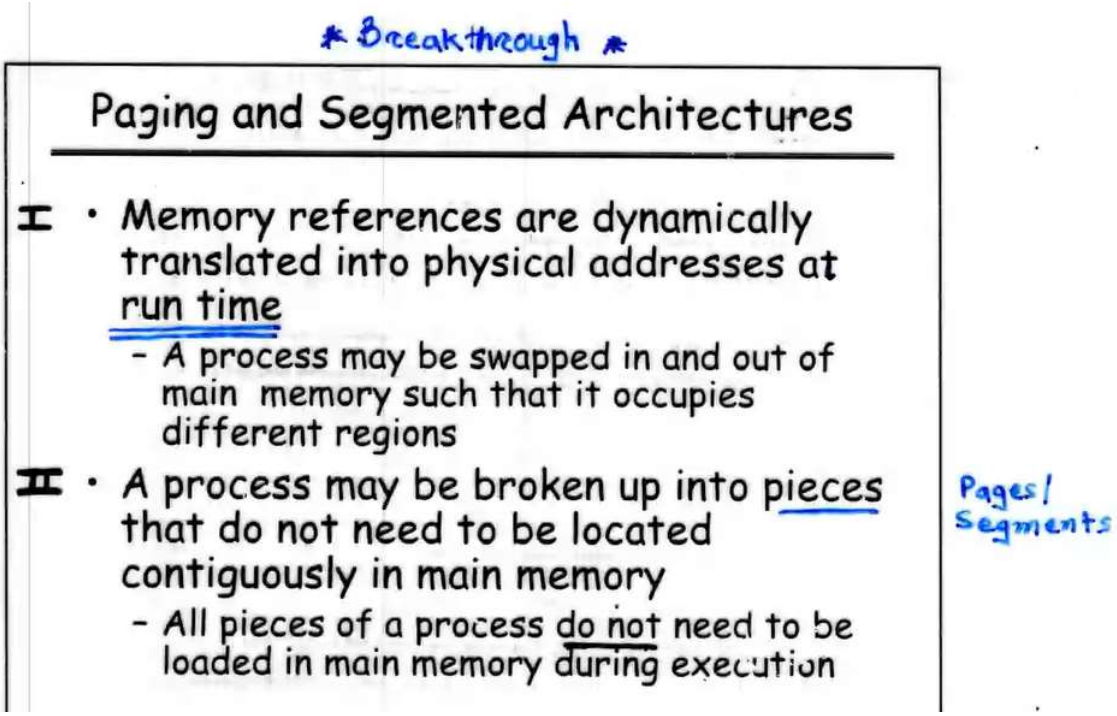


Comparison of simple paging and simple segmentation w. dynamic partitions

- Simple segmentation suffers from external fragmentation (this comes from dyn. partitions)
- Paging only yields internal fragmentation (1/2 page in average per program)
- Address calculation is slightly more complicated with segmentation (addition instead of concatenation)
- Segmentation is visible to the programmer whereas paging is transparent
- Segment protection and sharing is more logical than page protection because normally one wants protection to be based on logical program portions.
- Segmentation allows dynamic linking of segments.
- Happily, segmentation and paging can be combined (next chapter) ...



VIRTUAL MEMEORY



Execution of a Program

- Operating system brings into main memory a few pieces of the program
  - *Resident set* - portion of process that is in main memory
- An interrupt is generated when an address is needed that is not in main memory
  - Operating system places the process in a blocking state

mem. fault/
page fault

Page fault :
I bounds error - outside add. range
II validation " - non-resident page
III protection " - non-permitted access.

Execution of a Program

- Piece of process that contains the logical address is brought into main memory
  - Operating system issues a disk I/O Read request
  - Another process is dispatched to run while the disk I/O takes place
  - An interrupt is issued when disk I/O complete which causes the operating system to place the affected process in the Ready state

Advantages

1. More processes may be maintained in main memory
   - Only load in some of the pieces of each process
   - With so many processes in main memory, it is very likely a process will be in the Ready state at any particular time

2. A process may be larger than all of main memory

→ A programmer is now impression of large storage

more effᵗ utilⁿ of the Processor.

→ O/s & H/w is responsible for effᵗ. utilⁿ of m.mem.

Types of Memory

- ***Real memory***
  - Main memory
- ***Virtual memory***
  - Memory on disk
  - Allows for effective multiprogramming and relieves the user of tight constraints of main memory

Principle of Locality

- Program and data references within a process tend to cluster
- Only a few pieces of a process will be needed over a short period of time
- Possible to make intelligent guesses about which pieces will be needed in the future
- This suggests that virtual memory may work efficiently

Thrashing

- Swapping out a piece of a process just before that piece is needed
- The processor spends most of its time swapping pieces rather than executing user instructions

Principle of Locality

- Program and data references within a process tend to cluster
- Only a few pieces of a process will be needed over a short period of time
- Possible to make intelligent guesses about which pieces will be needed in the future
- This suggests that virtual memory may work efficiently

* Over a long period of time, the cluster in use change, but
* Over a short period of time, the processor is primarily working with fixed clusters of mem. references.

Virtual Memory Support

I • Hardware must support paging and/or segmentation

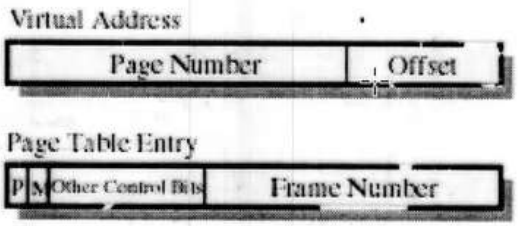
II • Operating system must be able to manage the movement of pages and/or segments between secondary memory and main memory

I. MMU H/w

Paging

- Virtual and physical memory divided into fixed size pages
- Translations maps virtual page to a physical page.
- Pages marked as resident or non-resident *
- non-resident pages cause page faults
- Policies: Fetch, placement, replacement
  - we focus on prepaging systems

Page Table Entries

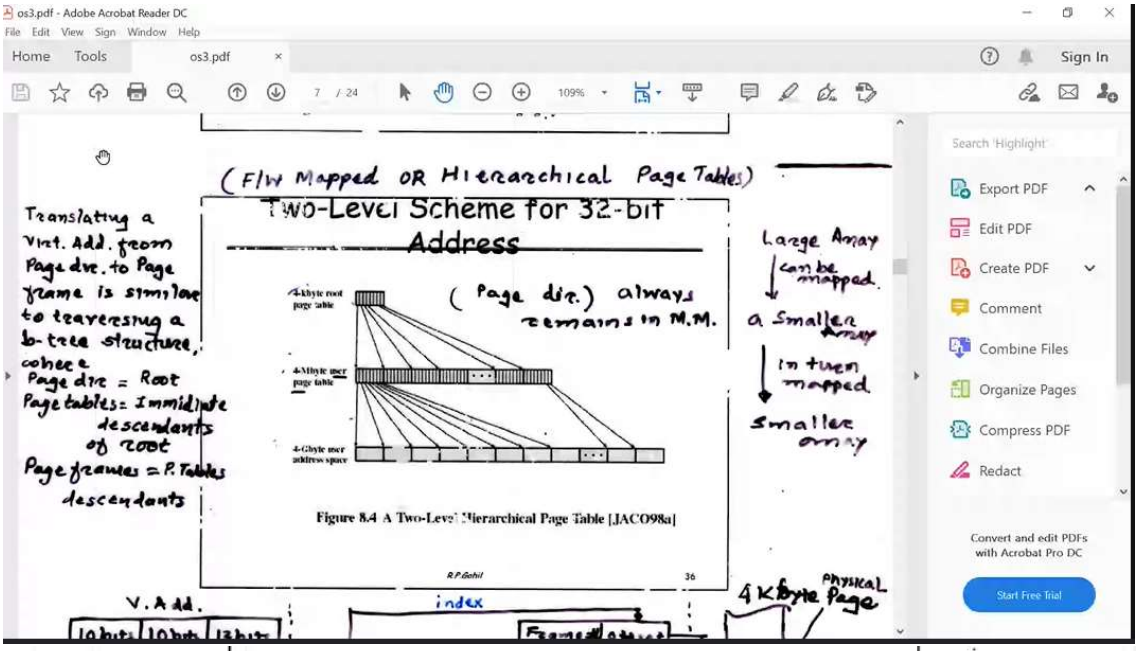


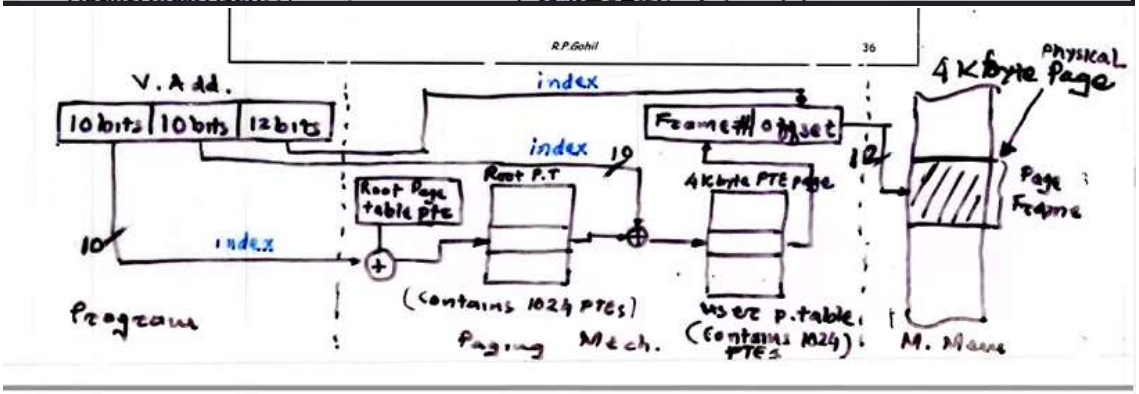
(a) Paging only

Figure 8.2 Typical Memory Management Formats

Resident and Modify Bit in PT

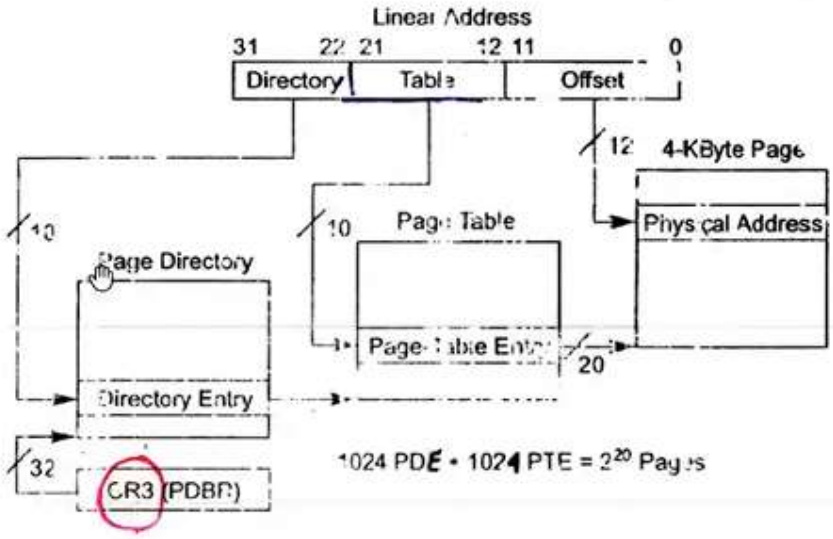
- Resident bit indicates if page is in memory (as few pieces are brought in)
- Another modify bit is needed to indicate if the page has been altered since it was last loaded into main memory
- If no change has been made, the page does not have to be written to the disk when it needs to be swapped out



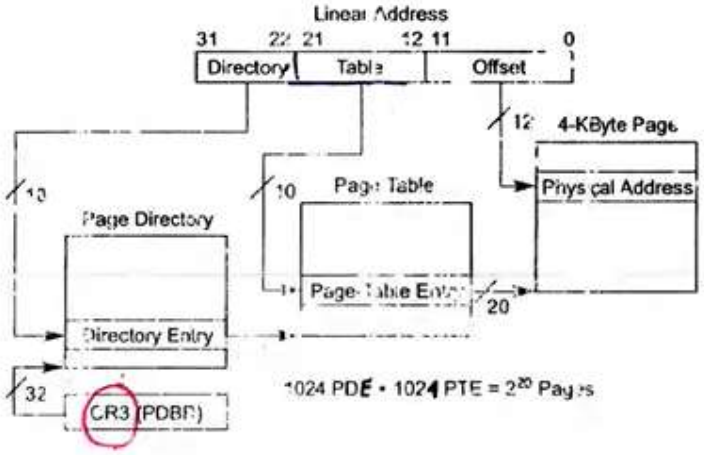


» Intel Pentium

- in addition to segmentation, Pentium **has** the option of paging
- the 32 bit *linear address* produced from segmentation translation is further translated to a physical address using page tables
- various different *modes* of operation available :
  - 32 bit physical addresses with 4Kb or 4Mb pages
  - 36 bit physical addresses with 4Kb or 2Mb pages (Pentium Pro onwards)
- for 32 bit physical addresses & 4Kb pages – *two level* page tables :

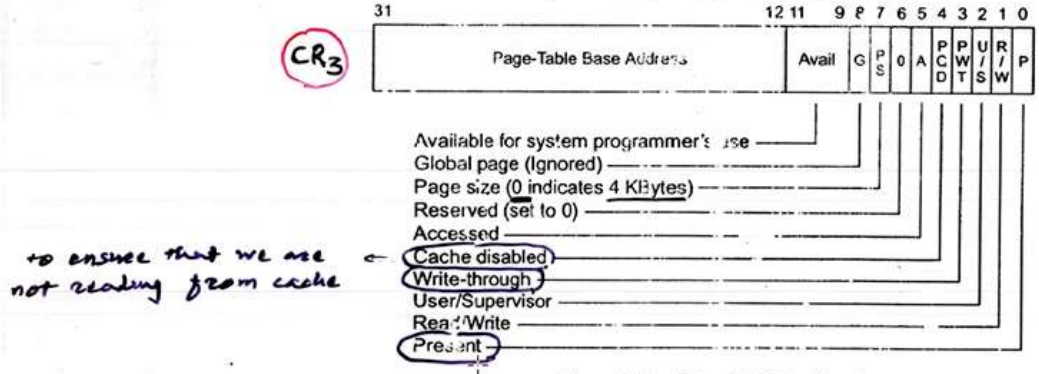


System Control Reg. of the Pentium Proc.



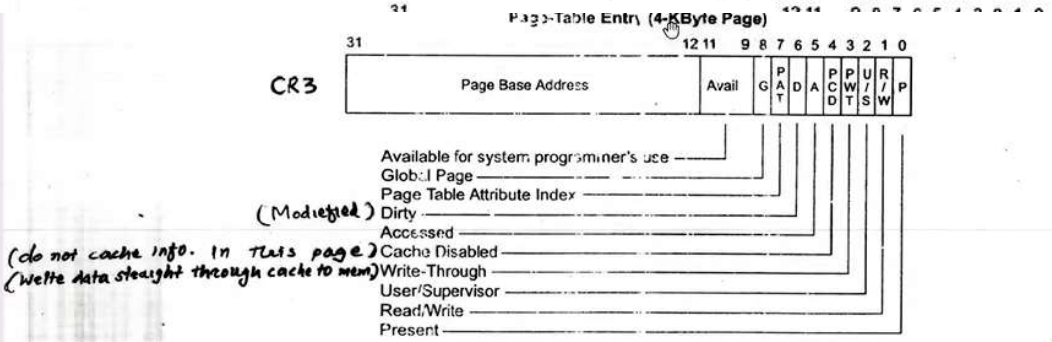
System Control Reg. of the Pentium Proc.

Page-Directory Entry (4-KByte Page Table)



to ensure that we are not reading from cache

Page-Table Entry (4-KByte Page)

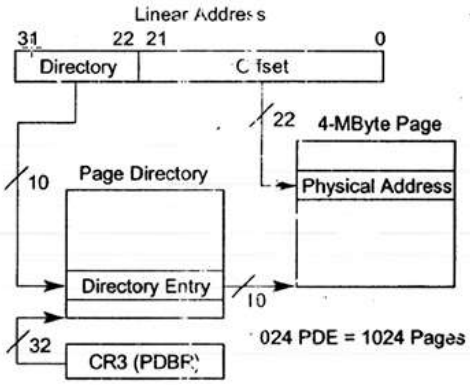


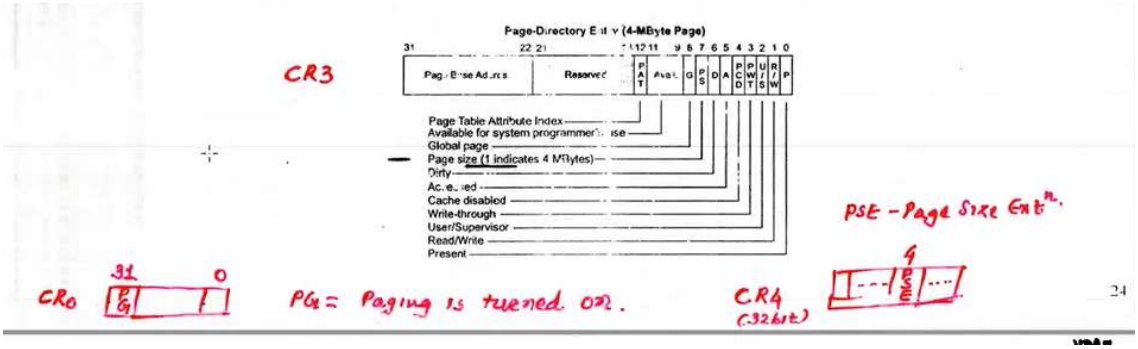
(Modified)

(do not cache info. in this page)
(write data straight through cache to mem)

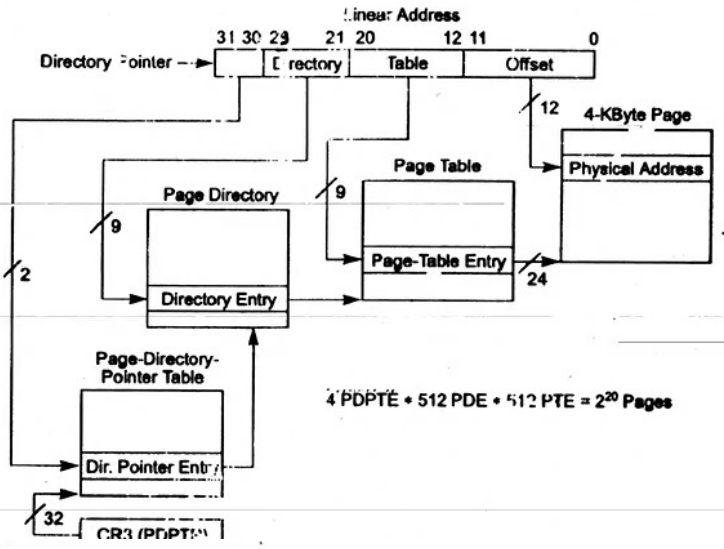
CR2: (32 bit) During page fault, the add., for which the page fault occured is saved in CR2

– for 32 bit physical addresses & 4Mb pages – *one level* page table :

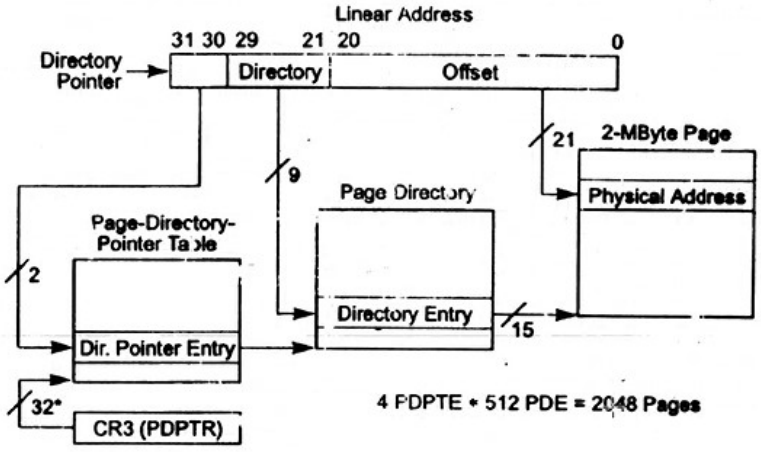




– for 36 bit physical addresses & 4Kb pages – *three level* page tables :



– 36 physical addresses and 2Mb pages – *two-level* page tables :



Where to put the page tables ?

Sol¹ in CPU Regs.
adv. - speed
dis.:- limited No. of pages per process, v.Mem. limited (regs. costly)

Sol²- in Main Mem.
adv. pot. very large size of v.M.
dis.:- see next

Sol³. Mixed
Page tables in M.M. but most used regs. are in CPU regs.

Try:- ps
pmap 478 PID OR VM#

Page Tables

Page Tables

- The entire page table may take up too much main memory
- Page tables are also stored in virtual memory
- When a process is running, part of its page table is in main memory

* Virtual Mem reference cause two phy. mem. access : one to fetch the appr. PTE & one to fetch the data. So doubling the mem. access time.

Soln :-

* Translation Lookaside Buffer (TLB)

- Contains page table entries that have been most recently used
- Functions same way as a memory cache

* a sp. high-speed cache for PTEs
* Stores a set of translations from virtual page number to page-frame number
* TLB translations much faster than going via page table in memory

R.P.Gohil 38

* effective access time = $(h*t+(1-h)*(t+m))+m$

* INTEL 486 : 32 entries (claimed 98% hit ratio)
  " Pentium : 128 entry Inst^n
  64 entry data (4 way associative)

m = m.mem access time
t = TLB acc. time
h = hit ratio

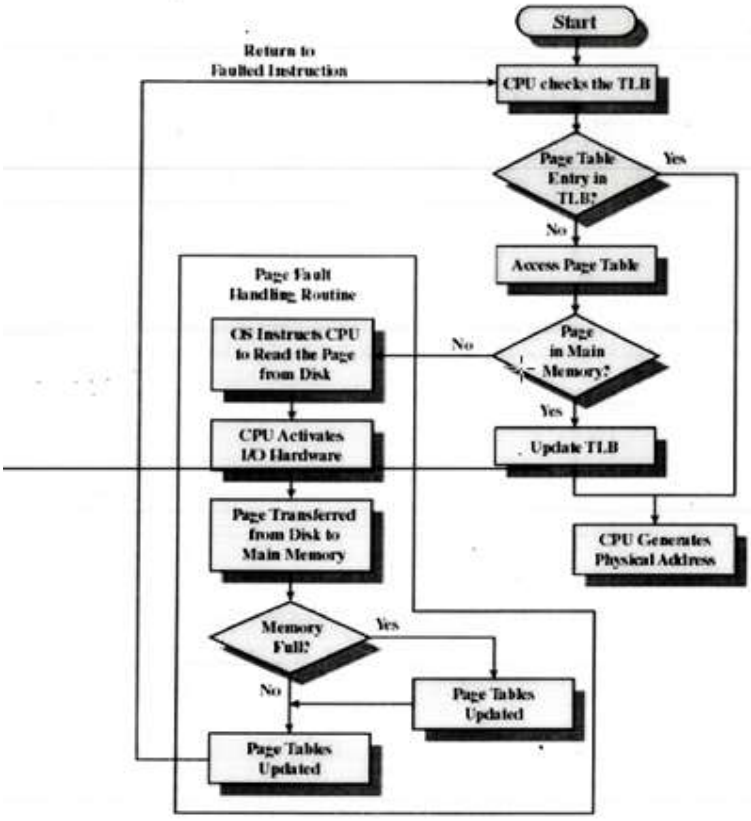
Translation Lookaside Buffer

- Given a virtual address, processor examines the TLB
- If page table entry is present (a hit), the frame number is retrieved and the real address is formed
- If page table entry is not found in the TLB (a miss), the page number is used to index the process page table

Mapping

1. Associative
2. Direct
3. Set. Assoc.....

R.P.Gohil 39



Intel Itanium (IA-64 bit Arch.): TLB for ← Data / Inst.

2 Levels of Data TLB : L1 (32 entries) } Fully
L2 (96 entries) } Associative

Inst^n TLB : 64 entries — Fully Assoc.

Observe the Complexity of the CPU HW in single mem req.

i.e. Transl^n Virt. Add. → Real Add. — Involves → Reference to PTE (which may be in

disk/M.Mem/cache ← & Referenced word may be in — disk/M.Mem/TLB

... requires updation

....

os3.pdf - Adobe Acrobat Reader DC

File Edit View Sign Window Help

Home Tools os3.pdf os4.pdf Sign In

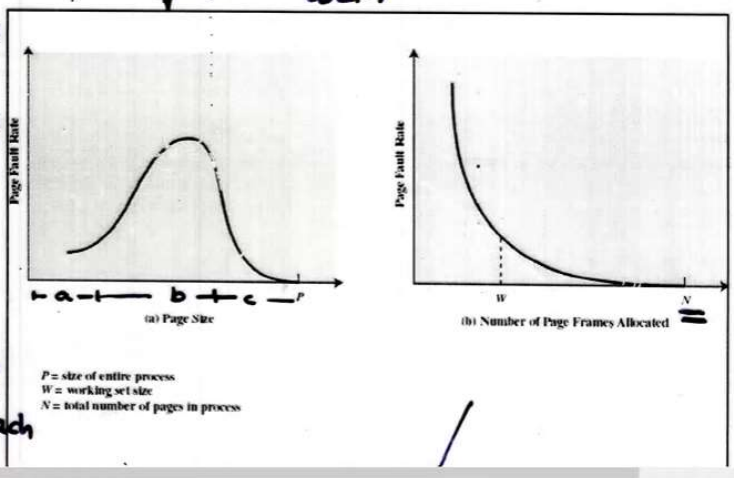
18 / 24 150%

(based on Pri. Loc.)

(a) Small Page Size
Many Pages in Mem – Low P.F (Prin. Locality)

(b) Increased Page Size causes each prog. to contain more code that is not used
↳ P.F rise (Weak Prin. Loc.)

(c) P.F decrease if we can approach



8.26 x 11.68 in

- Most operating system support only one page size

R.P.Gohil 46

Small pages are grouped together by OSes to make larger effective pages

e.g. Sun SPARC 4 Kb pages,
Intel x86 : 4 Kb Pages (with 4Mb option)

8

Monitoring V. Mem. with vmstat (Linux) ← VM13 (9)

```
vmstat 5 10
       delay  How many updates.

r  swpd  free  buff  cache   si  so  bi  bo  in  cs  us  sy  id
                    Memory      Swap      io      System      CPU
↓
free mem.
```

page-ins / page outs ↓

If 0 ⇒ free of paging activity

8.26 x 11.68 in

2-Level H.Level page table contains entries (pages) pointing to the page tables and each page table contains entries of different pages of a process.

32-bit : The root directory is kept in one page (4Kbyte) having each entry 4-byte (2^2) in length, and hence consists of 2^{10} entries, and each such entries points to one user page table which again consists of 2^{10} page-table entries. In this way, the total 2^{20} virtual pages are mapped by the root page table with 2^{10} entries.

* TLB → mapping cache

different

memory cache.

See (1) My Computer → Properties → Advanced → Performance → Settings → Advanced → Virtual Mem. VM14

(2) Control Panel → Admin Tools → Performance → Syst. Monitor

Rt. clk on Graph → Add Counters → Performance Object

Example Page Sizes

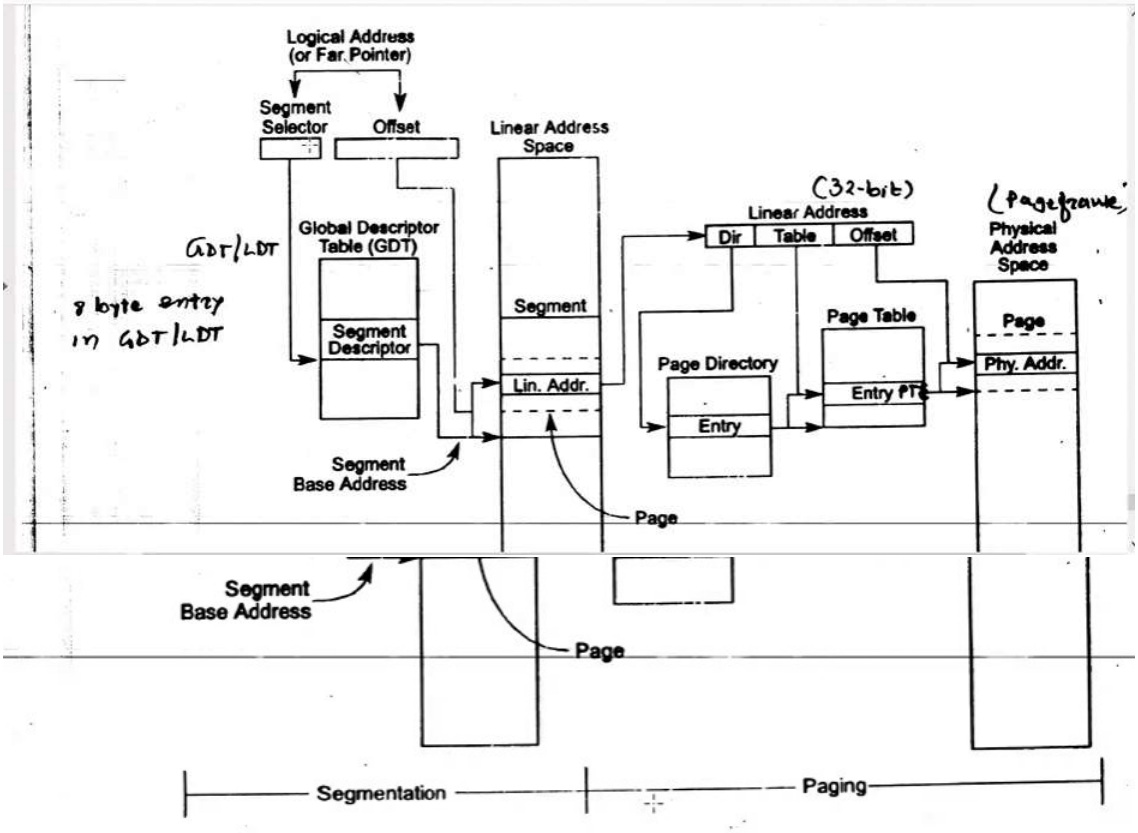
Table 8.2 Example Page Sizes

Computer	Page Size
Atlas	512 48-bit words
Honeywell-Multics	1024 36-bit words
IBM 370/XA and 370/ESA	4 Kbytes
VAX family	512 bytes
IBM AS/400	512 bytes
DEC Alpha	8 Kbytes
MIPS R400	4 kbytes to 16 Mbytes
UltraSPARC	8 Kbytes to 4 Mbytes
Pentium	4 Kbytes or 4 Mbytes
PowerPC	4 Kbytes

Itanium C64 bit Arch. 8 kB

R.P.Gohil

47



Intel Memory Management

The memory management facilities of the **IA-32** architecture are divided into two parts:

Segmentation

Segmentation provides a mechanism of isolating individual code, data, and stack modules so that multiple programs (or tasks) can run on the same processor without interfering with one another.

When operating in protected mode, some form of segmentation must be used.

Paging.

Paging provides a mechanism for implementing a conventional demand-paged, virtual-memory system where sections of a program's execution environment are mapped into physical memory as needed. Paging can also be used to provide isolation between multiple tasks.

These two mechanisms (segmentation and paging) can be configured to support simple single program (or single-task) systems, multitasking systems, or multiple-processor systems that used shared memory.

Intel Memory Management

See Figure 3-1.

Segmentation gives a mechanism for dividing the processor's addressable memory space (called the **linear address space**) into smaller protected address spaces called **segments**.

Segments are used to hold code, data, and stack for a program andr to hold system data structures (such as a TSS or LDT).

Each program running on a processor, is assigned its own set of segments.

The processor enforces the boundaries between segments and insures that one program doesn't interfere with the execution of another .

The segmentation mechanism allows typing of segments to restrict operations that can be performed.

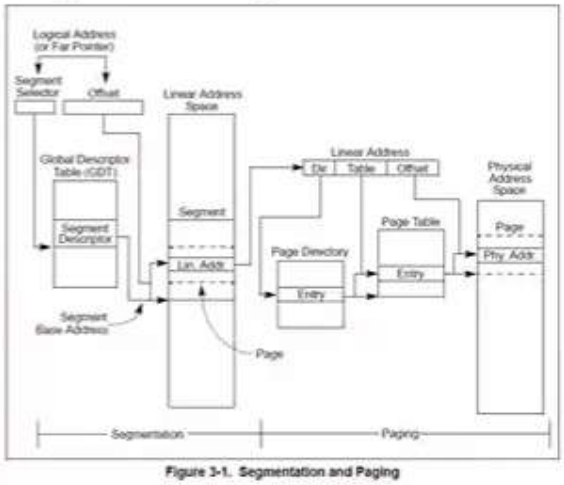


Figure 3-1. Segmentation and Paging

Intel Memory Management

See Figure 3-1.

All the segments in a system are contained in the processor's **linear address space**.

To locate a byte in a particular segment, **a logical address** (also called a far pointer) must be provided.

A logical address has :

1. **The segment selector – a** unique identifier for a segment - provides an offset into a descriptor table (such as the global descriptor table, GDT) to a data structure called a segment descriptor.

   This segment descriptor specifies the size of the segment, the access rights and privilege level for the segment, the segment type, and the location of the first byte of the segment in the linear address space (called the base address of the segment).

   See 3.4.2 Segment Selectors" for more details.

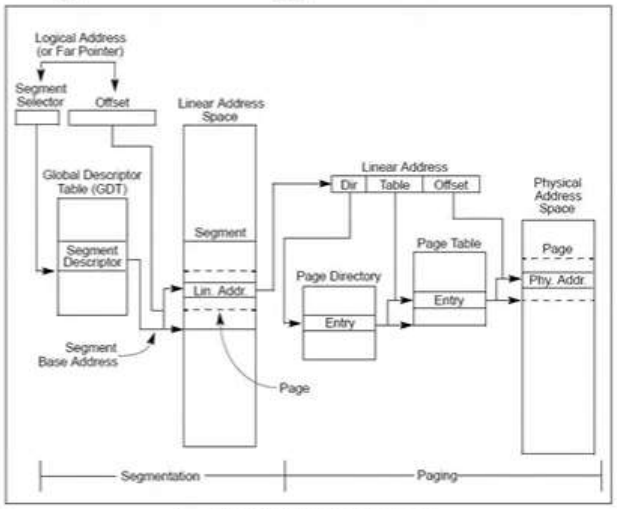


Figure 3-1. Segmentation and Paging

2. **The offset** part of the logical address -added to the base address for the segment to locate a byte within the segment. The base address plus the offset thus forms a **linear address** in the processor's linear address space.

Intel Memory Management

3.2.1 Basic Flat Model

The simplest memory model for a system is the basic "flat model,"

the operating system and application programs have access to a continuous, unsegmented address space.

.

To implement a basic flat memory model with the IA-32 architecture, at least two segment descriptors must be created:

- one for referencing a code segment and
- one for referencing a data segment (see Figure 3-2).
- both segments, however, are mapped to the entire linear address space: that is, both segment descriptors have the same base address value of 0 and the same segment limit of 4 GBytes.

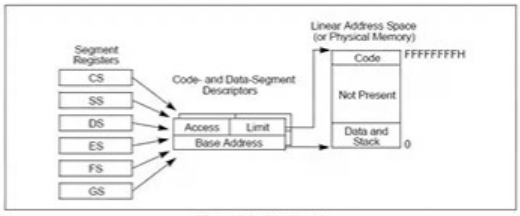


Figure 3-2. Flat Model

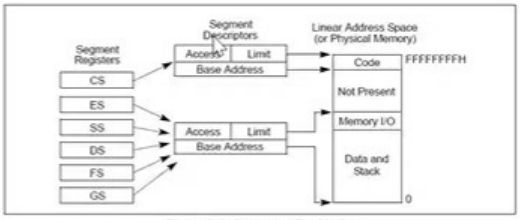


Figure 3-3. Protected Flat Model

Intel Memory Management

3.4 LOGICAL AND LINEAR ADDRESSES

The processor uses two stages of address translation to arrive at a physical address: logical-address (via segments) translation and linear address space (via paging) translation.

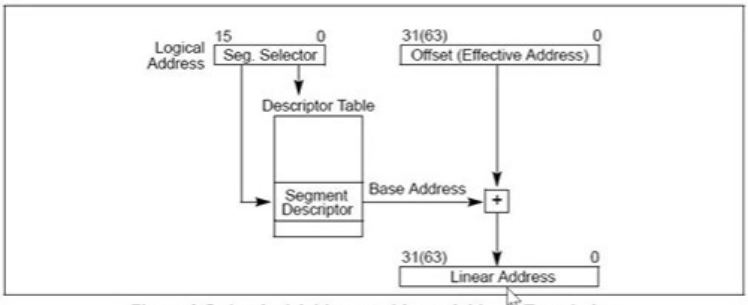


Figure 3-5. Logical Address to Linear Address Translation

Intel Memory Management

3.4 LOGICAL AND LINEAR ADDRESSES

Every byte in the processor's address space is accessed with a logical address. A **logical** address consists of a 16-bit segment selector and a 32-bit offset (see Figure 3-5).

.A **linear** address is a 32-bit address in the processor's linear address space. The linear address space is a flat (unsegmented), 2^{32}-byte address space, with addresses ranging from 0 to FFFF,FFFFH.

The linear address space contains all the segments and system tables defined for a system.

To translate a logical address into a linear address, the processor does the following:

1. Uses the offset in the segment selector to find the descriptor for the segment in the GDT or LDT and reads it into the processor, or uses the appropriate segment register.
2. Examines the segment descriptor to check the access rights and range of the segment – makes sure the segment is accessible and has legal offset.
3. Adds the base address of the segment to the offset to form a linear address.

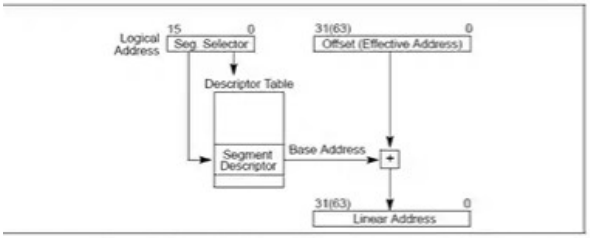


Figure 3-5. Logical Address to Linear Address Translation

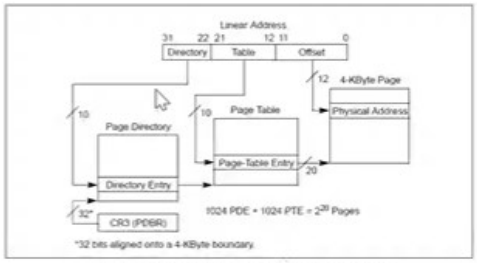


Figure 3-12. Linear Address Translation (4-KByte Pages)

Fetch Policy

- Fetch Policy
  - Determines when a page should be brought into memory
  - Demand paging only brings pages into main memory when a reference is made to a location on the page.
    - Many page faults when process first started
  - Prepaging brings in more pages than needed
    - More efficient to bring in pages that reside contiguously on the disk

I
Pro: Prog. may not be always using all its pages
cons: slows down the processing
II : clustering as another method
takes adv. that pages are written seq'ly (e.g. error module needed only when error)
Pro II Reduces Repeated disk access time
Pro Reduces Repeated disk access time
cons: fails if the use extra pages found to be of no use.
II : Another Method. Anticipatory fetching

Replacement Policy

- Placement Policy
  - Which page is replaced?
  - Page removed should be the page least likely to be referenced in the near future
  - Most policies predict the future behavior on the basis of past behavior

Replacement Policy

- Frame Locking
  - If frame is locked, it may not be replaced
  - Kernel of the operating system
  - Control structures
  - I/O buffers — time-critical access
  - Associate a lock bit with each frame

Basic Replacement Algorithms

- Optimal policy
  - Selects for replacement that page for which the time to the next reference is the longest
  - Impossible to have perfect knowledge of future events
  - Serves as a standard to compare with the other Algo.

Basic Replacement Algorithms

- Least Recently Used (LRU)
  - Replaces the page that has not been referenced for the longest time
  - By the principle of locality, this should be the page least likely to be referenced in the near future
  - Each page could be tagged with the time of last reference. This would require a great deal of overhead.

Basic Replacement Algorithms

- First-in, first-out (FIFO)
  - Treats page frames allocated to a process as a circular buffer
  - Pages are removed in round-robin style
  - Simplest replacement policy to implement
  - Problem → Page that has been in memory the longest is replaced (frequently used page is often the oldest)
  - These pages may be needed again very soon

Basic Replacement Algorithms

FIFO with 2nd chance

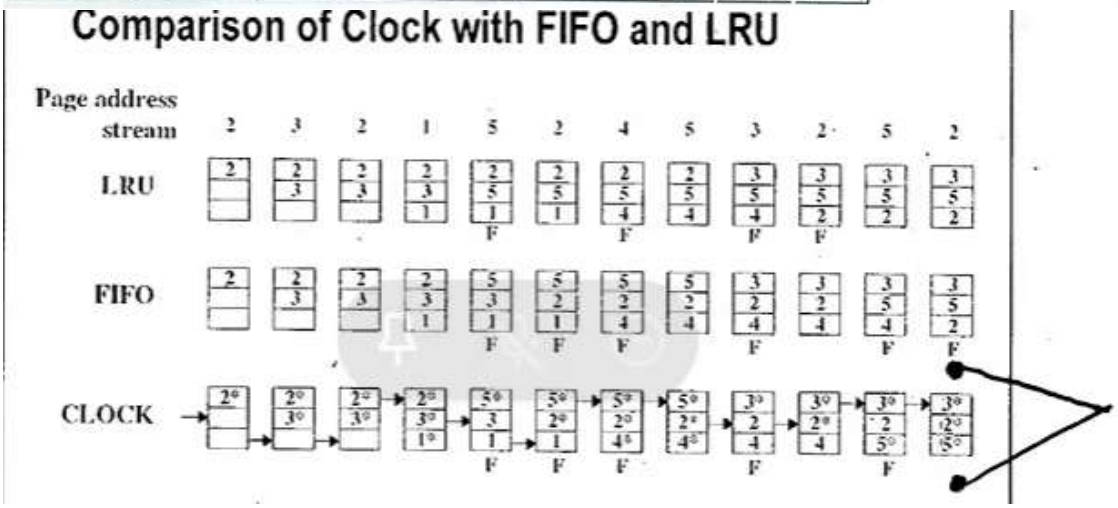
- Clock Policy (give 'a chance' to recently used pages)
  - Additional bit called a use bit
  - When a page is first loaded in memory, the use bit is set to 1
  - When the page is referenced, the use bit is set to 1 (subsequently) *
  - When it is time to replace a page, the first frame encountered with the use bit set to 0 is replaced.
  - During the search for replacement, each use bit set to 1 is changed to 0

* the reference that generated pagefault

Adv. - Fast and does not replace a heavily used page.

- Influenced by Locality principles

Comparison of Clock with FIFO and LRU



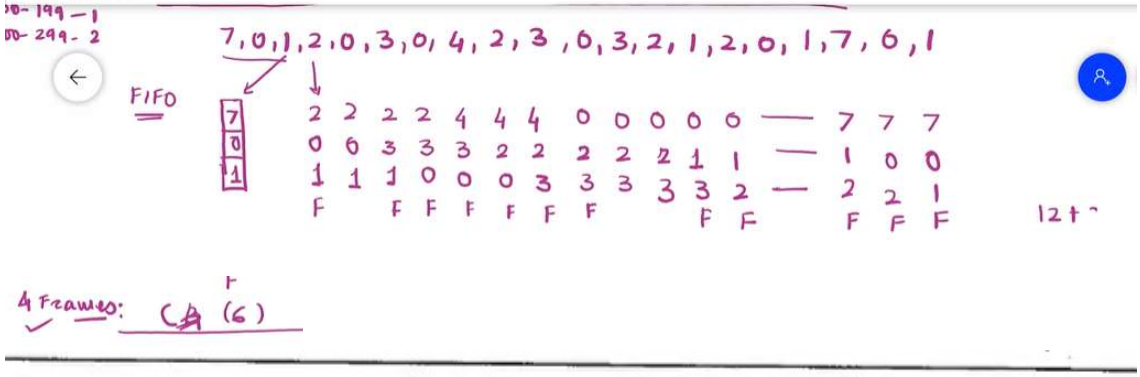
7012030423032120170 1

USING 3 FRAMES AND 4 FRAMES

-LRU,FIFO

Microsoft Whiteboard

100-199 – 1
200-299 - 2



4 Frames: (6)

Comparison of Clock with FIFO and LRU

- **Numerical experiments tend to show that performance of Clock is close to that of LRU**
- **Experiments have been performed when the number of frames allocated to each process is fixed and when pages local to the page-fault process are considered for replacement**
  - When few (6 to 8) frames are allocated per process, there is almost a factor of 2 of page faults between LRU and FIFO
  - This factor reduces close to 1 when several (more than 12) frames are allocated. (But then more main memory is needed to support the same level of multiprogramming)
- **Note however that modern computers have thousands of frames so in practice the difference between these algorithms may not be very important.**

Enhanced Second Chance

- It is cneaper to replace a page that has not been written
  - OS need not write the page back to disk

  ⇒ OS can give preference to paging out un-modified pages

- Hardware keeps a *modify* bit (in addition to the reference bit)

'1': page is modified (different from the copy on disk)
'0': page is the same as the copy on disk

Enhanced Second Chance

- The reference bit and modify bit form a pair (r,m) where

1. (0,0) neither recently used nor modified - replace this page!
2. (0,1) not recently used but modified - not as good to replace, since the OS must write out this page, but it might not be needed anymore.
3. (1,0) recently used and unmodified - probably will be used again soon, but OS need not write it out before replacing it
4. (1,1) recently used and modified - probably will be used again soon and the OS must write it out before replacing it

- On a page fault, the OS searches for the first page in the lowest

Nonempty class

Page Replacement in Enhanced Second Chance

- The OS goes around at most three times searching for the (0,0) class.
  1. Page with (0,0) ⇒replace the page.
  2. Page with (0,1) ⇒initiate an I/O to write out the page, locks the page in memory until the I/O completes, clears the modified bit, and continue the search
  3. For pages with the reference bit set, the reference bit is cleared.
  4. If the hand goes completely around once, there was no (0,0) page.
     - On the second pass, a page that was originally (0,1) or (1,0) might have been changed to (0,0) ⇒replace this page
     - If the page is being written out, waits for the I/O to complete and then remove the page.
     - A (0,1) page is treated as on the first pass.
     - By the third pass, all the pages will be at (0,0).

- This factor reduces close to 1 when several (more than 12) frames are allocated. (But then more main memory is needed to support the same level of multiprogramming)
- Note however that modern computers have thousands of frames so in practice the difference between these algorithms may not be very important.

73 (NRU) R.P.GOHIL

m ⇒ Set-0 after first load
- 1 - after a write
used in Mac.

Enhanced FIFO with 2nd chance (Adv. over clock – Pages that are unchanged are given preference for replacement.) Cand. for Repl.

Clock Improvement: Not accessed recently, not modified (u=0 ; m=0) ✓

use of modi. bit (u m)
Step 1 : look for 0,0 (Don't change 'u' in traverse) " " " " u=1 ; m=0 (used but clean, but will be used again)
S-2 : If 1. fails 0,1 (first page with 0,1) (change u to 0) " " " " u=0 ; m=1 (Need to be written out before ...)
S-3 : If 2. fails (U R at orl. posn) " " " " u=1 ; m=1 (used again and need to be written out to disk bef. repl.)
Now all with u=0, Repeat S-1 and S-2 if necessary

01 is preferred over 10. Because a page that has been modi. must be written out before being replaced, there is an immediate saving of time

OBSERVATIONS *
* LRU & CLOCK superior to FIFO but involves complexity & overhead.
* cost of replacing a modi. page is greater.

Soln:

Page Buffering (improve paging perf. & use of simple p. repl. alg.)

- Pages to be replaced are kept in main memory for a while to guard against poorly performing replacement algorithms such as FIFO
- Two lists of pointers are maintained: each entry points to a frame selected for replacement
  - a free page list for frames that have not been modified since brought in (no need to swap out)
  - a modified page list for frames that have been modified (need to write them out)

- A frame to be replaced has a pointer added to the tail of one of the lists and the *present* bit is cleared in the corresponding page table entry
  - but the page remains in the same memory frame

75

Page Buffering

- At each page fault the two lists are first examined to see if the needed page is still in main memory
  - If it is, we just need to set the present bit in the corresponding page table entry (and remove the matching entry in the relevant page list)
  - If it is not, then the needed page is brought in, it is placed in the frame pointed by the head of the free frame list (overwriting the page that was there)
    - the head of the free frame list is moved to the next entry
- The modified list also serves to write out modified pages in clusters (rather than individually)

(reduces number of I/O disk access time)

Plus:- Process is put back to ready queue faster.

Minus:- Less pages are in use overall.

• VAX/VMS version – basic FIFO replacement with a free frame pool.

Cleaning Policy

- **When should a modified page be written out to disk?**
- **Demand cleaning**
  - a page is written out only when its frame has been selected for replacement
    - but a process that suffers a page fault may have to wait for 2 page transfers
- **Precleaning**
  - modified pages are written out before their frame are needed so that they can be written out in batches
    - but makes little sense to write out so many pages if the majority of them will be modified again before they are replaced

Cleaning Policy

- **A good compromise can be achieved with page buffering**
  - recall that pages chosen for replacement are maintained either on a free (unmodified) list or on a modified list
  - pages on the modified list can be periodically written out in batches and moved to the free list
  - a good compromise since:
    - not all dirty pages are written out but only those chosen for replacement
    - writing is done in batch

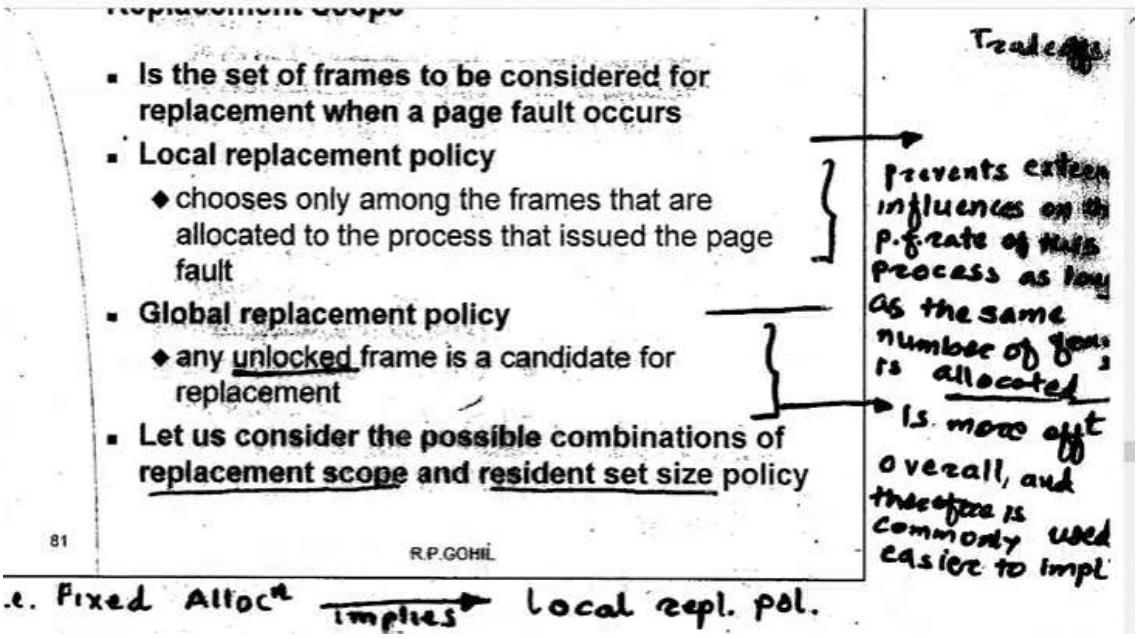
Resident Set Size

- **The OS must decide how many page frames to allocate to a process**
  - large page fault rate if to few frames are allocated to a process
  - low multiprogramming level if too many frames are allocated to a process

Resident Set Size

- **Fixed-allocation policy**
  - allocates a fixed number of frames that remains constant over time
    - the number is determined at load time and depends on the type of the application
- **Variable-allocation policy**
  - the number of frames allocated to a process may vary over time
    - may increase if page fault rate is high
    - may decrease if page fault rate is very low
  - requires more OS overhead to assess behavior of active processes

REEPLACEMENT SCOPE:



Fixed allocation + Local scope

- **Each process is allocated a fixed number of pages**
  - determined at load time and depends on application type
- **When a page fault occurs: page frames considered for replacement are local to the page-fault process**
  - the number of frames allocated is thus constant
  - previous replacement algorithms can be used
- **Problem: difficult to determine ahead of time a good number for the allocated frames**
  - if too low: page fault rate will be high, causing multiprog ing syst. to run slowly.
  - if too large: multiprogramming level will be too low

alloc. concept is also valid

Fixed allocation + Global scope

- **Impossible to achieve**
  - if all unlocked frames are candidate for replacement, the number of frames allocate to a process will necessary vary over time

Variable allocation + Global scope

- Simple to implement--adopted by many OS (like Unix SVR4)
- A list of free frames is maintained
  - When a process issues a page fault, a free frame (from this list) is allocated to it
  - Hence the number of frames allocated to a page fault process increases, resi. set grows (i.e. vari.)
  - The choice for the process that will lose a frame is arbitrary: far from optimal ← limit
- Page buffering can alleviate this problem since a page may be reclaimed if it is referenced again soon

84 R.P.GOHIL

when no free frames available. The sel^n is made from all unlocked frames using any of the prev. policy.

Thus, there is no discipline to determine which process should lose a page from its resident set.

Therefore, the process that suffers the reduction in resi. set size may not be optimum.

Overcome the prob. with above scheme:-

Variable allocation + Local scope (Base for WSS)

- May be the best combination (used by Windows NT)
- Allocate at load time a certain number of frames to a new process based on application type
  - use either prepaging or demand paging to fill up the allocation
- When a page fault occurs, select the page to replace from the resident set of the process that suffers the fault
- Reevaluate periodically the allocation provided and increase or decrease it to improve overall performance
- based on assesment of future demand.

- based on assesment of future demand

Key ele. of this method : - criteria to determine res.size
- timing of changes

But, watch for... Thrashing

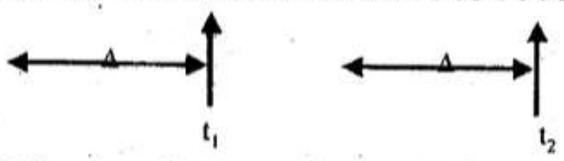
- Attempt to have a lot of processes simultaneously executing in main memory (increasing the level of multiprogramming) may result in *thrashing*.
- This is the phenomenon that occurs when the machine is so busy with paging I/O that can find little time for actually executing user processes.
- The *working set* strategy was invented to prevent thrashing.

The Working Set Strategy

- Is a variable-allocation method with local scope based on the assumption of locality of references
- The working set for a process at time t, $W(\Delta,t)$, is the set of pages that have been referenced in the last Δ virtual time units
  - virtual time = number of page references executed / elapsed time for process in exe
  - Δ is a window of 'time'
  - at any t, $|W(\Delta,t)|$ is non decreasing with Δ
  - $W(\Delta,t)$ is an approximation of the process *locality*

Working set

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ... page trace



$W(10,t_1) = \{1, 2, 5, 6, 7\}$ $W(10,t_2) = \{3, 4\}$

$\Delta = 10$

(WINDOW SIZE)

Variations in WS size

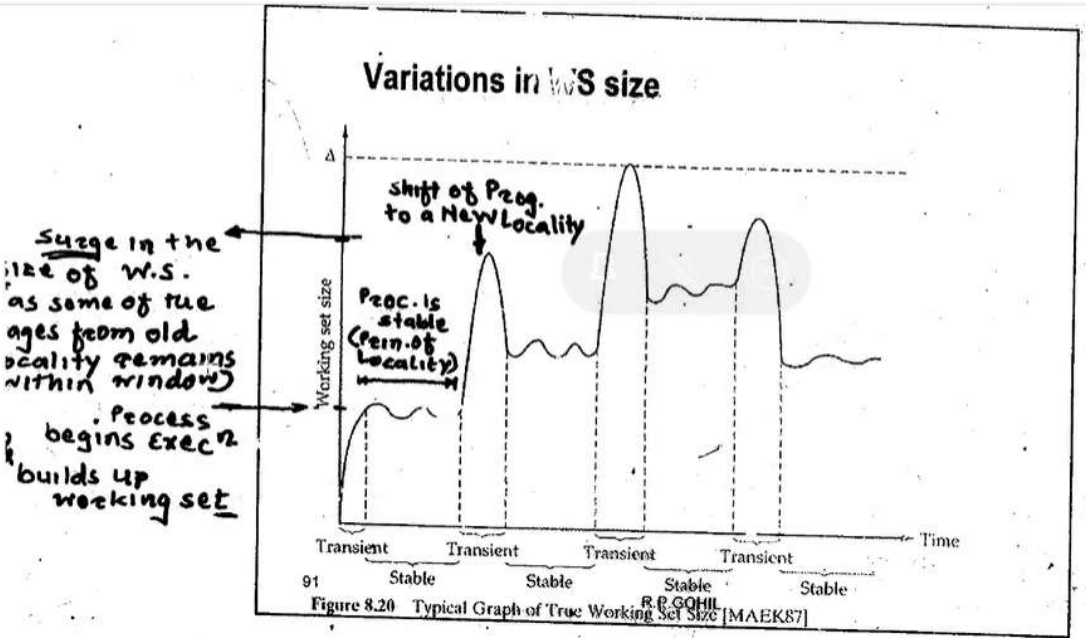


Figure 8.20 Typical Graph of True Working Set Size [MAEK87]

The Working Set Strategy

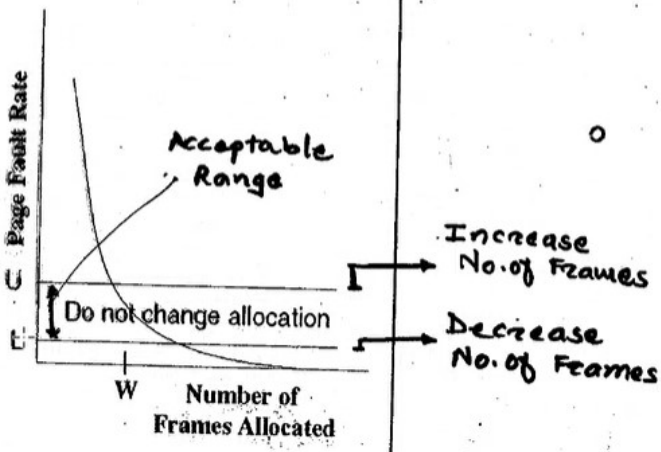
- **The working set of a process first grows when it starts executing**
- **Then stabilizes by the principle of locality**
- **It grows again when the process enters a new locality (transition period)**
  - up to a point where the working set contains pages from two localities
- **Then decreases after a sufficient long time spent in the new locality**

The Working Set Strategy

- **Practical problems with this working set strategy**
  - measurement of the working set for each process is impractical
    - necessary to time stamp the referenced page at every memory reference
    - necessary to maintain a time-ordered queue of referenced pages for each process
  - the optimal value for Δ is unknown and time varying
- **Solution: rather than monitor the working set, monitor the page fault rate!**

The Page-Fault Frequency Strategy

- Define an upper bound U and lower bound L for page fault rates
- Allocate more frames to a process if fault rate is higher than U
- Allocate less frames if fault rate is < L
- The resident set size should be close to the working set size W
- We suspend the process if the PFF > U and no more free frames are available



R.P.GOHIL

me decide "too high" ? (Paging rate)

criterion"

Max utilization

L = Mean time bet" Page faults

S = " " to service a Page fault

and control tries to keep L = S

Paging Disk is underutilized.

If L ≥ S process is swapped out

new can handle L ≤ S " " " in.