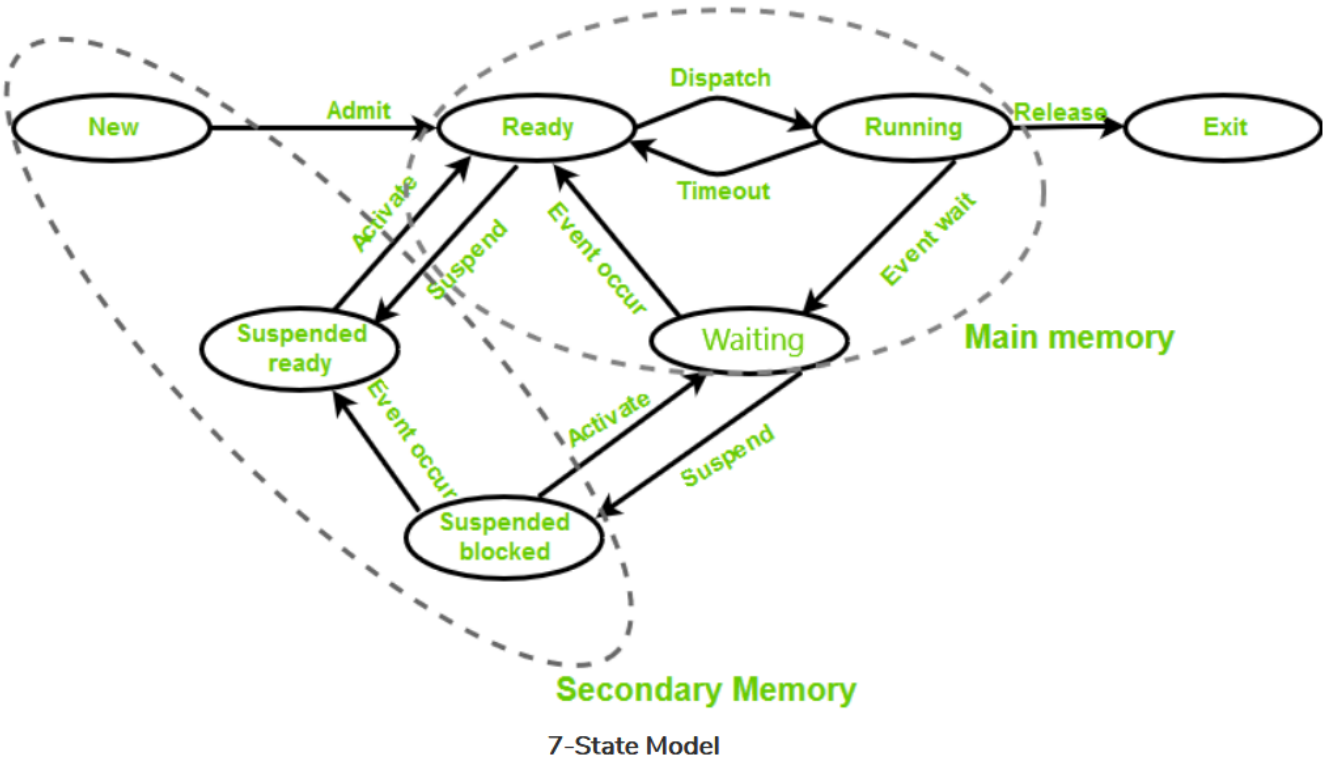
Process Management

Process

Text Section *:* A Process, sometimes known as the Text Section, also includes the current activity represented by the value of the ***Program Counter***.
Stack *:* The Stack contains the temporary data, such as function parameters, returns addresses, and local variables.
Data Section *:* Contains the global variable.
Heap Section : Dynamically allocated memory to process during its run time.

STATES OF PROCESS :

1. new
2. ready
3. run
4. blocked or wait
5. terminated or completed
6. suspended ready
7. suspended blocked or suspended wait



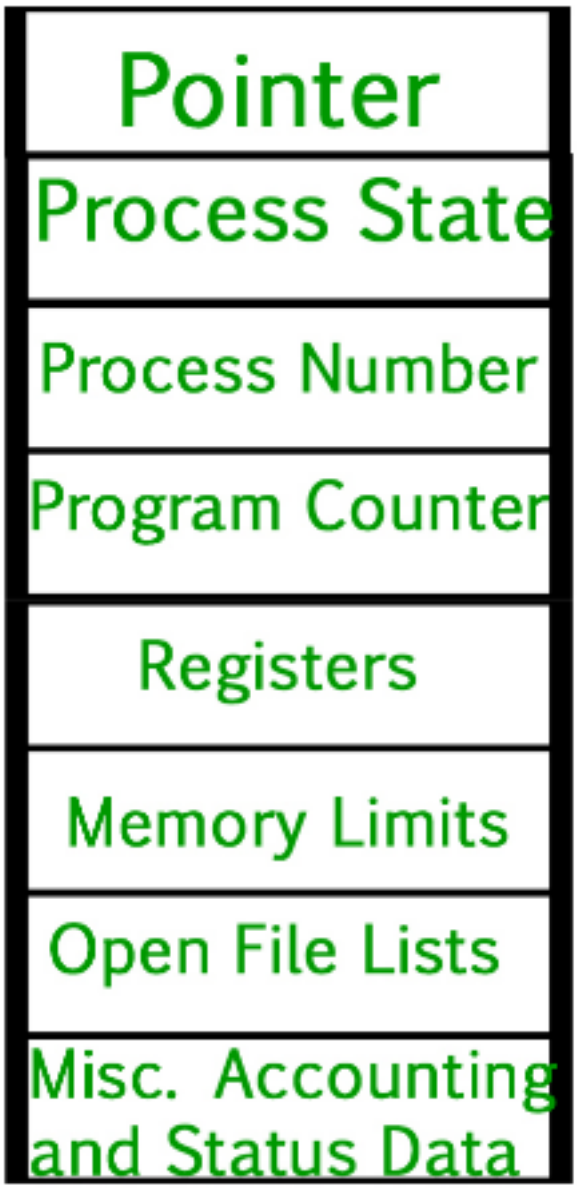
7-State Model

ATTRIBUTES OR CHARACHERTISTICS OF PROCESS (PCB - PROCESS CONTROL BLOCK) :

1. proces id
2. process state

3. cpu registers
4. accounts information
5. I/O status information
6. cpu scheduling information

all these attributes are called as context of process.



Process Control Block

program counter : it stores the address of next instructions to be executed for the process

PROCESS TYPES :

1. zombie process :

a process which finished its execution, but still has entry in the in the process table to report its parent process.

2. orphan process :

a child process currently in execution whose parent process no longer exits. In this case, the child process is adopted by the init process and then is reapes off

Process Scheduling

There are 3 types of schedulers :

1. Long term or job scheduler:
   It brings new process to ready state
   It controls Degree of Multiprogrmming
   It makes a carefull selection of both cpu and io bound processes.

2. Short term or CPU scheduler :
   It is responsible for selecting one process from ready state for scheduling it on the running process
   It only selects the process , but not loads it for running
   Dispatcher is responsible for loading process into running state
   Context swithing is also done by dispatcher

3. Medium term scheduler :
   It is responsible for suspending and resuming a process
   It mainly does swapping (moving from memory to disk and vice versa)

There are two types of multitasking :

1. pre-emption :
   process is forcefully removed from the cpu
   it is also called as multi tasking or time sharing

2. non pre-emption :
   process are not removed untill they complete the execution

Degree of multiprogramming :
The number of processes residing in a ready state at maximum

There are different types of queues in os :

1. Ready queue :

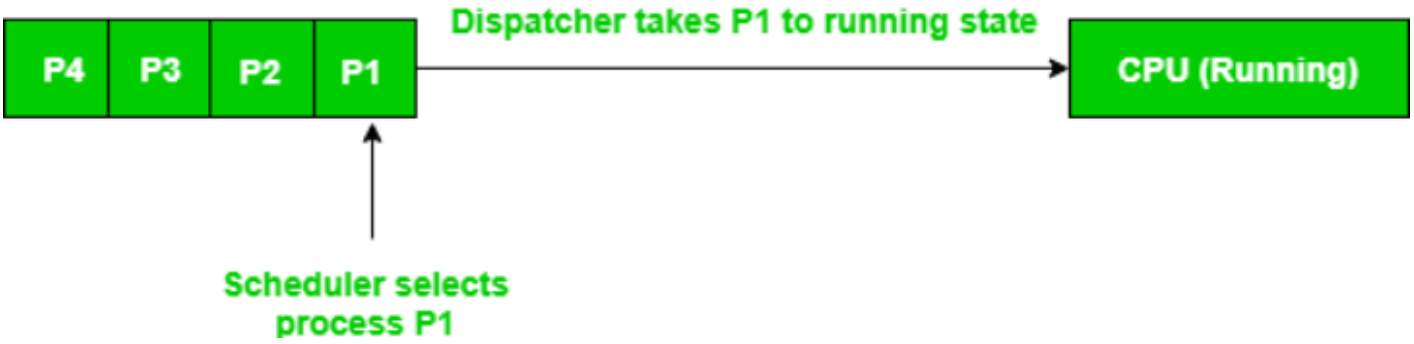
set of all processes are in main memory and are waiting for cpu time, are kept here

2. Job queue :
   each new process goes into job queue.
   process reside in mass storage and awaits the allocation of main memory

3. Waiting queue :
   set of process waiting for the alloction of i/o devices reside here

The short term scheduling selects a process from ready queue and yields control of the cpu to process

Scheduler and dispatcher :



Dispatcher is also responsible for :
1. content switching
2. switch to user mode
3. jumping to proper location when process again restarts

Scheduling Algorithms

1. First come first serve (FCFS) :
   follows first in first out (queue)
   Simple and easy to implement
   non pre-emptive
   causes convoy effect (the os slows down due to some slow processes)

2. Shortest job first (SJF) :

process is executed which has shortest burst time

non pre-emptive

sort all process in ascending order based on their burst time

now perform fcfs on the sorted list of processes

it has the minimum avg waiting time among all scheduling algorithm

may lead to starvation

it is greedy algo

it is practically infeasible since the os dont know the burst time and thus cant sort them

3. Shortest remaining time first (SRTF) :

pre-emptive version of sjf

if the burst time of the incoming process is less than the burst time of the currently running process, then the currently running process is kept in waiting and the incoming process is executed.

when all the process have arrived , this srtf will behave as sjf

practically infeasible since the burst time is unknown

may lead to starvation

4. Longest remaining time first (LRTF) :

pre-emptive version of ljf (longest job first)

when first process arrives cpu executes it , when a new process arrives the burst time of both the processes are compared.

if the remaining burst time of 1st process is less than that of new process , then the 1st process kept in waiting stage and the process with longest burst time is executed.

this process is repeated whenver a new process is arrived

5. Non pre emptive priority scheduling :

process with highest priority is executed first

process with same priority are executed in fcfs method

priority can be based on memory or time or any other resource requirements

6. Pre emptive priority scheduling :

process with highest priority is schedules first

when a process with priority greater than that of the running process arrives, the new process is executed

NOTE :

In both priority scheduling , it may lead to starvation or indefinte blocking

solution to this is Ageing.

Ageing is the process of gradually increasing the priority of the process that waits in the

system for a longer period of time.

7. Round Robin scheduling :

every process is allocated a fixed amount of time slot called time quantum

once the time slot finishes , the process is content switched with the next process in the in the queue

it is most common and practically used scheduling algorithm

the only problem is , time gets wasted in some processes.

8. Multilevel Queue scheduling (MLQ) :

It is required when some processes are grouped into categories.

the priority list is :

1. system process
2. interactve process (foreground)
3. batch process (background)
4. student process

all queue has their own schedulings :

queue 1,2 - round robin

queue 3 - fcfs

if there are some processes in all queue , then there are two scheduling processes :

1. fixed priority pre emptive scheduling method :

since priority list is : q1 > q2 > q3

only after completing q1 ,the remaining queues get the cpu

2. time slice method :

each queue gets a specific part of the cpu

eg: if q1 gets 50%

q2 gets 30% then

q3 gets 20%

example :

consider priority : q1 > q2

Process	Arrival Time	CPU Burst Time	Queue Number
P1	0	4	1
P2	0	3	1
P3	0	8	2
P4	10	5	1

grantt chart is :

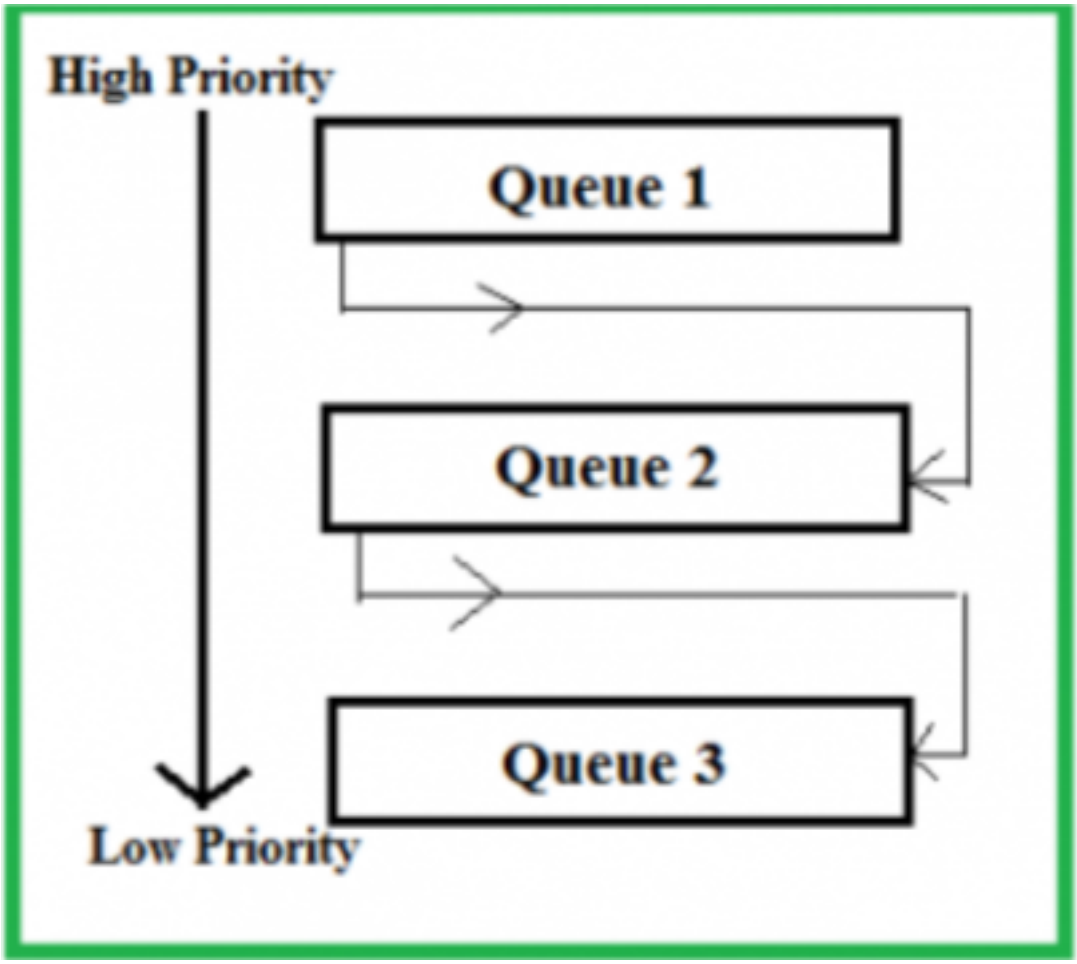
9. Multilevel feedback queue scheduling (MFLQ) :

it is modified mlq

processes can move between the queues

it keeps analyzing the time of execution of the processes, according to which priority can be changed

example :



let us consider that q1 , q2 follows round robin and q3 follows fcfs

q1, q2 has time quantum of 4 , 8 respectively

when a process starts executing , it enters q1

if it is completed or enters i/o operation its priority is not changed and if it comes to ready state then will execute in the q1 itself

if it is not completed in the given time then it is sent to q2

when the process in q2 is not completed in 8 quantum time , then it is sent to q3

q3 follows fcfs

processes in the lowest level suffers from starvation

it can be solved by boosting the priority of all processes and placing them in the highest priority queue

Application :

1. it is more flexible than mlq

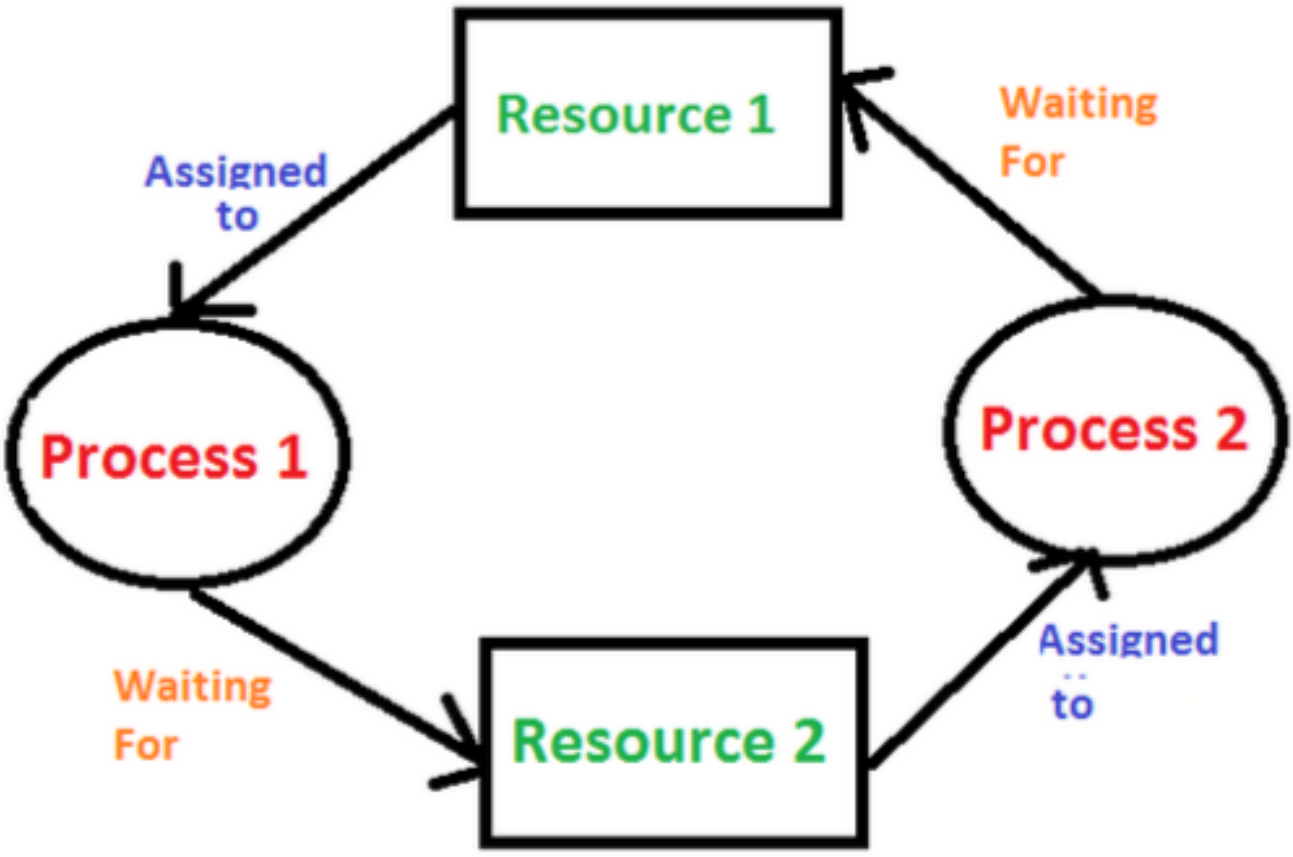
2. it reduces response time
3. it optimizes the turn around time algorithms like sjf in which run time of a process is needed to schedule them.

Deadlock characteristics

a process in os uses different resources in different ways :

1. request a resource
2. use the resource
3. release the resource

Deadlock is the situation in which the processes are blocked because each processes is holding some resources and waiting for another resource aquired by some process in a cyclic way



Deadlock can arise if the following four conditions hold simultaneously (Necessary Conditions)

1. **Mutual Exclusion :** One or more than one resource are non-sharable (Only one process can use at a time)

2. **Hold and Wait :** A process is holding at least one resource and waiting for resources.

3. **No Preemption :** A resource cannot be taken from a process unless the process releases the resource.

4. **Circular Wait :** A set of processes are waiting for each other in circular form.

Resource Allocation Graph (RAG):

Banker's algorithm uses a table like allocation to understand the state of the system (in terms of resources and processes)

tables are easy to understand
we can also use graph to understand the same thing
this graph is called resource allocation graph

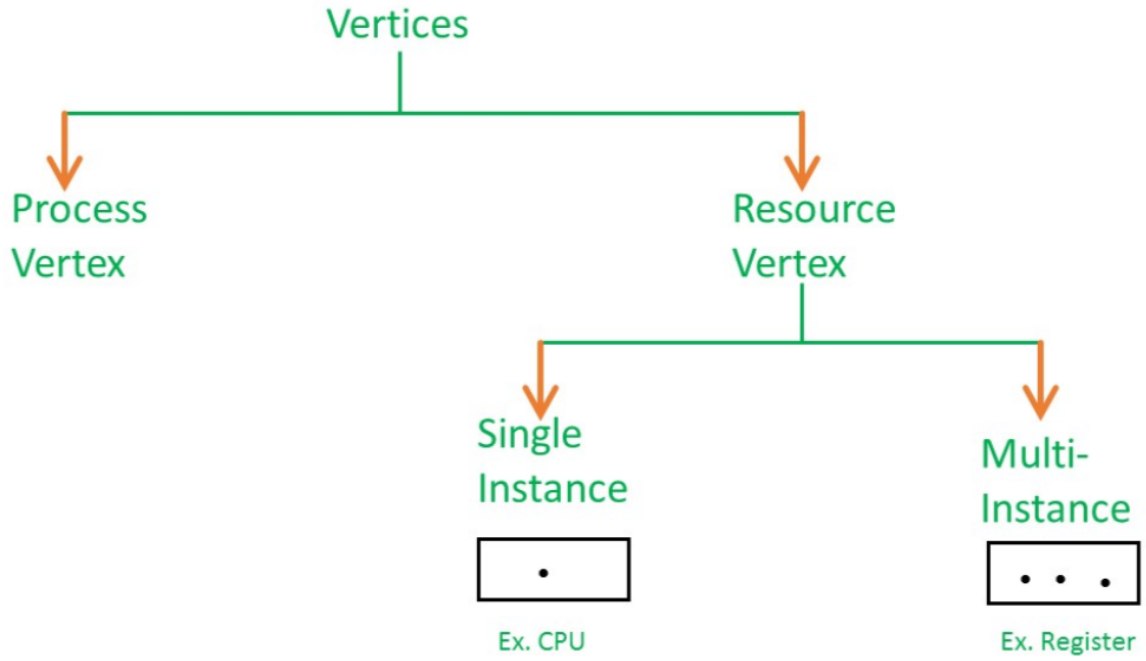
tables are better if the system has more num of process and resources
graphs are better if the system has less num of process and resources

RAG has vertices and edges :

there are 2 types of vertices

1. process vertex:
   every process will be represented by a process vertex
   every process is representes by a circle

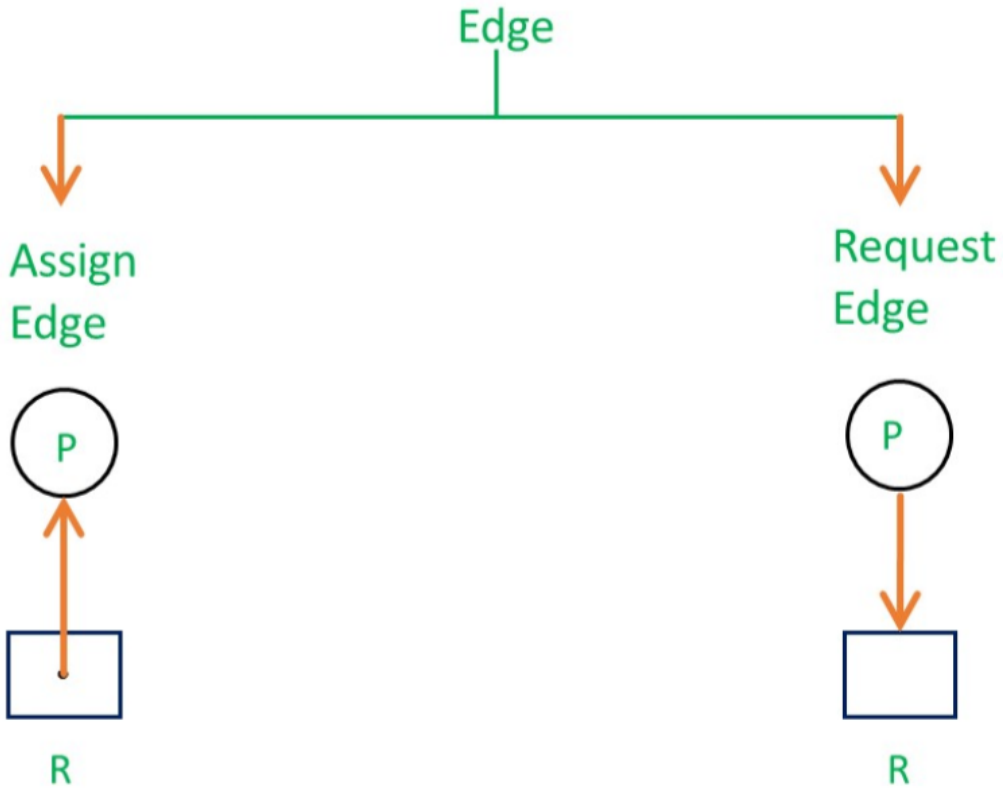
2. resource vertex :
   every resource will br reprensented by a resource vertex
   it is reprensented by a box
   there are 2 types:
   i) single instance type resouce :
   it has 1 dot inside the box
   num of dot reprensents how many instances are present of each resource type
   ii) multi instance type resource :
   it has many dots inside the box



there are 2 types of edges :

1. Assign edge :
   the resource is assigned to a process

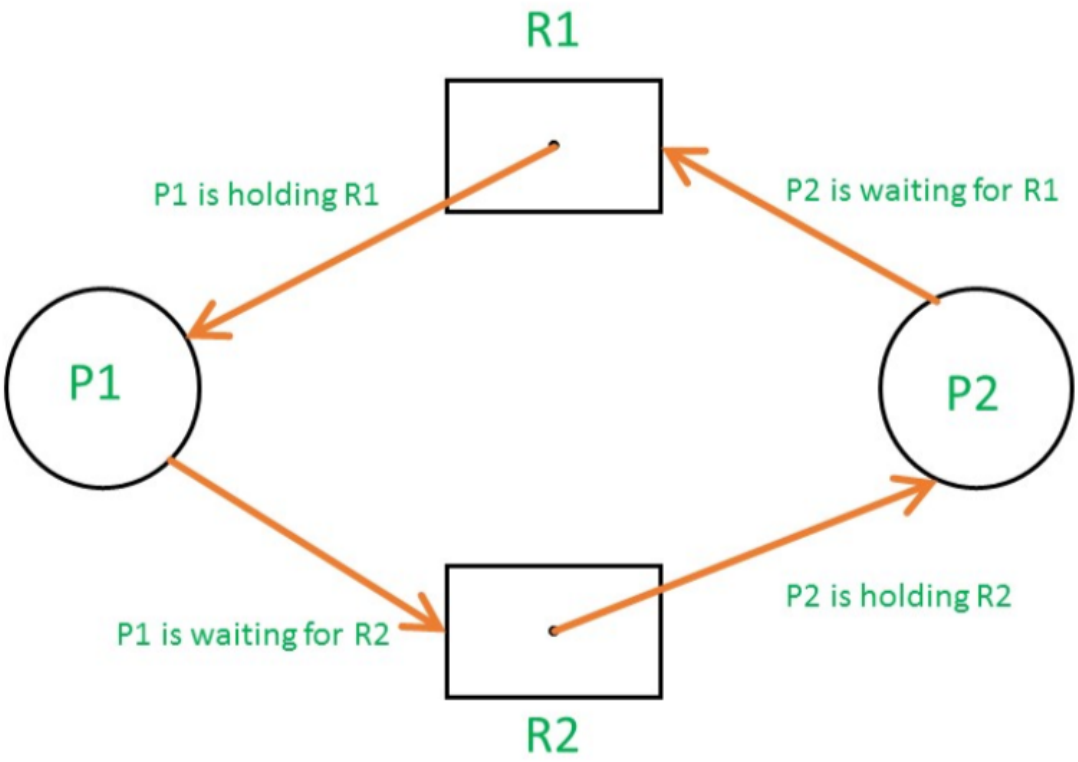
2. Request edge :
   in future the process will request a resource to finish its execution



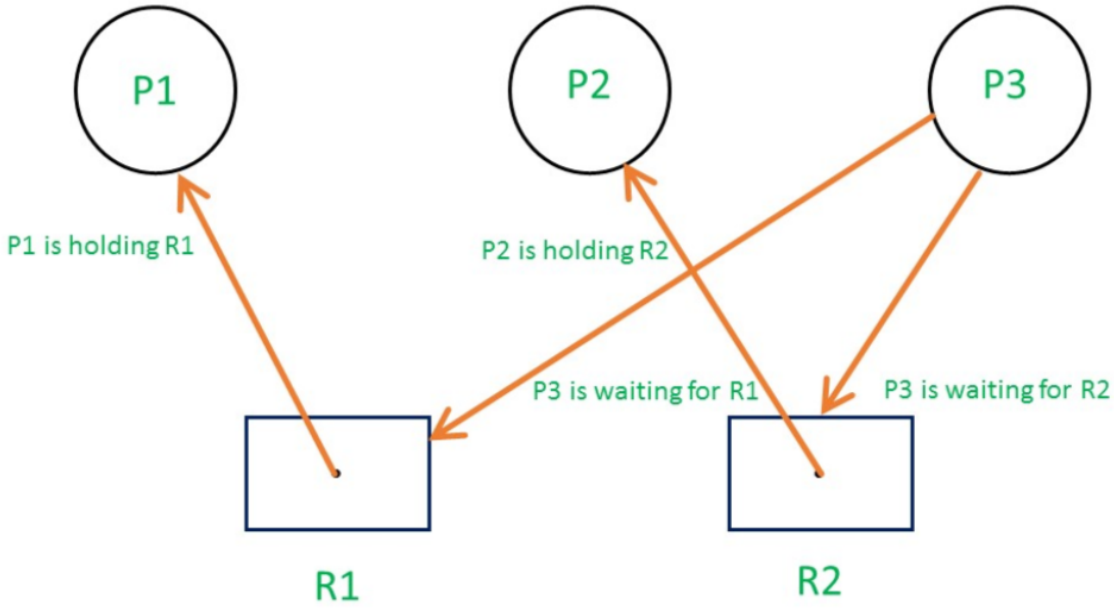
if process is using a resource , then an arrow is drawn from the resourse to the process

if process is req a resource , then an arrow is drawn from the process to the resource

example :



SINGLE INSTANCE RESOURCE TYPE WITH DEADLOCK



SINGLE INSTANCE RESOURCE TYPE WITHOUT DEADLOCK

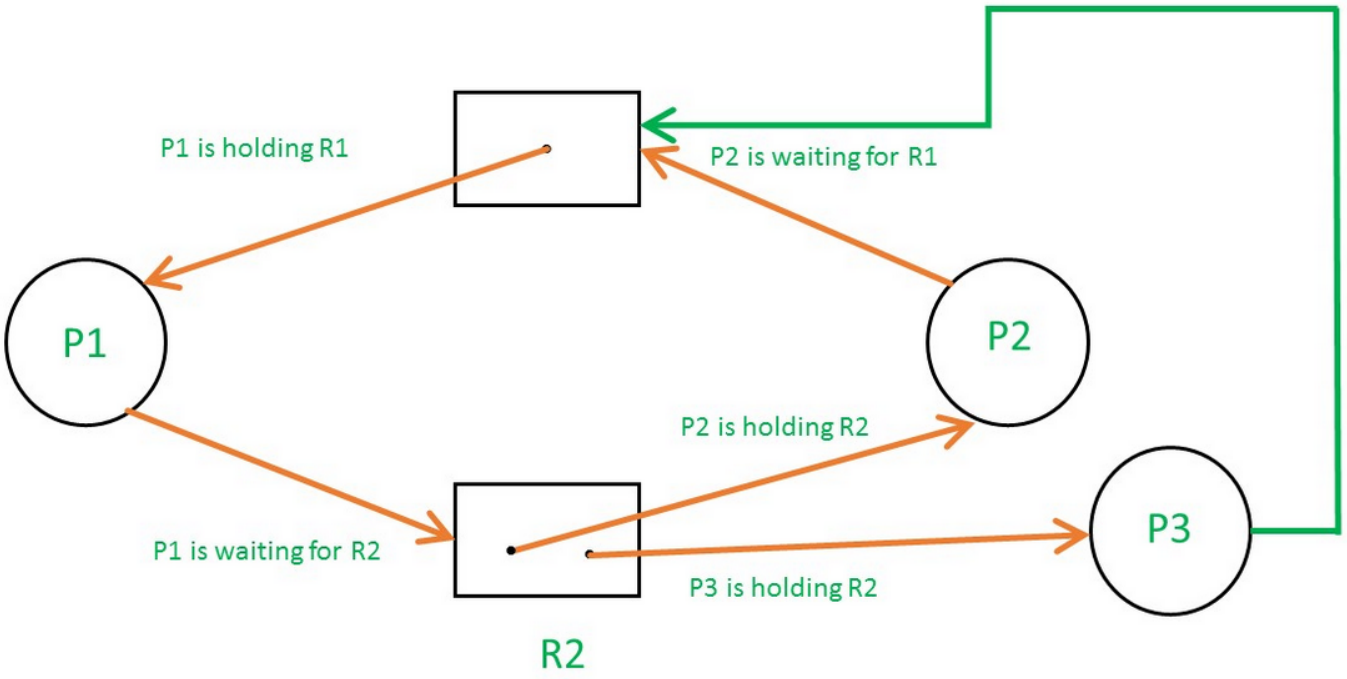
NOTE：

every cycle in a multi-instance resource type graph is not a deadlock, if there has to be a

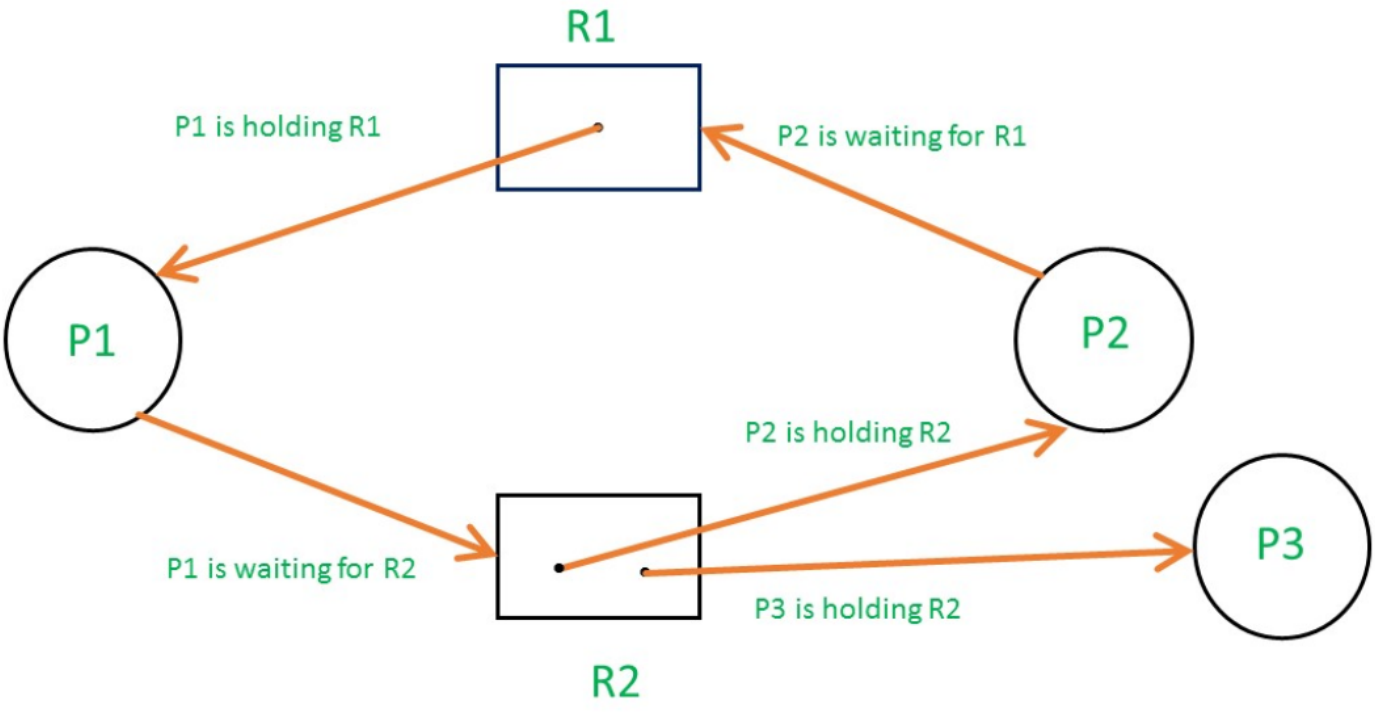
deadlock, there has to be a cycle.

So, in case of RAG with a multi-instance resource type, the cycle is a necessary condition for deadlock, but not sufficient.

example :



MULTI INSTANCES WITH DEADLOCK



MULTI INSTANCES WITHOUT DEADLOCK

s

Deadlock- handling

there are 3 ways to handle deadlock :

1) deadlock prevention
2) deadlock avoidance
3) deadlock detection and avoidance
4) ignore the problem altogether:
if deadlock is very rare , then let it happen and reboot the system.
this method is used in widows and unix systems

1) deadlock prevention :

the os accepts all sent requests . the idea is that any request that causes deadlock is not accepted

this is done by following any one of the conditions:

i) eliminate mutual exclusion
ii) eliminate hold and wait
iii) eliminate no preemption
iv) eliminate circular wait

2) deadlock avoidance :

the os carefully accepts and checks the requests . if any request causes deadlock then that request is avoided

it can be done with Banker's algorithm

Deadlock - recovery

There are three basic approaches to recover from deadlock:

1) Allow users to take manual intervention and inform the system operator.
2) Terminate one or more than one process which is involved in the deadlock.
3) Preempt resources.

There are 2 ways to terminate a process :

1) terminate all processes involved in deadlock. thus deadlock is broken but at the cost of all process termination

2) terminate processes one by one untill deadlock is broken . it is conservative approach and needs deadlock detection at every step

process pre emption:

1) selecting a victim
2) roll back
3) starvation

Process synchronication and critical section

based on synchronisatoin , processes are classified into 2 ways ":

1) independent process
2) co operative process (eg: cmd : ps | grep “chrome” | wc)

problems arises only in co operativr processes , which affecs other resources

the inconsistency among the interdependent stream of execution where each step is related to one another leads to the Race condition

Race condition :

several processes access and process the manipulation over the same data concurretly , then the outcome depends on the particular order in which the access takes place

Critical section :

places where shared variables are accessesd are called critical section
it is code segment that can be accessed by only one process at a time .
it contains shared variables which need to be synchronized to maintain consistency of data variables

Overview of process synchronisation

synchronisation mechanisms :

* disabling interupts
* locks or mutex
* semaphores
* monitors

applications :

* producer - consumer problems
* dining philospher problems
* bounded buffer problems
* reader - writer problem

goals of synchronisation mechanism:

* mutual exclusion
* progress
* bounded waiting (fair)

* performance

Interprocess communication (IPC)

IPC is a set interfaces, which is programmed in order for the processes to communicate woth each other and synchronize their actions

methods of IPC:

* pipes (same process) - have common origin
* names pipes (different process) - dont have a shared common process origin , eg:FIFO
* message queuing - managed by kernel and coordinated by an API
* semaphores - variable (>= 0) used to solve critical section problem
* shared memory - allows interchange of data thru a defined area of memory
* sockets - communicate with server and a client over a network

IPC thru shared memory (eg : global variables) s:

two or more processes can access the common memory
changes made by one process can be viewed by another proceess in the shared memory

only 2 copies of data are needed
write to shared memory and read from shared memory

IPC thru pipes, FIFO, and message queue :

for two processes to exchange informatio, it has to go thru the kernel
server reads and writes the data
client reads and writes the data
thus 4 copies of data are required (2 read and 2 write)

Processes can communicate with each other in two ways:

* shared memory
* message passing

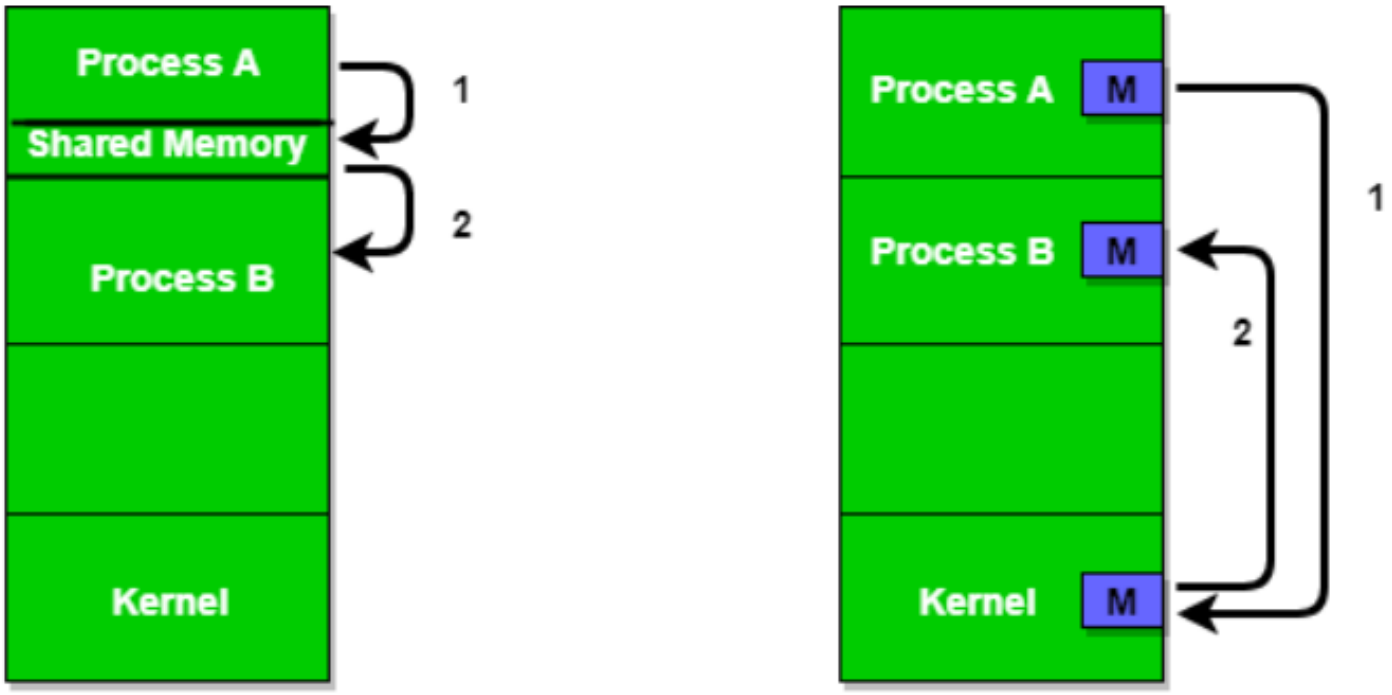
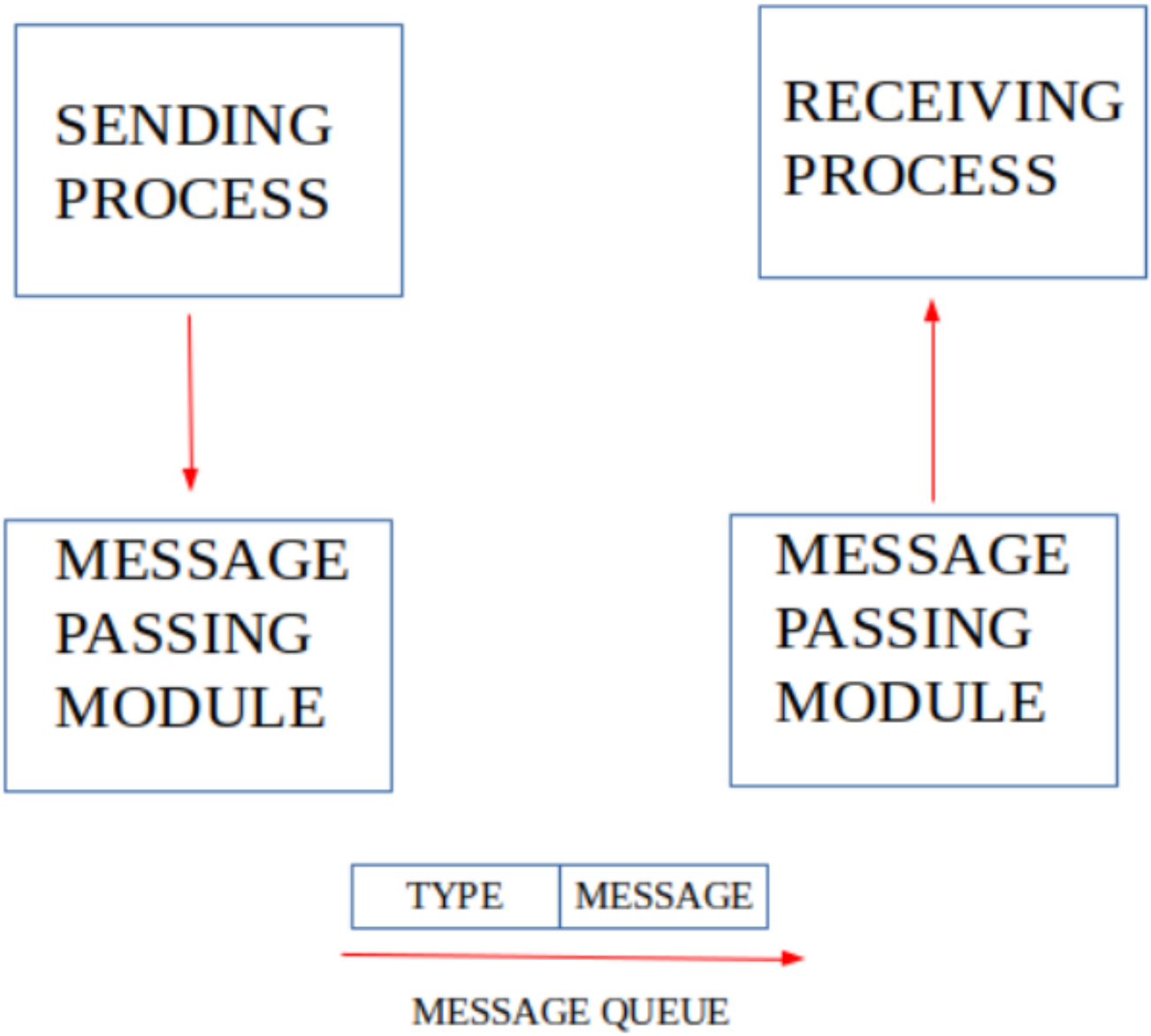


Figure 1 - Shared Memory and Message Passing

The standard messages have 2 parts :

* header
* body

msgget() - new queue is created
msgsnd() - new msg are added to end of the queue
msgrcv() - msg are fetched from the queue
msgctl() - destroy a msg queue
ftok() - generate a unique key



Mechanisms of synchronization

Lock

Lock method :

just setting lock alone is not enough
it might fail to provide mutual exclusion
so we implement TSL (test and set lock)

TSL is atomic

TSL works in both ways:
it waits while the lock is true and also
when the lock is updated as true if it is false

limitaitions :
while the processe have to wait and they went into a waiting loop until the process inside completes its execution .
this can be avoided by using semaphores

Semaphore

Semaphores :
it is a variable, used to solve critical section problems (used in multiprocessing env)
there are two types :
* counting semaphore
* binary semaphore

```
struct sem{
        int count;
        Queue q;
};
```

1) Counting semaphore:
analogy to restroom
process have to wait in a loop untill one process finishes and releases the lock
waiting in a loop is removed by this semaphore (acts like a guard and tells the process to do some othe task)
once the process in over inside , the semaphore tells the processes in the queue to enter into the restroom (critical section)
these semaphores maintians all the in and out records

wait() - it keeps the record of the count
stores the num of resources available
every time a process is allocated to a resource , the value of count is decreased
the -ve value indicates , how many processes need the resource

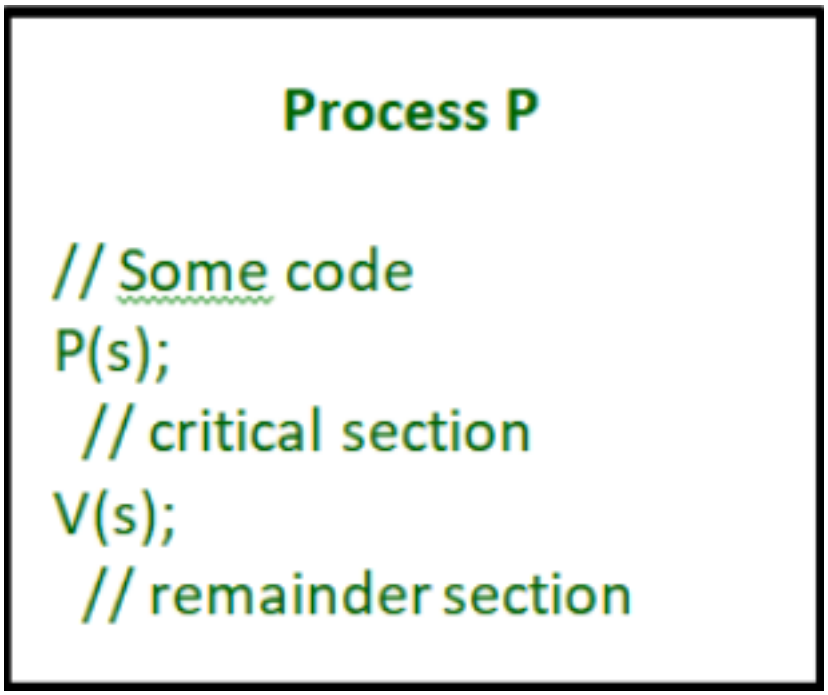
signal() - this increments the value of count by one , every time a process has finished execution
the processes in the queue are released in FCFS basis

P operation - called wait, sleep, or down operation

V operation - called signal, wake-up, or up operation

NOTE :

both p,v operations are atomic and semaphore is always initialized to 1
a critical section is surrounded by both operations to implement process synchronization

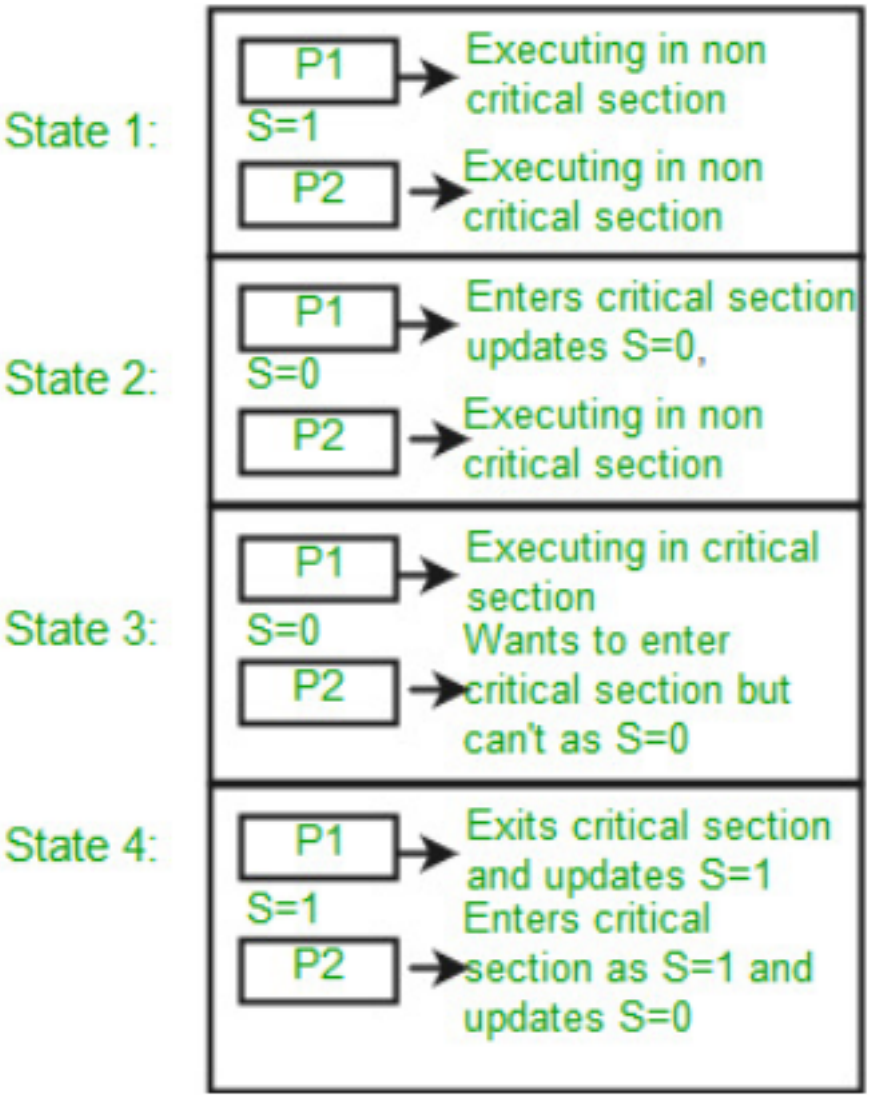


2) Binary semaphore :

the counter logically goes b/w 0 and 1 , similar to lock with two values : closed / opened

only two processes can enter into the critical section at any time
mutual exclusion is guranteed

implentation is as follows :
p1, p2 are processes
s is the semaphore



limitations in implementing semaphore:

* process in a queue checks the semaphore value and wastes CPU cycle

solution :

to keep in waiting queue , system calls block() on that process

when the process is completed, it calls the signal func and one process in queue is resumed using wakeup() system call

Mutex :

mutex is a binary semaphore

it provides locking mechanism and mutual exclusion to a section of code

NoTe: binary semaphore and mutex are not same, they are different

binary semaphore apart from locking mech, also provides two atomic operation signal and wait

TIPS:

solution to a common producer - consumer problem :

consider a 4096 byte buffer. producer writes to the buffer and consumer reads from the buffer. But only one can be done at a time in a buffer

using Mutex:

mutex maintains mutual exclusion and gives the key either to producer or consumer

one have to wait untill the other one is finished

using Semaphore :

it is a generalised mutex

it splits the 4KB buffer into 4 * 1KB buffer

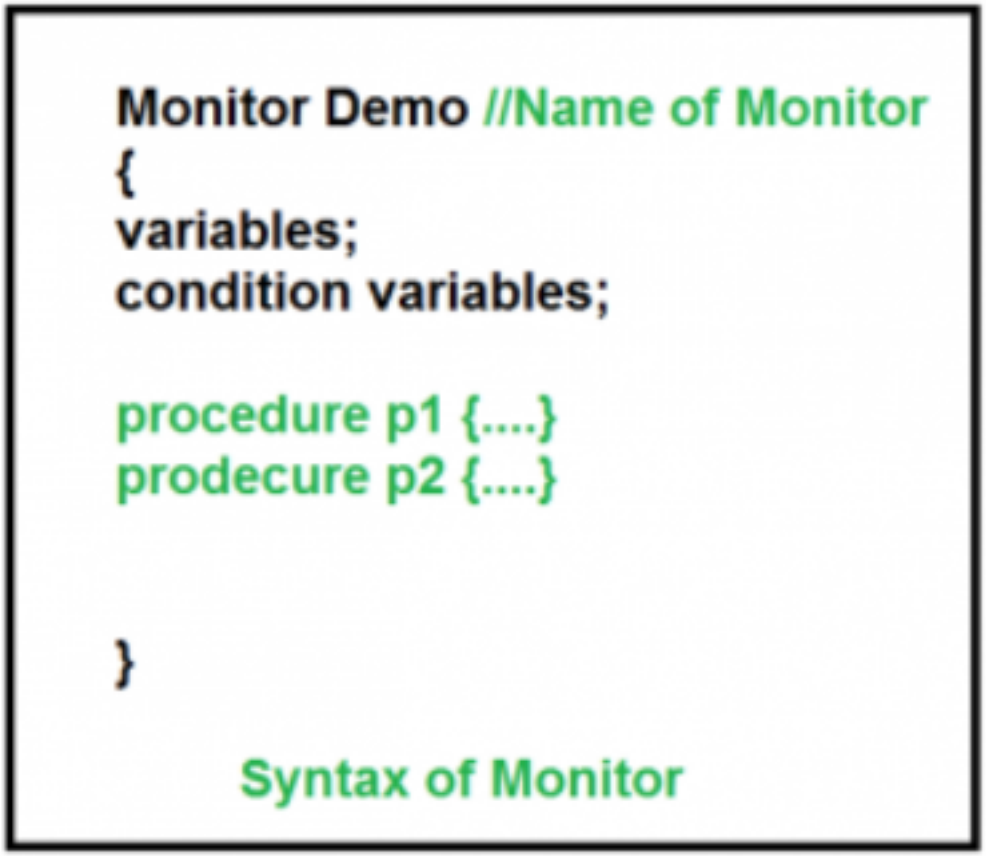
it can be assignes with all 4 buffers

thus producer and consumer can run at the same time at differnet buffers

Monitor

monitor is one of the way to achieve process synchronisation
it is supported by a programming language
ex: java synchronisation method . it has wait() ans notify()constructs

Syntax of Monitor



wait operation (x.wait()) :

process performing wait operation on any conditon variable are suspended

suspended processes are placed in a block queue of that conditional varible

each condition variable has its unique block queue

signl operation (x.signal()) :

when a process performs a signal variable on a condition variable , one of the blocked process is given chance

Priority inversion

L - lower priority process
M - middle priority process
H - higer priorith process
CS - critical section

L, H has shared CS
M dont have any shared resources with L, H

consider a situation :

1) L is running , but not in CS; H wants to run , H preempts L . H releases control ; L resumes

2) L is running in CS, H needs to run but not in CS . H preempts L , H releases control ; L resumes

3) L is running in CS, H wants to run in CS . H waits for L to finish. After L finishes , H starts executing

this situation is called priority inversion
L is executing and H is waiting for L

consider the priority as : L < M < H

4) L is running in CS, H wants to run in CS. H is waiting for L. Now M dont want to run in CS, hence it preempts L and it takes the CS
this creates H to wait for a longer time causing severe problems

there are many solution for this problems , one such thing is priority inheritance

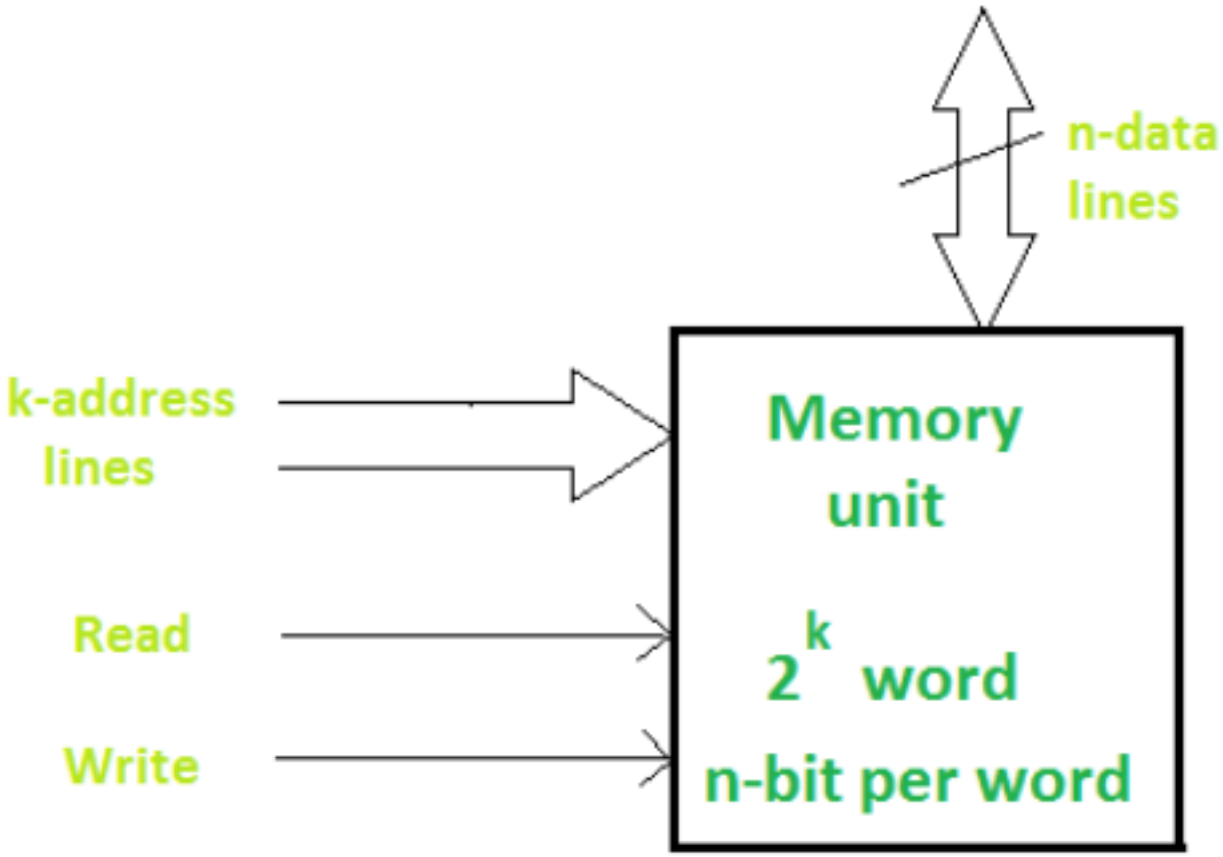
Priority Inheritance :

when L is running in CS and H wants to run in CS, H waits for L .
now when L acquires the priority of H
hence M cant preempt L (which is running with priorith of H)
after L finishes , it will take back its lower priority back again
Now H starts to run in CS , M waits for H to release control

Memory Management

memory is made up of registers
each register is one storage location (also called as memory location)

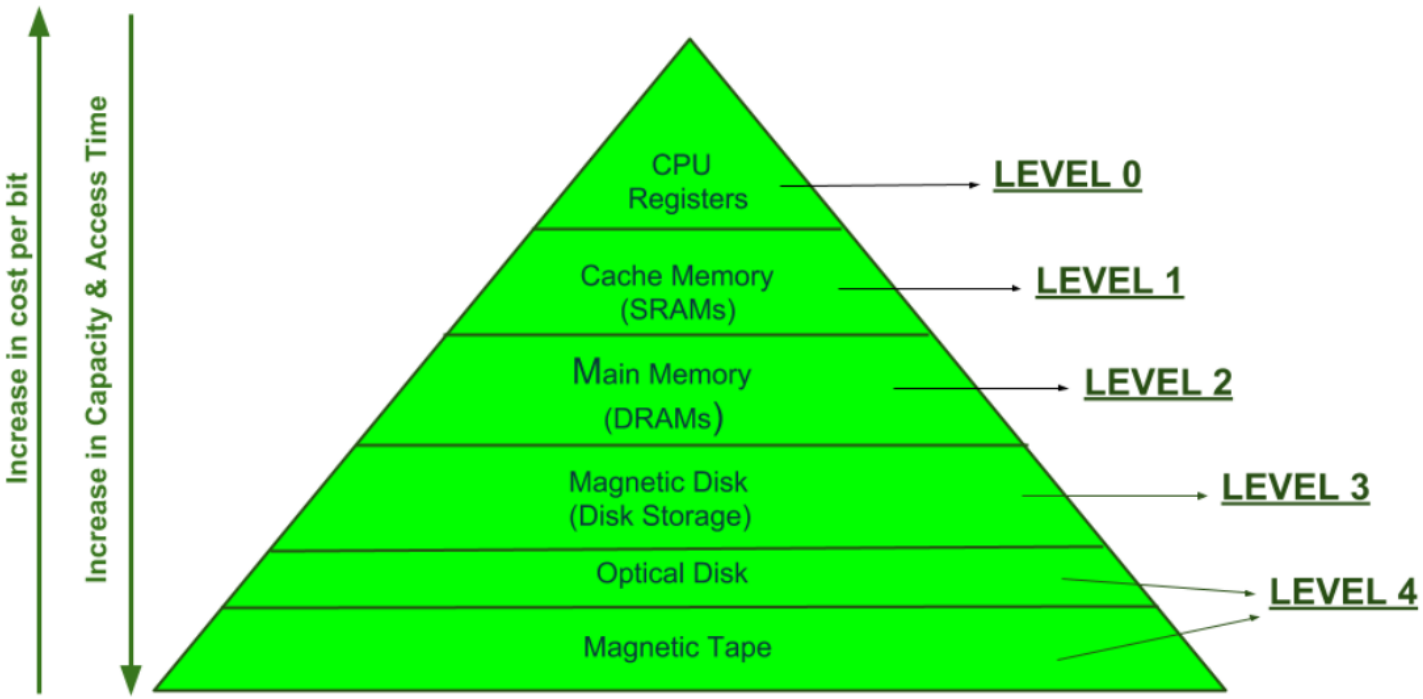
A storage location is made up of storage elements called cell, in which one bit of data is stored



dataline provides the information to be stored in memory
the control inputs specifies the direct transfer
k-address line specifies the chosen word
when there are k address line , 2**k words can be accessed

Memory hierarchy

Memory hierarchy :



MEMORY HIERARCHY DESIGN

memory hierarchy is divided into 2 types :

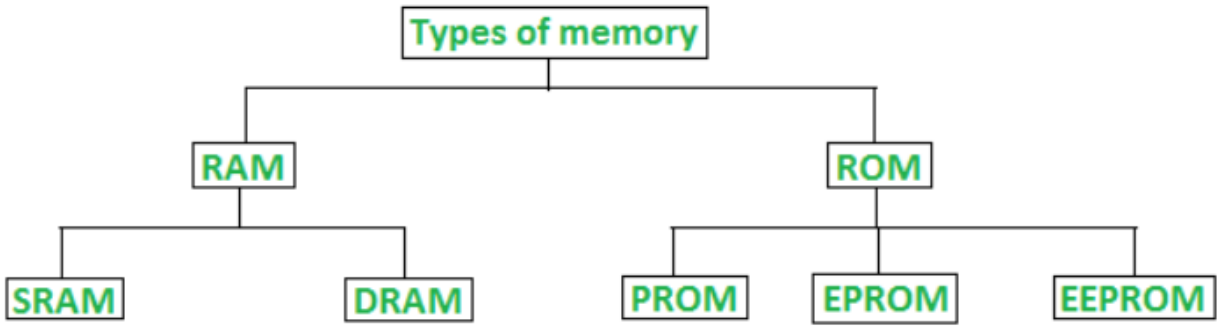
* external or secondary memory :
  can be accessible by the processor via I/O module
  lvl 4, 3 memories comes under this

* internal or primary memory :
  can be directly accessible by the processor
  lvl 2, 1, 0 memories comes under this

Memory types

memory is of 2 basic types :

* volatile / primary memory - RAM
* non-volatile / secondary memory - ROM



Classification of computer memory

RAM - random access memory
SRAM - static RAM
DRAM - dynamic RAM

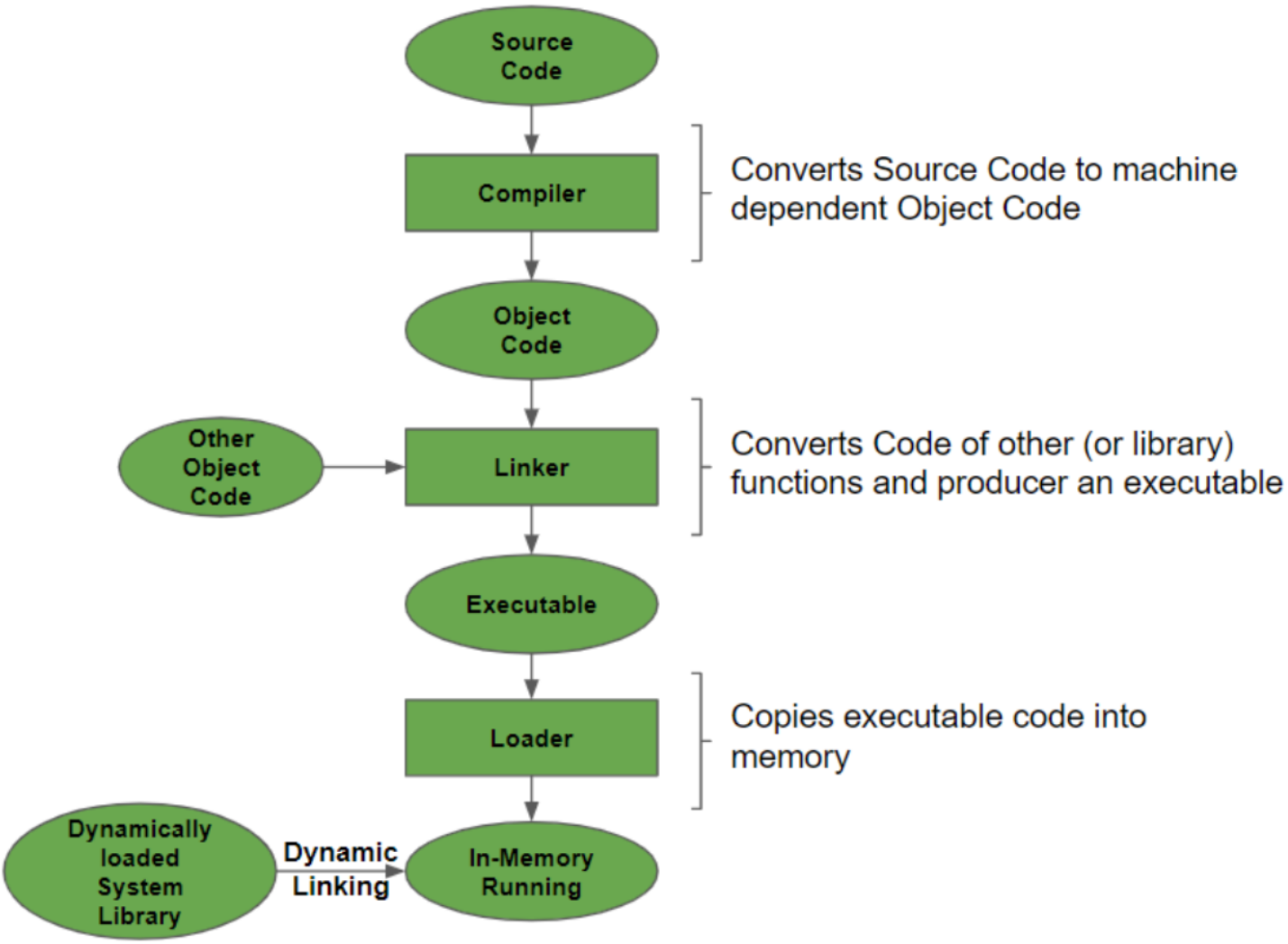
ROM - read only memory
PROM - programmable ROM
EPROM - erasable programmable ROM
EEPROM - electrically erasable programmable ROM

SRAM is faster and expensive than DRAM
SRAM can not store many bits per chip like DRAM

RAM	ROM
1. Temporary Storage.	1. Permanent storage.
2. Store data in MBs.	2. Store data in GBs.
3. Volatile.	3. Non-volatile.
4.Used in normal operations.	4. Used for startup process of computer.
5. Writing data is faster.	5. Writing data is slower.

Difference between RAM and ROM

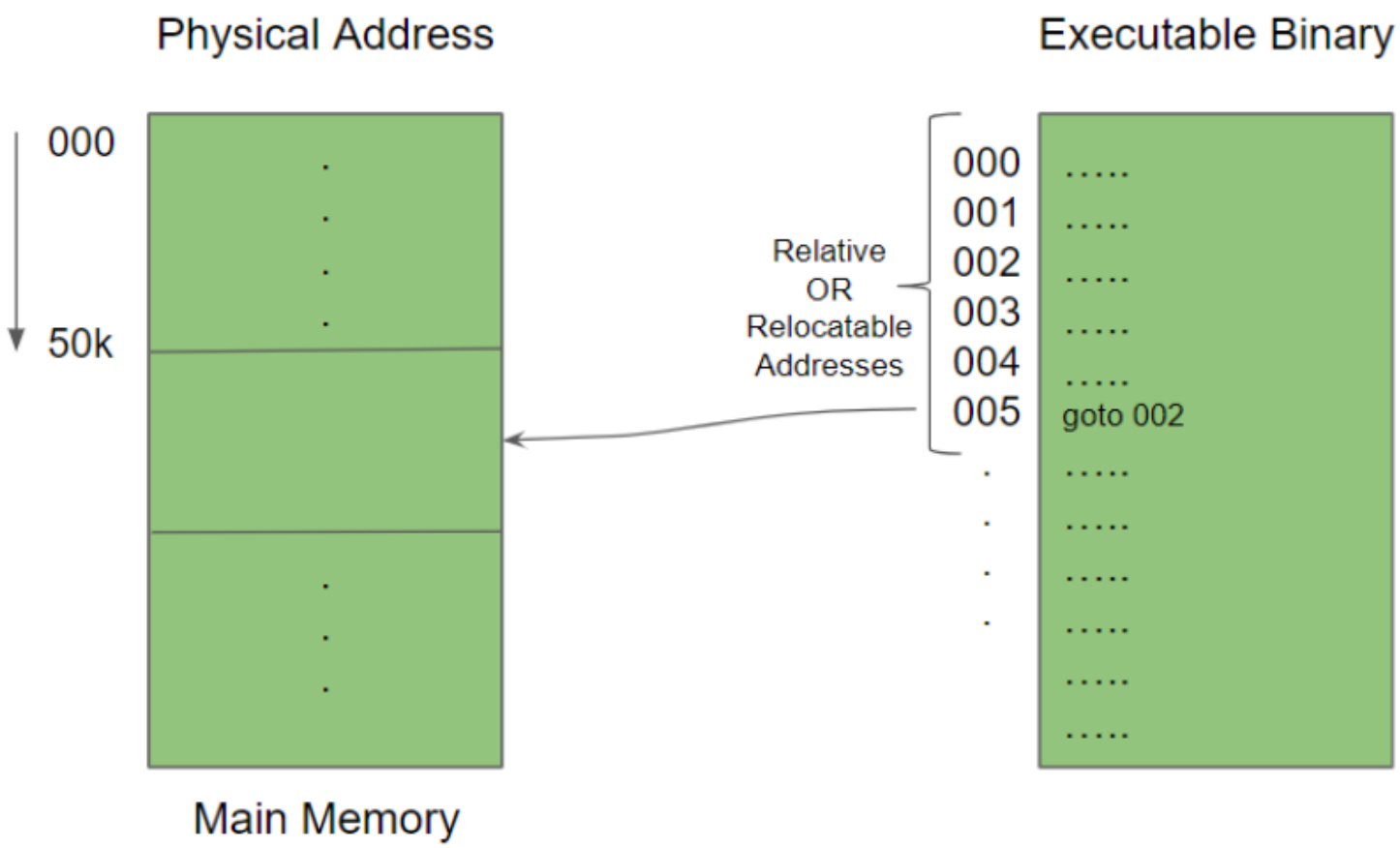
Programs compiled and run



Linkers are of two type :

* static linking (execution is done by line by line)
* dynamic linking (done during load or run time)

Address binding



address binding refers to the mapping of relocatable or relative address to physical addresses
ex : functions like goto , load , add , etc
to execute the sort of binary files , they need to be loaded into the main memory , which has physical slots
when the binary files are loaded into the slots after 50k , they are adjusted to 50k, 50k+1, 50k+2 , etc

These bindings can happen at various stages of execution :

1) compile time :
   - static
   - generates virtual / logical address
   - instructions are translated into an absolute address

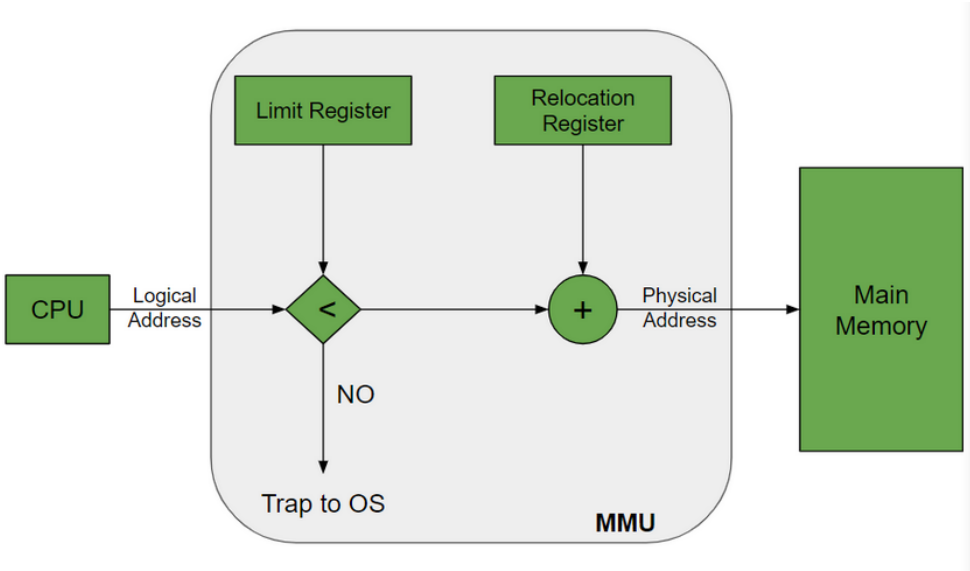
2) load time :
   - static
   - generates physical address
   - absolute addresses are converted to relocatable address
   - instructions are loaded in memory
   - once loaded into the main memory , it cant be relocated

3) run time :
   - dynamic and preferred in modern os
   - dynamic absolute address is generated
   - it works b/w and helps in execution

from memory, instructions are executed by CPU
the whole process takes place in the hardware

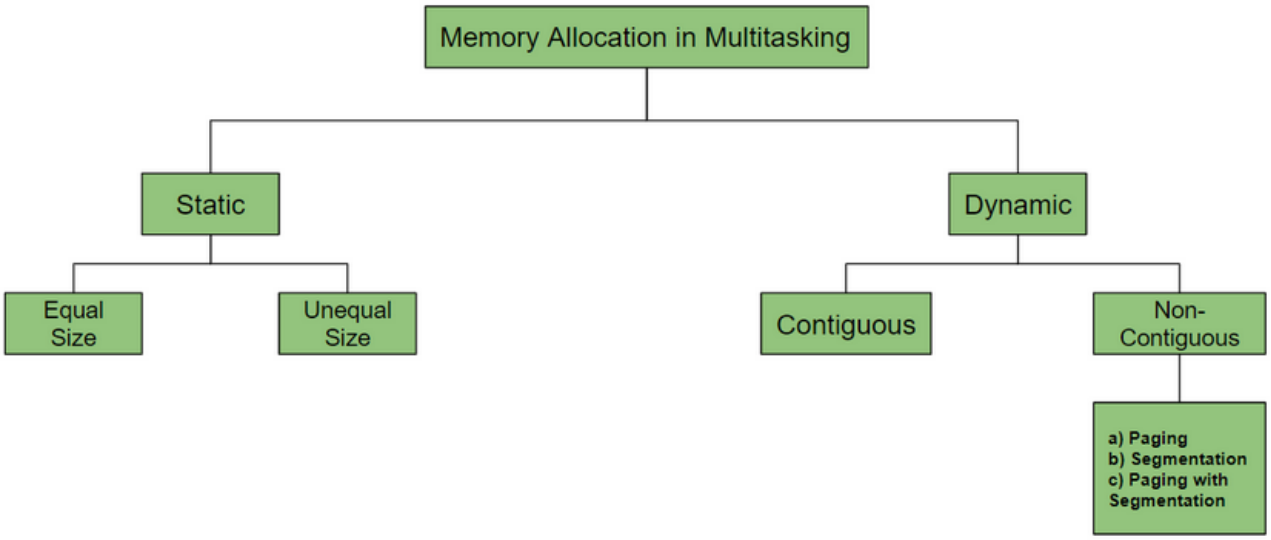
it has two registers
* limit register
* relocation register

the below diagram is for runtime binding :



Evolution of memory management

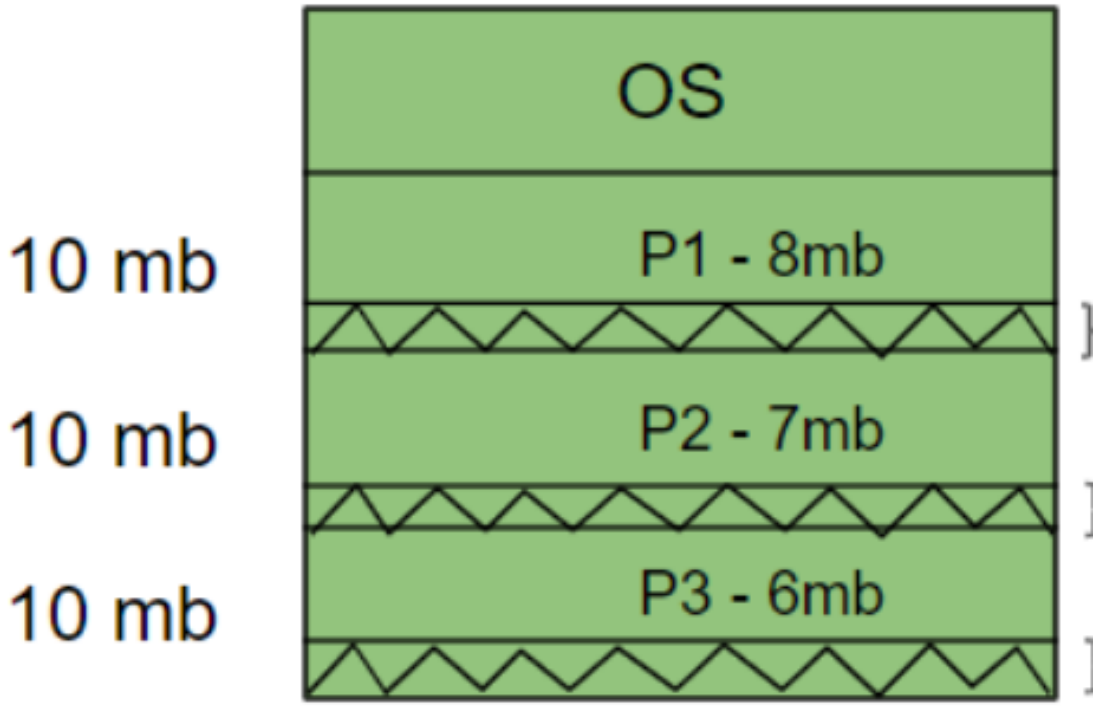
memory management involves 2 phase :
* single tasking system (processes are executed one by one)
* multu tasking system



1) static / fixed partitioning technique (contiguous allocation) :

used to put more processes in the main memory
non-overlapping partitions in ram is fixed but size may be same or different
no spanning is allowed
partition in main memory are made before execution or during system configure

1.1) equal size memory allocation :



each process is allocated with 10mb , but extra memory are wasted
total wasted memory is 2+3+4 = 11mb
but an P4 with 5mb size cant be added
this is called external fragmentation

each partitions has some memory wasted (2mb, 3mb, 4mb)

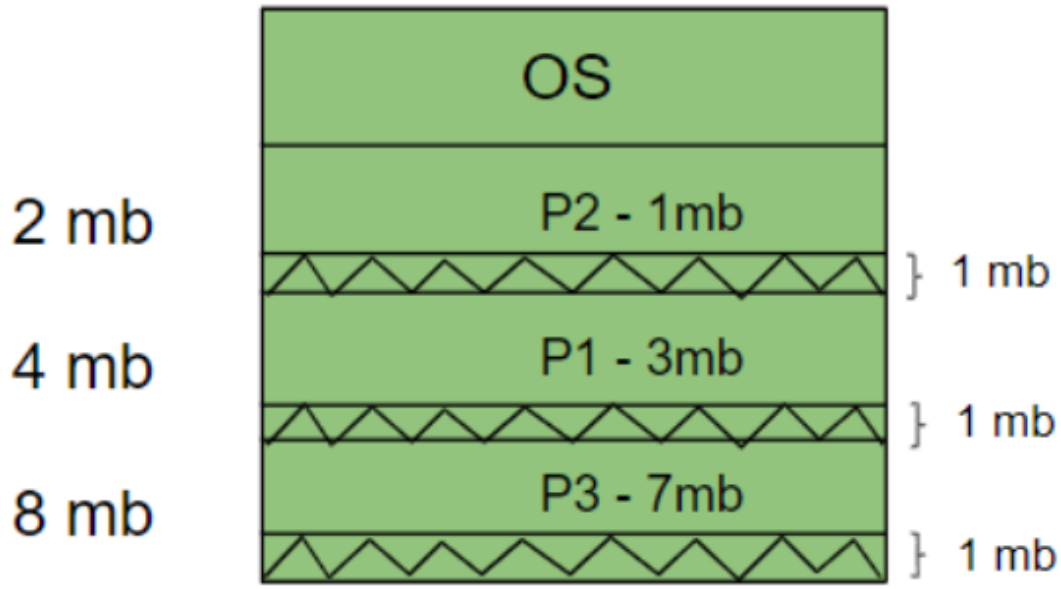
these are called internal fragmentation

in equal size memory allocation , the internal fragmentation leads to external fragmentation

NOTE :

fragmentation occurs in a dynamic memory allocation system when many of the free blocks are too small to satisfy any request

1.2) unequal size memory allocation :



there is a total loss of 3mb

Advantages of fixed partitioning :

* easy to implement
* little os overhead

Disadvantages of fixed partitioing :

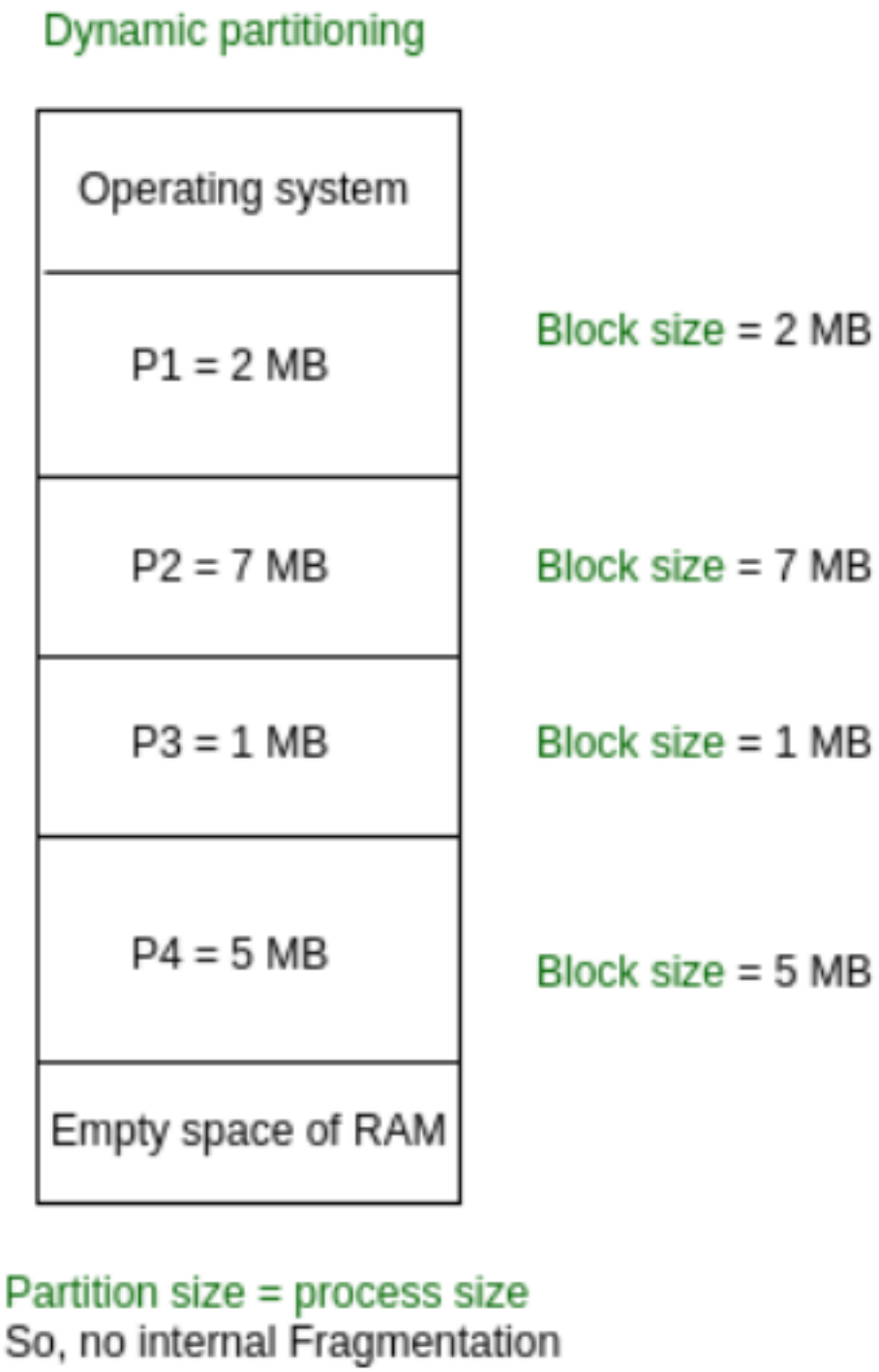
* internal fragmentation
* external fragmentation
* limit process size
* limitaition on degree of multiprogramming

2) Dynamic / variable partitioning technique (contiguous allocation) :

initially ram is empty and partitions are made during run time according to the process

need

size of partition will be equal to the incoming process
internal fragmentation can be avoided

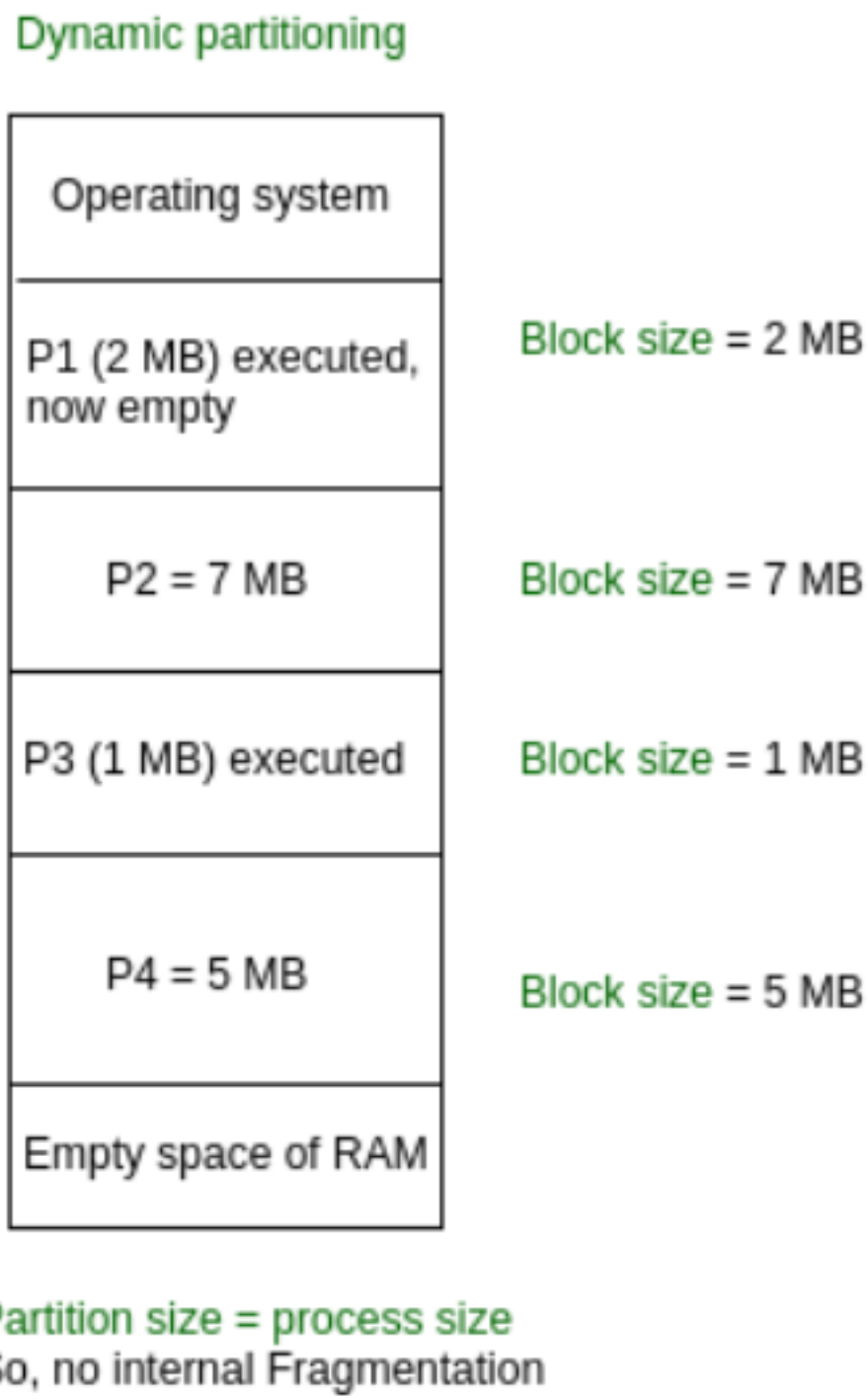


Advantages of variable partitioning :

* no intenal fragmentation
* no restriction on degree of multiprogramming
* no limitaitions on the size of the process

Disadvantages of variable partitioning :

* difficult to implement
* external fragmentation



the empty space in memory cant be allocated as no spanning is allowed in contiguous allocation,

since the process must be contiguously present in main memory to get executed

hence it causes external fragmentation

these spaces are called holes

Dynamic partitioning

the problem with dynamic partitioning is that after execution , the memory is left as hole when more such holes are left , it leads to external fragmentation

there are 2 ways to solve this problem :

* bitmap
* linklist

1) Bitmap :

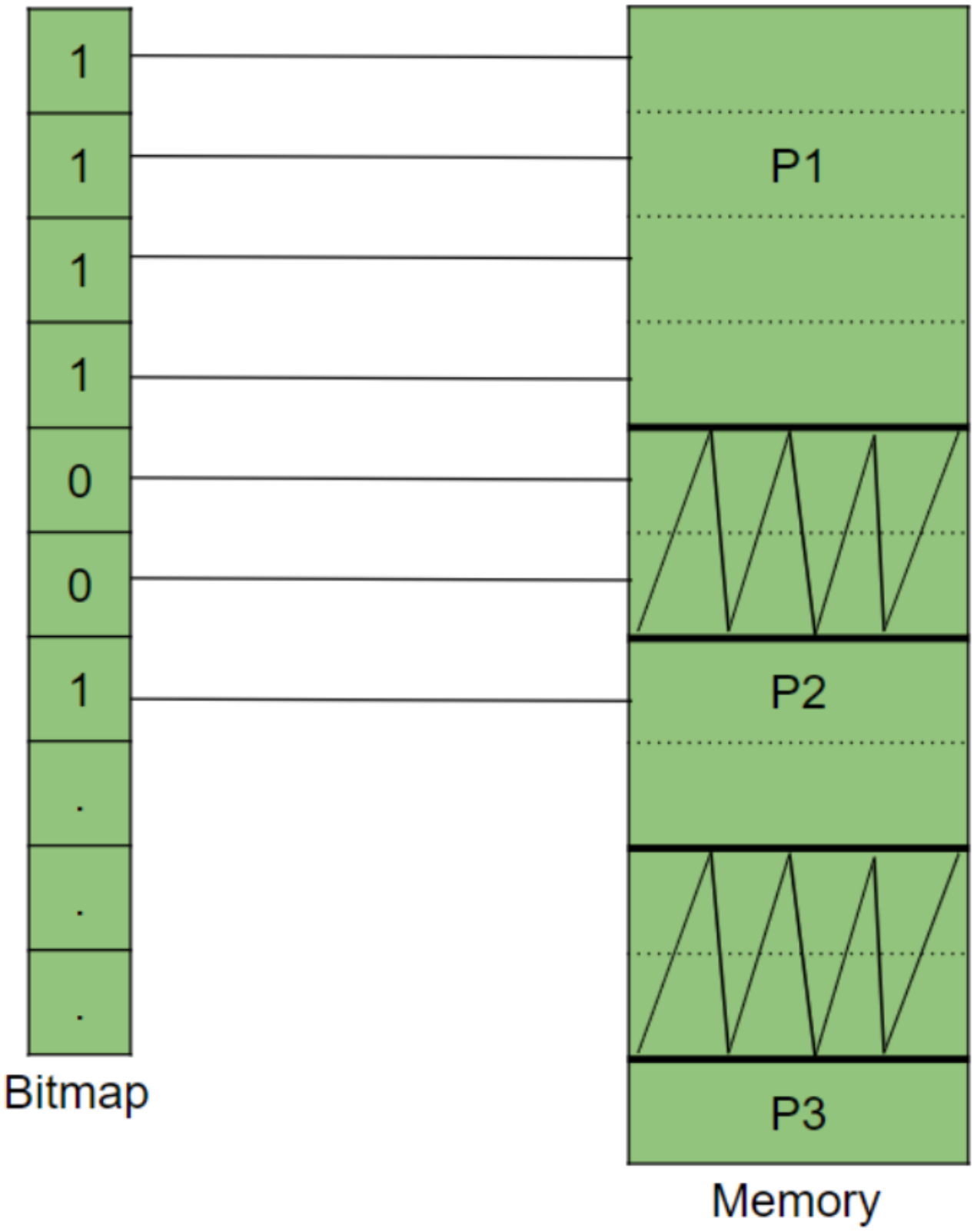
it keeps track of memory units as 1 ans 0

memory unit is occupied - 1

memory unit is free - 0

new processes are allocated to memory unit bitmap value 0

thus the problem is solved



Limitations :

it needs more space

for every memory unit , one bit of memory is used for the bitmap to store value

if memory units are taken in larger size , then internal fragmemtation occurs

thus both the processes are not feasible

2) Linklist :

the processes and the holes are connected using a doubly linklist

the holes are joined together to accomodate any incoming larger new process

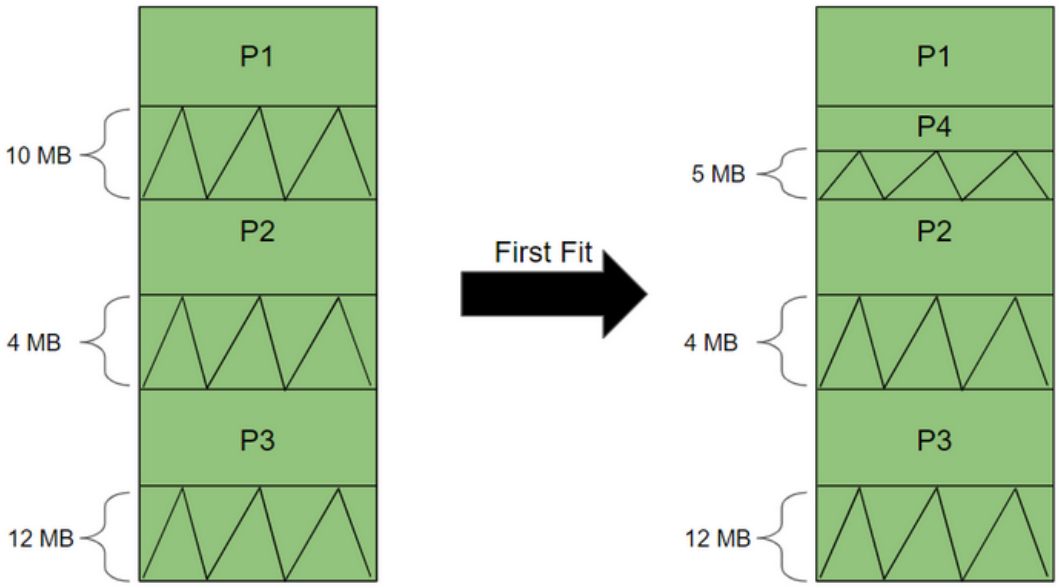
if many partitions are free, then whcih partition has to be allocated to the incoming process there are many ways to allocate partitions

Partition allocation schemes :

1) First fit :

the partition is allocated which is first sufficient block from top to bottom in the main memory

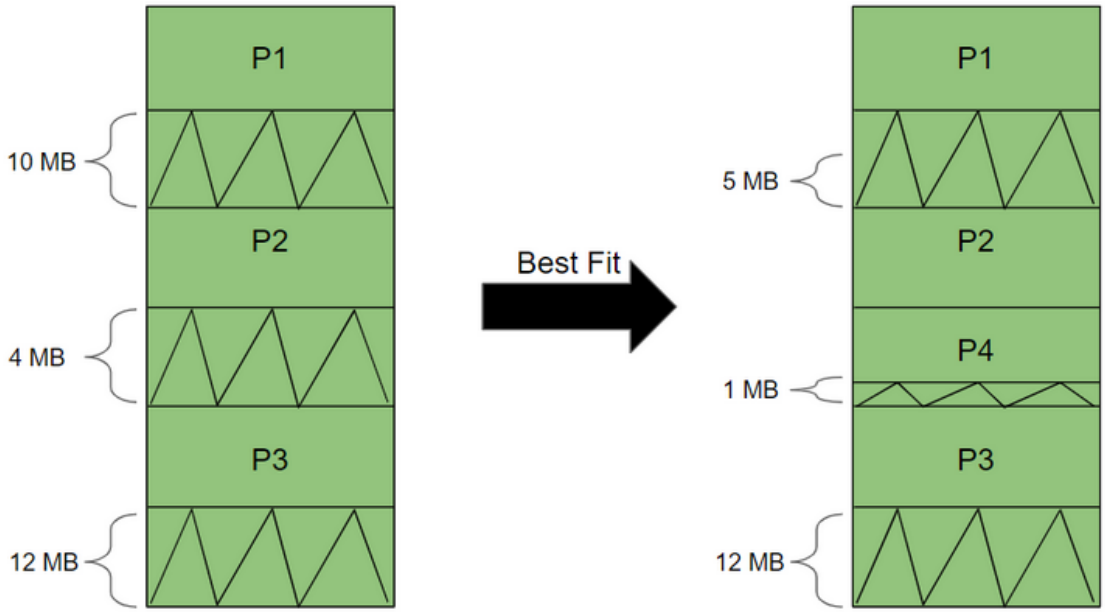
new process p4 of 5 mb is incoming



2) Best fit :

allocate the process to the partition which is the first smallest sufficient partition among the free available partition

a new process p4 of 3mb is incoming



Disadvantages :

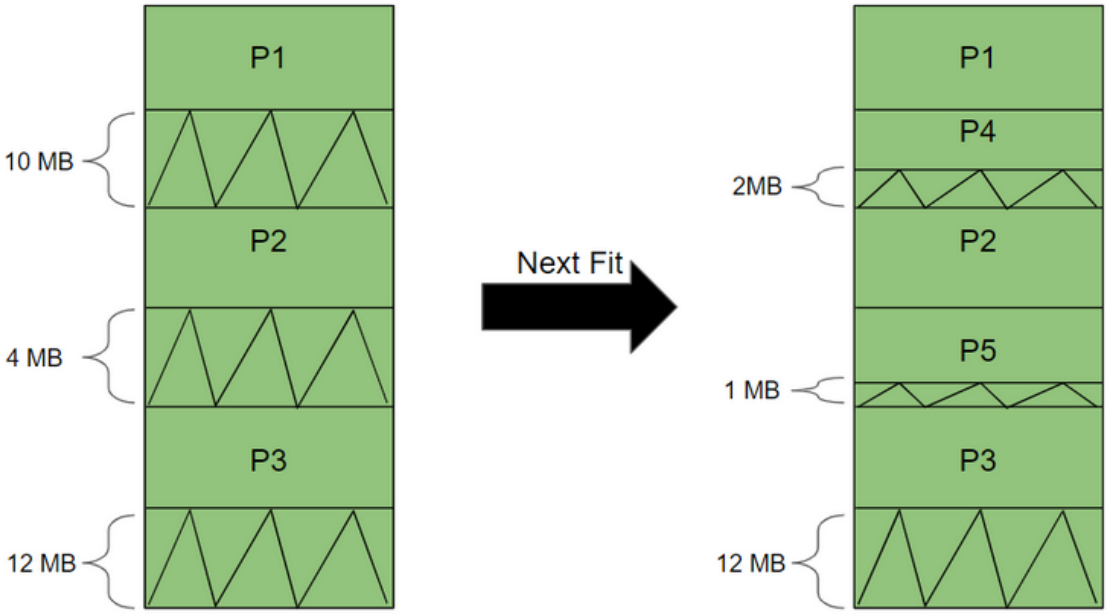
* for every new incoming process the whole linklist has to be traversed, thus it has greater time complexity

* various small holes are created which remains unused by other new process

3) Next fit:

it is similar to first fit

but it will search for the first sufficient partition from the last allocation point



first p4 comes and then p5 comes

4) worst fit :

allocate the process to the largest sufficient partition among the freely available partitons in the main memory.

NOTE :

after many anlysis , it is found that

* first fit is the best approach
* since best-fit algorithm needs to traverse the whole list every time a request is made

Paging in OS

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory
this permits the processes to be non-contiguous

Types of address and spaces :

1) logical / virtual address : created by CPU

2) logical / virtual address space : set of all logical addresses generated by a program

3) physical address (in bits) : an address actually available in memory unit

4) physical address space (in words or bytes) : set of all physical address corresponding to the logical address

NOTE :

* if adddress = n bits , then address space = 2**n word

* if address space is x words , then address = logx (base 2)

* 1M = 2**20
  1G = 2**30

Paging technique :

The mapping from the virtual to physical address is done by the memory management unit (MMU) which is a hardware device and this mapping is called as paging technique

Frames :

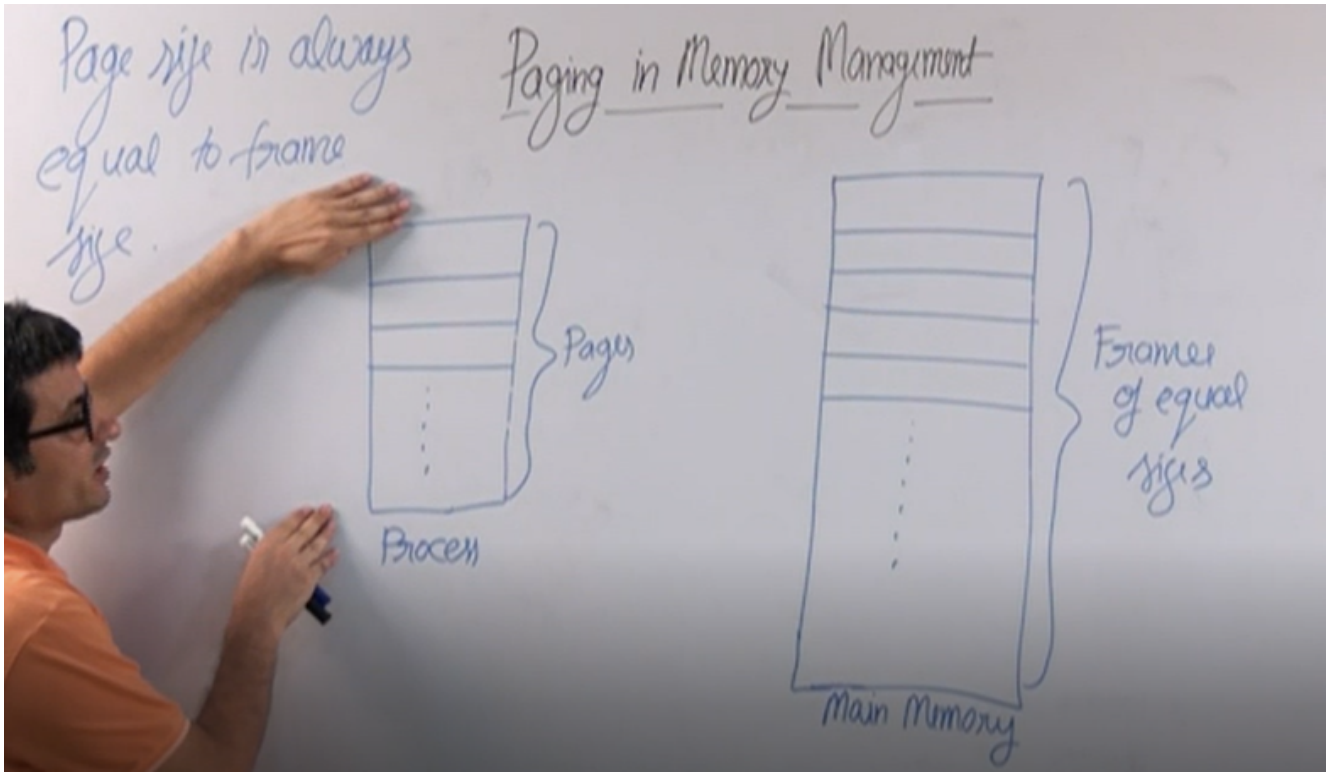
the physical address space is conceptually divided into a number of fixed-size blocks

Pages :

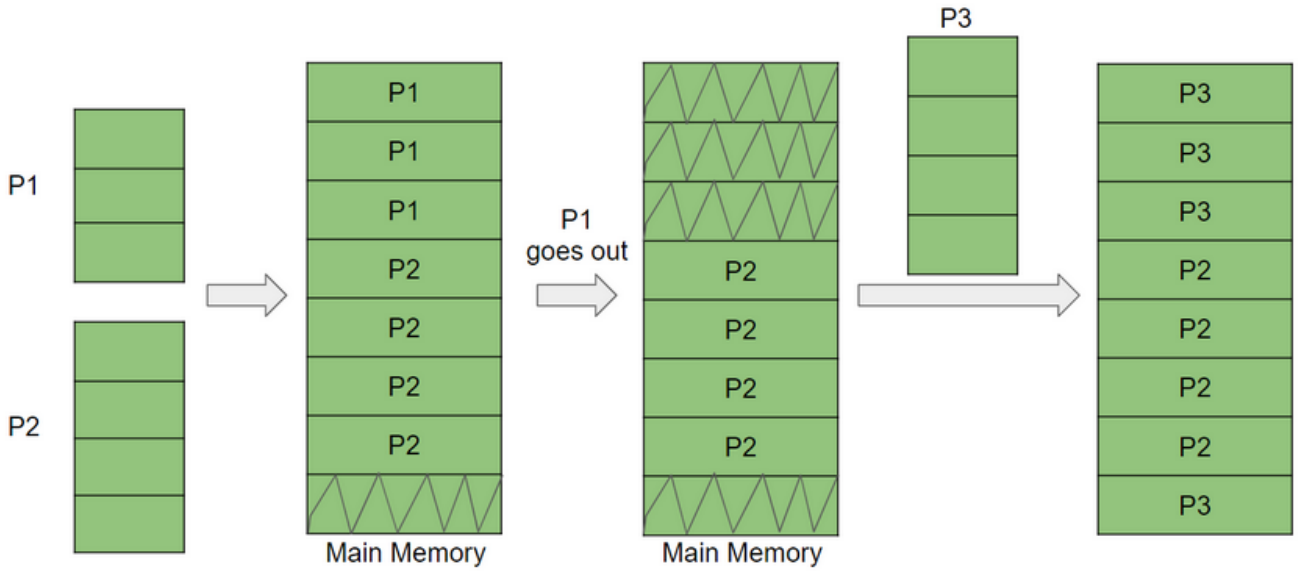
the logical address space is also split into fixed-size blocks

NOTE :

page size and frame size is always equal



example :



process p3 is allocated memory in non-contiguous way

runtime binding is used in paging to faciliate swapping during run time
for this we need logical address
these logical addresses are contiguous
but we need only non-contiguous memory
hence we convert this logical address into physical address
this bring us to the concept of Page Table

Page Table :

it stores the mapping of the logical address spaces to memory frames

to facilitate this task , the hardaware provides us with 2 registers:

1) Page table base register:

used to store the starting address of the created page table when context switching occurs

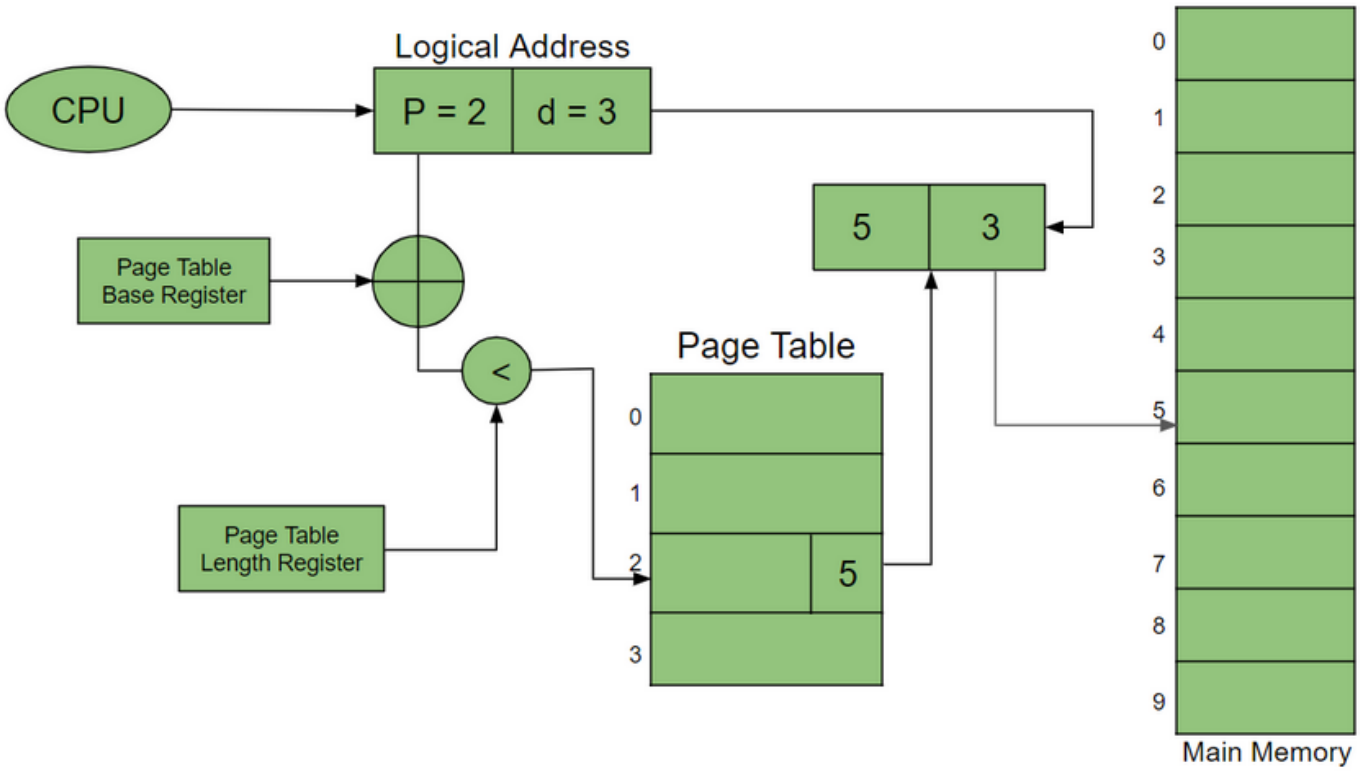
2) Page table length register :

used for security purposes

these registers helps during context switching

for every frame in main memory , we have corresponding page address in the page table

the below is the control flow diagram of paging :



in page table :

p - i th row (page num)

d - j th column (offset)

these page table contains the location of frames

logical address [p=2, d=3] = 5 , which is the location of the frame in main memory

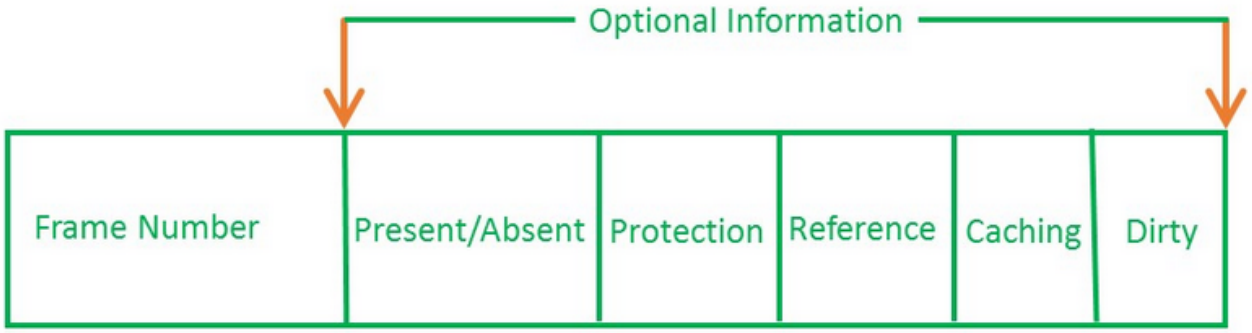
this is how the logical address is converted into physical address

Page Fault :

sometimes , the address in a page table points to a invalid location of main memory

this is called page fault

this occurs when a program attempts to access data or code that is in its address space but is not currently llocated in the system RAM.



PAGE TABLE ENTRY

num of bits for frame = size of physical memory / frame size

Dirty bit is also called as Modified bit

NOTE :

all the binaries , executables, are stored in the hard disk
frames are in main memory
the processes in hard disk is brought to main memory in the form of frames, where the pages are mapped to the frames
the pages , which are swapped out from the main memory due to some I/O processes , are also stored in hard disk

Segmentation

the chunks that a program is divided into which are not necessarily all of the same sizes are called segments
segmentation gives the user's view of the process that paging does not give
here the user's view is mapped to physical memory

there are two types :

1) virtual memory segmentation :
each process is divided into segments , not all of which are resident at any one

point of time

2) simple segmentation :

each process is divided into segments, all of which are loaded into memory at run time , though not necessarily contiguous

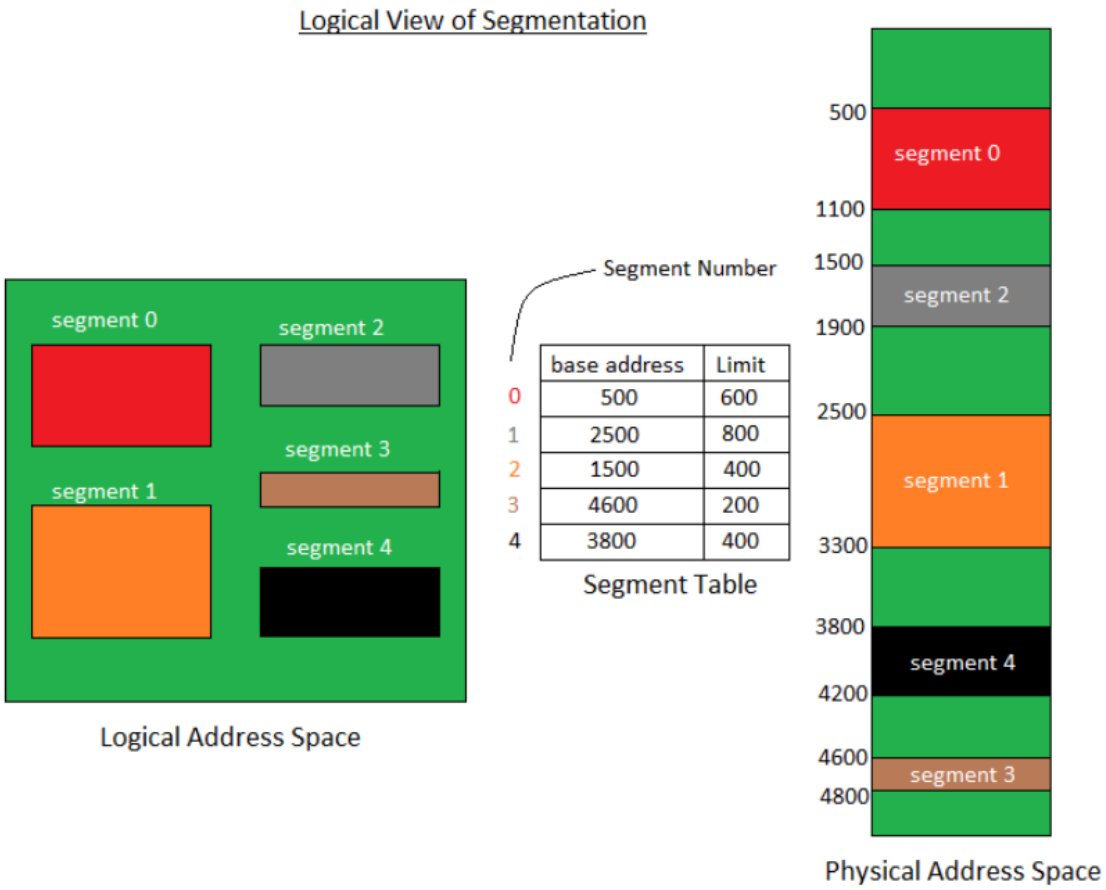
Segment Table :

a table stores the information about all such segments

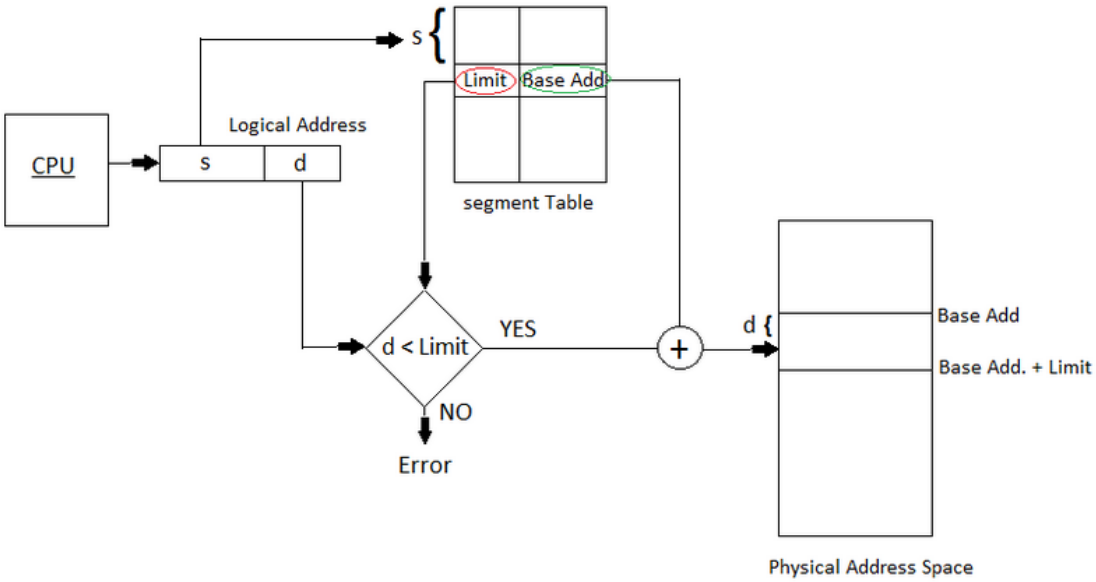
it maps 2D logical address into 1D physical address

each table entry has :

* base address (starting physical address of the segment)
* limit (lenght of the segment)



Translation of Two-dimensional Logical Address to one-dimensional Physical Address.



the address generated by the cpu is divided into :

* segment numder : num of bits required to represen the segment
* segment offset : num of bits required to represent the size of the segment

Advantages of segmentation :

* no internal fragmentation
* segment table consumes less space than the page table

Disadvantages of segmentation :

* as processes are loaded and removed from the memory , the free memory space is broken into little pieces , causing external fragmentation

Swapping

when computer needs more physical memory , we swap some memory on hard disk
swap space is the space in hard disk which is substitute for physical memory
it is used as a virtual memory that contains the process memory image
this swapping process is a very slow process

swap space helps the computer's os in pretending that it has more ram than it actually has
it is also called as swap file

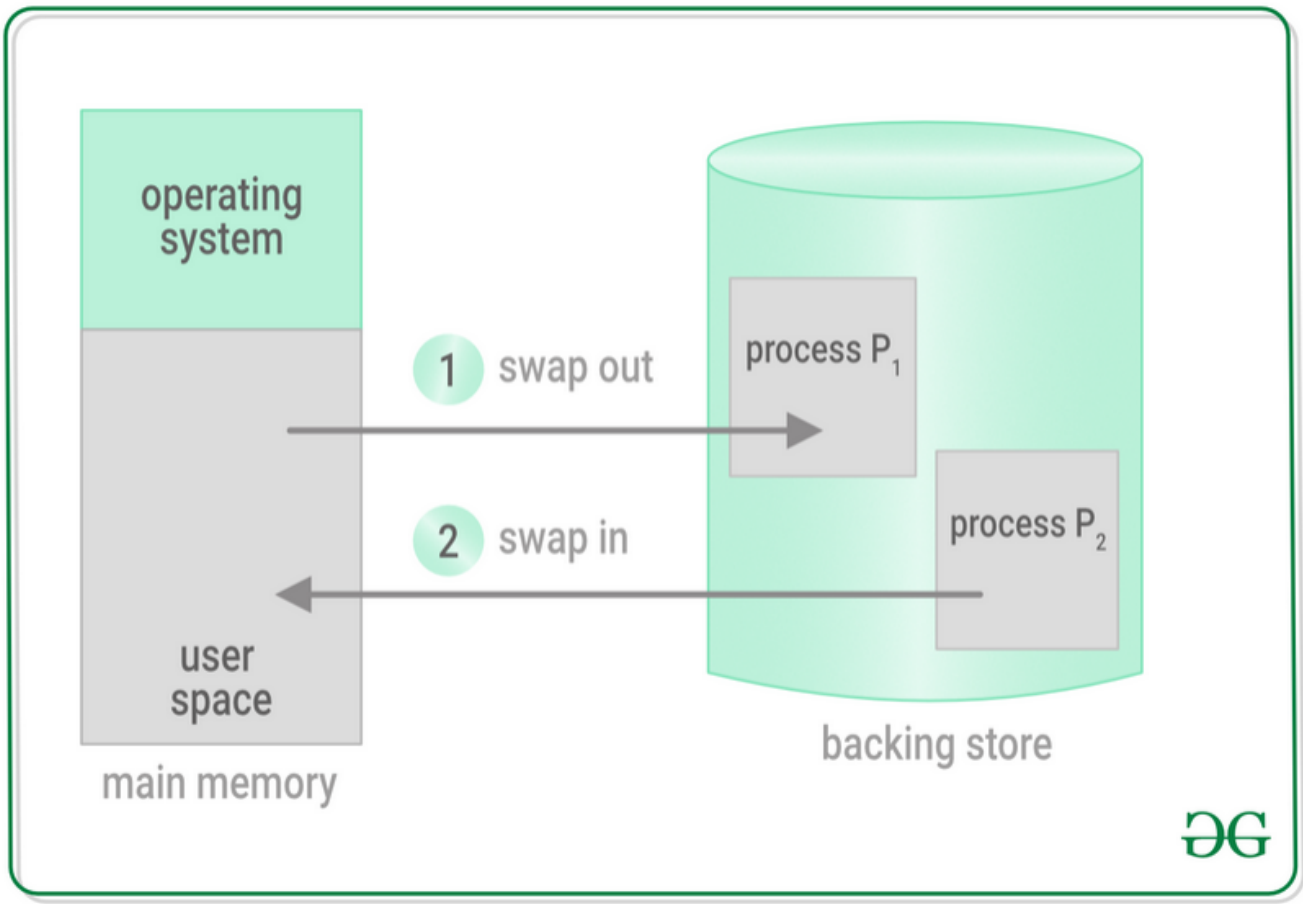
Uses of swap space to computer :

* can be used as single contiguous memory to reduce I/O operation to read or write a file
* less used or not used applications can be kept in swap file
* it helps to keep physical memory free all the time
* freed swap space in hard disk can be used by os to do some other important tasks

NOTE :

modern os doesnt use swapping, since it is a very slow process
they use faster alternative , i.e Paging

if needed (when system is extremely full) some unix system use swapping and later it will discontiue swapping



swapping in mobile systems :

on mobile devices , we have flash memory

so we dont have much space available
flash memory has limited number of times to write before it becomes unreliable
there is low bandwidth to flash memory

* apple's IOS :
it asks applications to voluntarily free up memory
os must uninstall the apps which are failed to free up memory

* android os :
it also follows a similar strategy
it writes application state to flash memory for quick restarting , instead to terminate a process

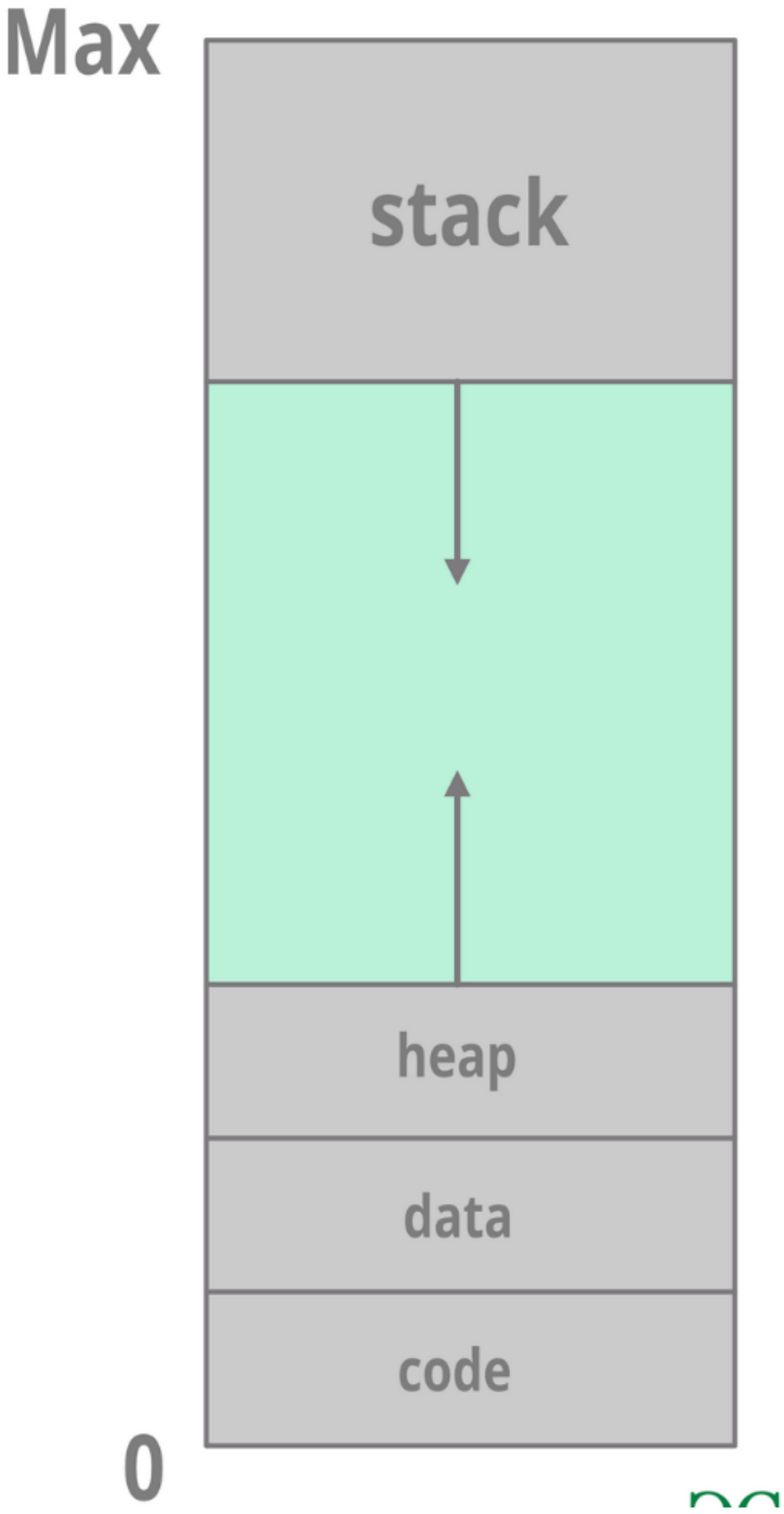
Virtual memory management

virtual memory is a storage allocation scheme in which secondary memory can be addressed as though it were a part of the main memory
the virtual memories are dynamically converted into physical address during the run time
virtual memory is implemented using Demand Paging or Demand Segmentation

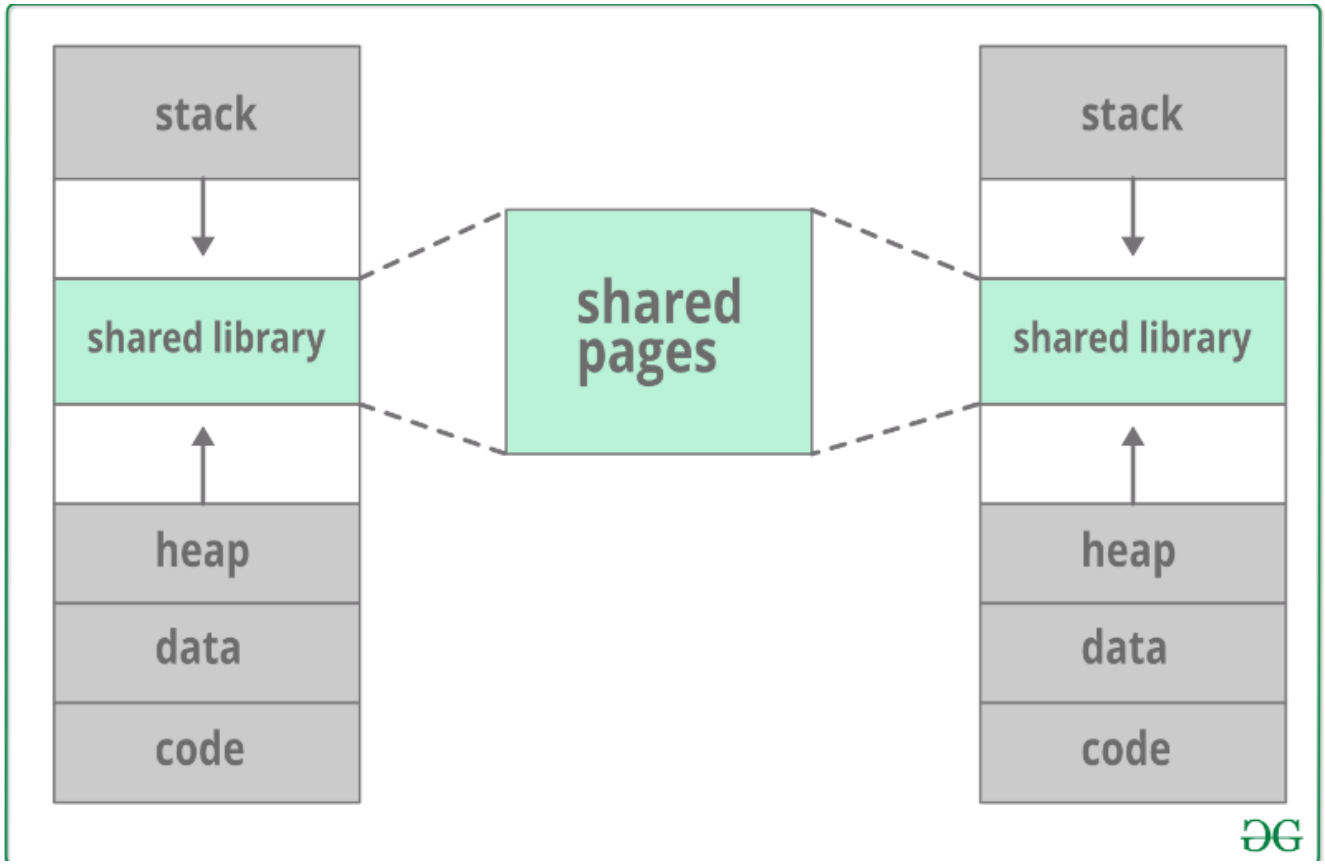
virtual memory management is a technique that is implemented using both hardware and sofrware
it maps memory addresses used by a program called virtual address , into physical addresses in computer memory

1) there is no need to load all the pages into the main memory
* error handling, routine to balance the federal budget are loaded only when needed
* advantage of this is :
:- programs can be wriiten in larger space in virtual memory than the physical memory
:- it improvise the cpu utilisation and system throughput
:- less I/O is needed for swapping processes , thus speed up things

2) the address space shown in blow figure is sparse
a great hole in the middle of the address space is never used unless the stack and / or the heap grow to fill the hole


Max
stack
heap
data
code
0

3) by mutiple processes , virtual memory also allows the sharing of files and memory
libaries, virtual memory , system call process pages can also be shared



Translation lookaside buffer (TLB)

in os , for each process , page table will be created , which will contain page table entry (PTE)
pte will contain the info like frame number and some othe useful bits

Now the problem is , where to place this PTE to get lesser overall access time

1) PTE stored in CPU registers
speed is good and overall things are better
but the problem is that , registers can store only pte of size 0.5K to 1k
when a pte of size 1M wants to be stored , registers cant store them

to overcome this PTE are stored in main memory

2) PTE in main memory :
this has overcame the size issies
but the problem is that . two main memory references are required:
1) to find the frame number
2) to go to the address specified by frame number

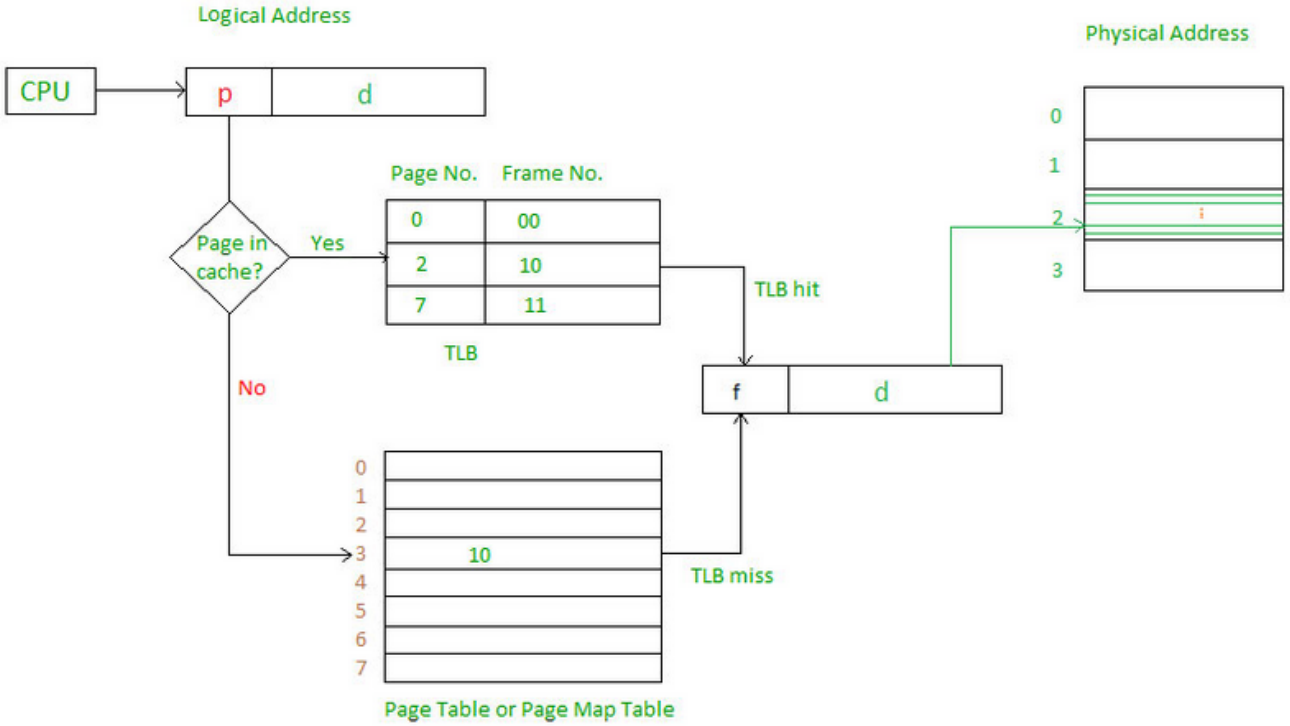
to overcome this, a high speed cache is set up for page table enteries called a Translation Lookaside Buffer (TLB)

Translation Lookaside Buffer (TLB) :

tlb is a cache used to keep track of recently used transactions
tlb has the pte that has been recenty used

TLB hit - pte is present in TLB
TLB miss - pte is absent in TLB

if pte is not found in the tlb (tlb miss) , then the page num is used to index the process page table.
tlb first checks if the page is already in the main memory ,
if not in the main memory , then a page fault is issued
then the tlb is updated to include the new page entry



Effective memory access time (EMAT) :

tlb is used to reduce emat as it is a high speed associative cache

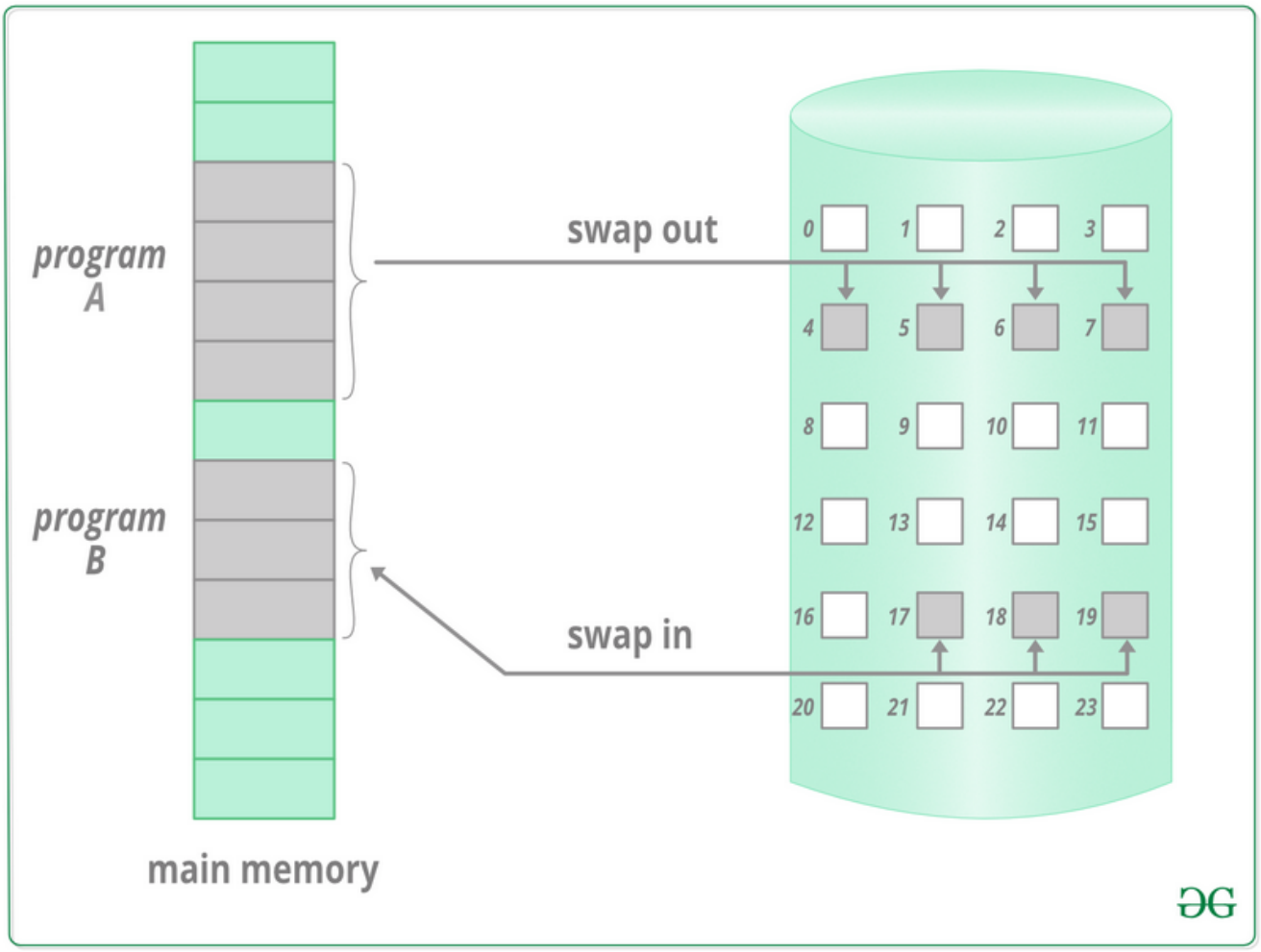
emat = h*(c=m) + (1-h)*(c+2m)

h - hit ratio of tlb
m - memory access time
c - tlb access time

Demand paging

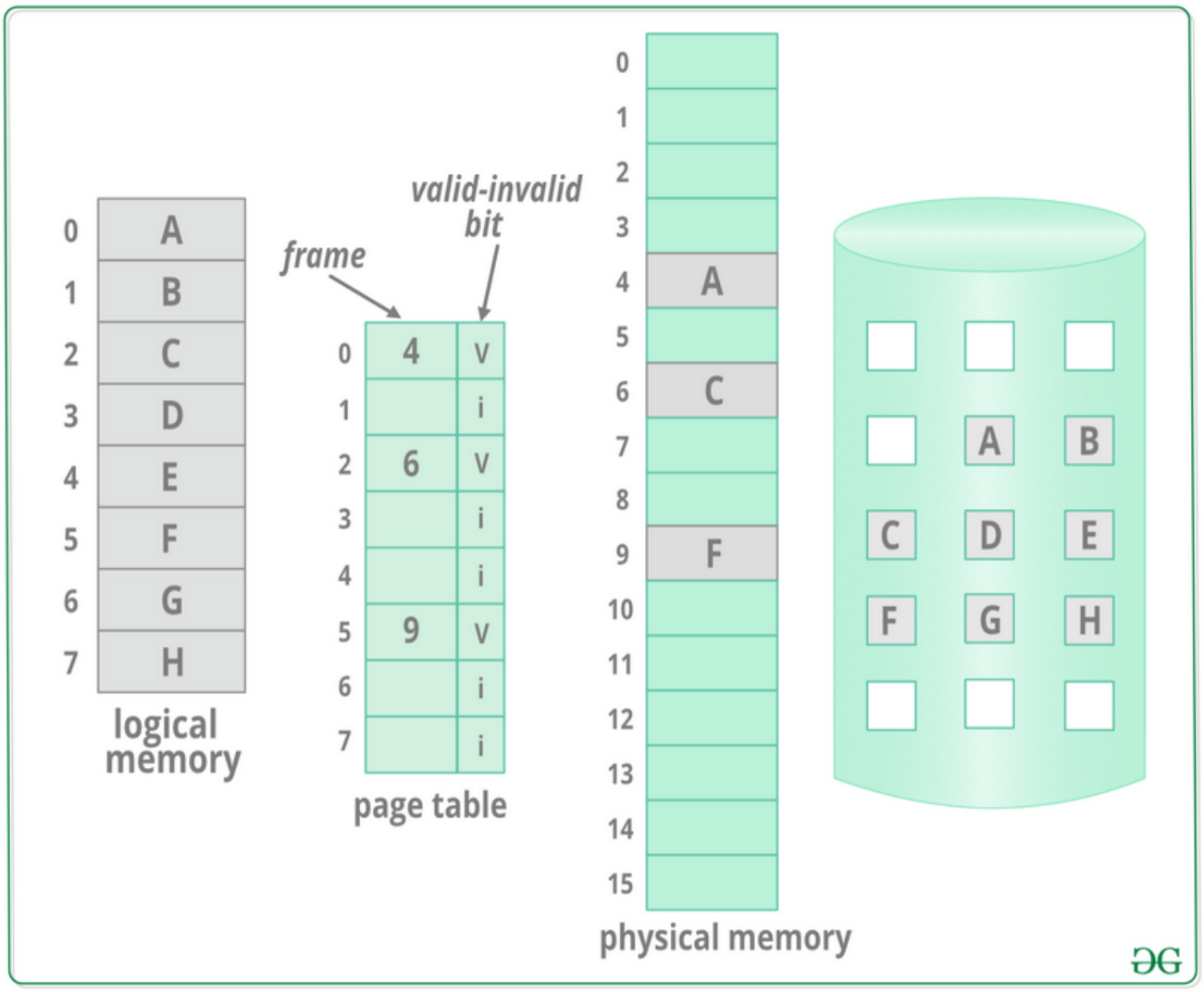
demang paging or lazy swapper:

when a process is swapped in , all its pages are not swapped in at once
rather they are swapped in only when the process demands them or needs them
this is the basic idea behind demand paging
this is also termed as lazy swapper



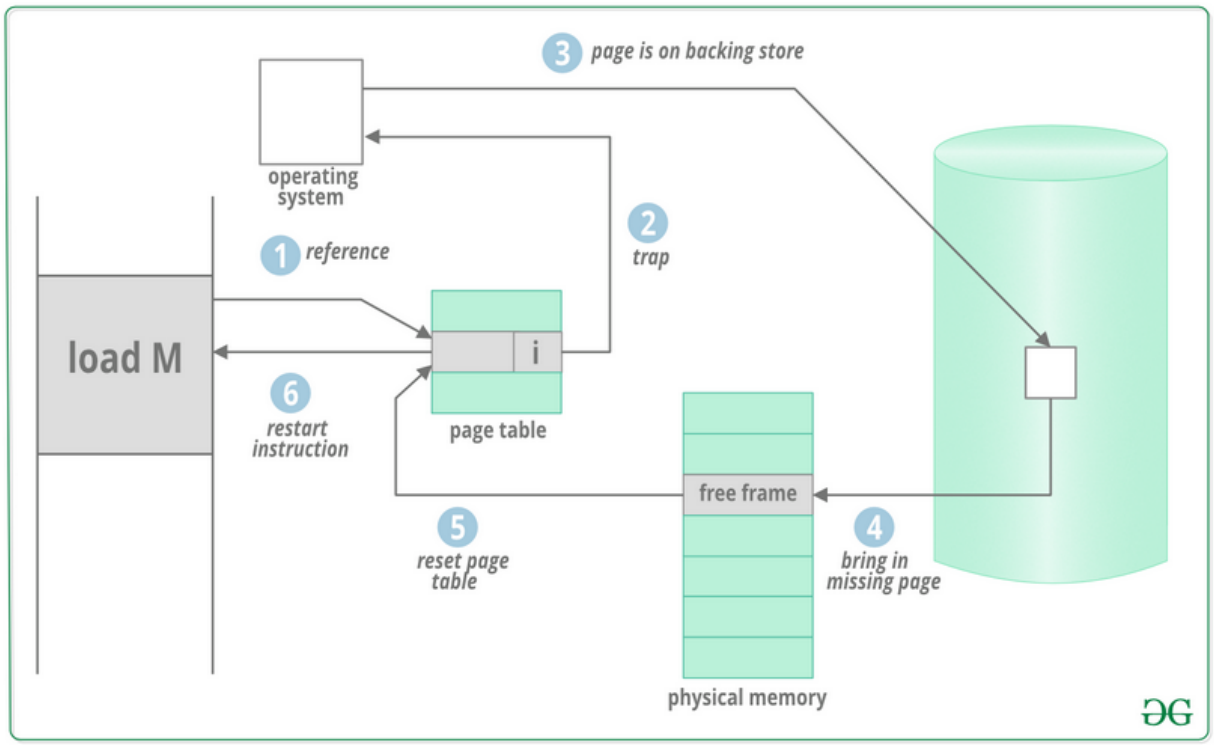
Basic concept :

* when a process is swapped in , the pager loads only the pages that are required right away into the memory
* the pages that are not loaded into the memory are marked as invalid pages table entry using a invalid bit
* if the process only ever access page that are loaded in memory , then the process runs as if all the pages were loaded into the memory



If a page is needed that was not originally loaded up, then a page fault is generated, which must be handled in a series of steps :

* the requested memory address is checked whether valid or not
* it not valid, the process is terminated
* it valid , then the page is paged in
* a free frame is allocated and to bring in the page a disk operation is scheduled
* it will allow some other process to use cpu and block the cpu process on I/O wait
* when the I/O operation in over , the process's page table is updated as valid
* the instruction will be restarted from the step that caused the page fault



NOTE :

once the demanded page is in the memory , the instructions must be restarted from the scratch

but in some architecture, a large block of data is modified by a single instruction
if these modifications occur b4 page fault, it causes some problems

solution :

access both ends of the block b4 executing the instruction so that the necessary pages already paged b4 the instructions begins

Performance of Demand Paging :

page fault causes some slowdown and performance hit

lets consider normal memory access needs 300 ns (nano seconds) and servicing a page fault takes 8 ms (mili sec) ie 9,000,000 ns

effective access time = (1-p) * (300) + p*9000000
= 300 + 8,999,700 *p # p - page fault rate

it purely depends on p

if 1 in 1000 causes a page fault, effective access time drops from 200ns to 8.2 ms (40% slowdown factor)

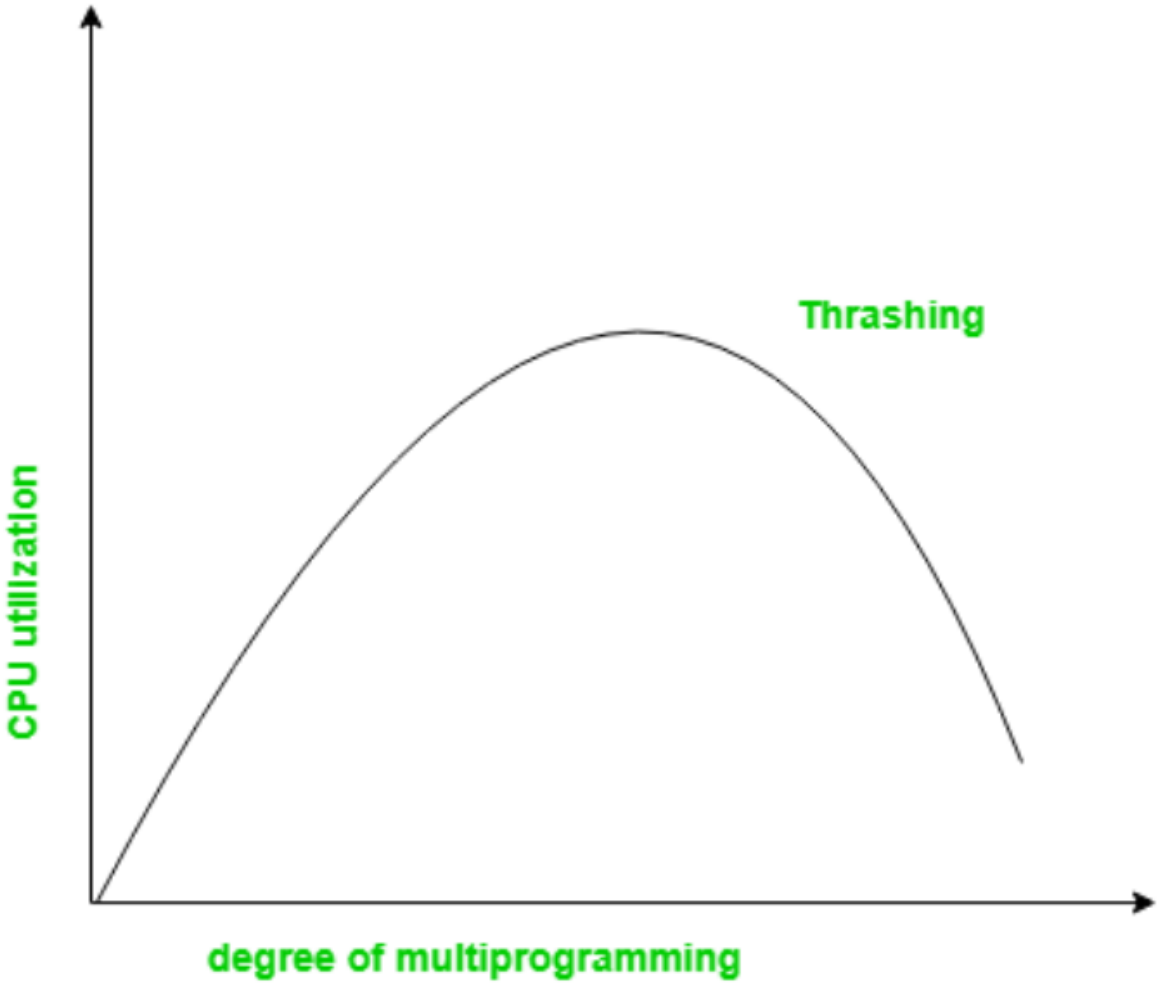
to keep slowdown less than 10%, page fault has to occur less than 1 in 399,990 accesses

Advantages of demand paging :

it increases the degree of multiprogramming
since many programms can be loaded into the memory

Thrashing

Thrashing is a condition in which the system is spending a major portion of its time in servicing the page faults, but the actual processing done is very neglible



when too many programms are loaded , the page fault occurs too frequently
thus making the cpu utilisation less efficient
this is called thrashing

if page fault is too high then it indicates that the process has too few frames
on contrary , a low page fault rate indicates that the process has too many frames

causes of thrashing :

* high degree of multiprogramming:
when more programs comes in , each process will have only limited frames

* Lacks frames :
if a process has less number of frames , then fewer pages of that process only can reside in memory , thus causing thrashing

Recovery of thrashing :

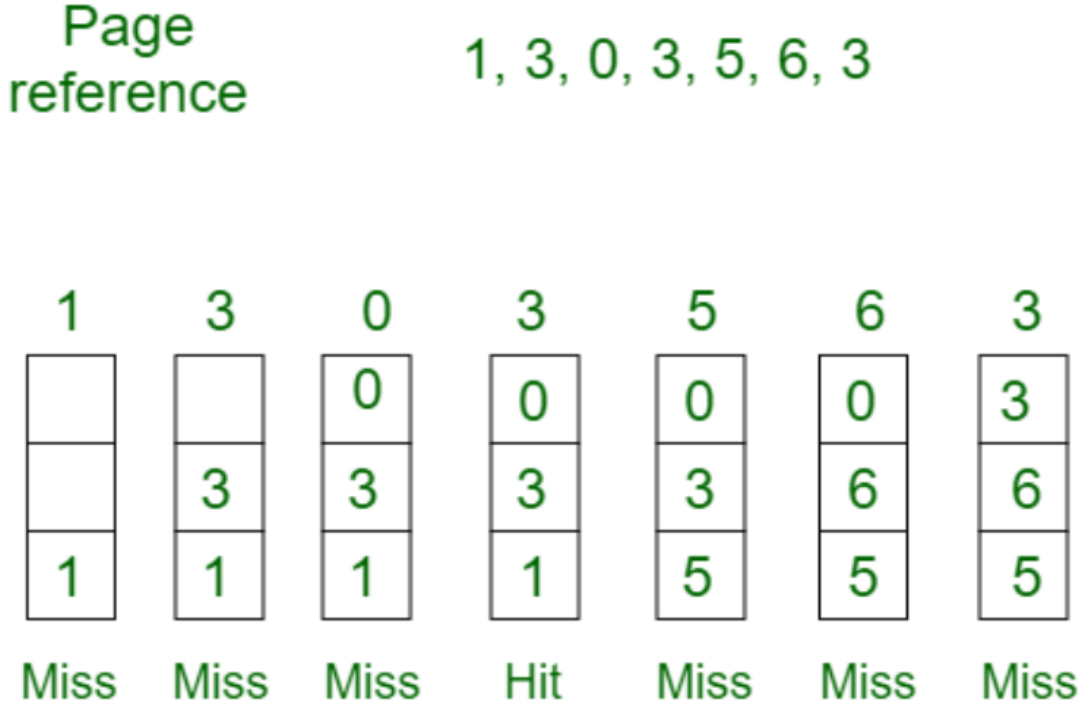
* if system is already in thrashing, then instruct the midterm scheduler to suspend from processes so that we can recover from the thrashing
* do not instruct long tern scheduler , not to bring the processes into memory after the threshold

Page replacement

Page algorithm decides which page need to be replaced in the main memory when a new page comes in
it is used when the main memory is full and a new page has to be swapped into the main memory

there are many algorithms for page replacement :

1) First In First Out (FIFO) :
simplest algo
the os keeps track of all pages in a queue
when a page needs to be replaced , the first page in the queue is replaced



first 1 comes , then 3 , then 0 - all are newly arrived
now again 3 comes - 3 is already present hence "hit"
then 5 comes , now first arrvied “1” is replaced
6 comes in , now the second arrived “3” is replaced
3 comes in , now the 3rd arrived “0” is replaced

Belady's anomaly :

the page fault may increase in some cases when increasing the number of page frames in FIFO

ex: consider refernence strings :

3,2,1,0,3,2,4,3,2,1,0,4 with 3 slots

we get 9 total page faults

if we increase the slot to 4 , we get 10 page faults

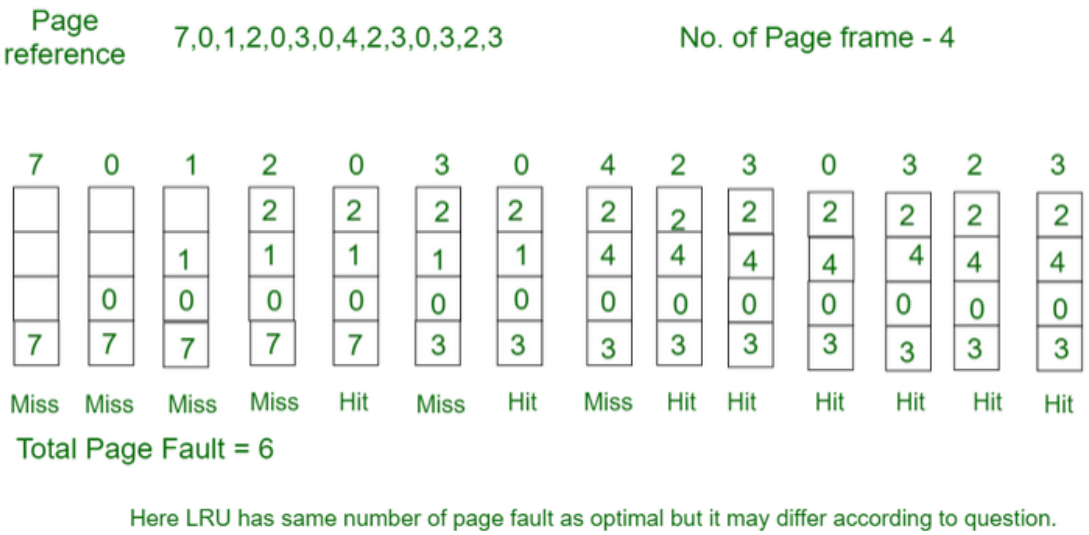
2) Optimal Page Replacement :

the pages are replaced which would not be used for the longest duration of time in the future

it is a ideal solution , but the problem is that we dont know which page will not be used in the future

3) Least Recently Used (LRU):

the page which is least recently used are replaced



Here LRU has same number of page fault as optimal but it may differ according to question.

Copy on write

the pages for a parent process do not have to be actually copied for the child process untill the page is changed by one or other of the processes
they can be shared b/w two procsses
the pages that can be modified also need to be labled as copy - on - write
code segment can be simply be shared
an alternative to the fork() system call is provided by some systems is called vfork() ie virtual memory fork
if parent is suspended , the child will use the parent's memory page. this is very fast but the child must not change any shared memory

Multi threading programming

Threads and its types

thread is a single sequence stream within a process
threads have same properties as of the process so they are called as light weight processes
they are executed one after another but gives the illusion as if they are executing in parallel
each thread has :

* a program conter
* a register set
* a stack space

Differences :

* threads are not independent of each other as they share the code, data, os , resources , etc. But processes can be independent
* threads are designed to assist each other, processes may or may not do it

Types of threads:

1) User level thread (ULT) :

* it is implemented in the user level libraary , they are not created using system calls
* thread switching need not call os and to cause an interrupt to kernel
* the kernel doesnt know about the user level thread and manages them as if they were single threaded processes

Advantages :

* can be implemented to os that doesnt support multythreading
* simple to create , since no intervention of kernel
* thread switching is fast , since no os call is needed

Disadvantages :

* no or less co ordination among threads and kernel
* if one thread caused page fault , the entire process blocks

2) Kernel level thread (KLT) :

* kernel knows and manages the threads
* instead of thread table in each process , the kernel itself has thread table (a master one) and keeps track of all the threads in the system
* kernel also maintains the traditional process table to keep track of the processes
* os kernel provides system call to create and manage threads

Advantages :

* kernel has full knowledge about the thread in the sysem, thus the scheduler may decide to give more time to processes having large num of threads
* gud for applications that frequently block

Disadvantages :

* slow and inefficent
* it requires thread control block so it is an overhead

NOTE :

* each ult has a process that keeps track of the thread using thread table
* each klt has thread table (TCB) as well as the process table (PCB)

Multithreading

a thread is a path which is followed during a program's execution
ex: a program cant read keystrokes while making drawings
these tasks cant be executed by the program at same time
this problem can be solved by multitasking

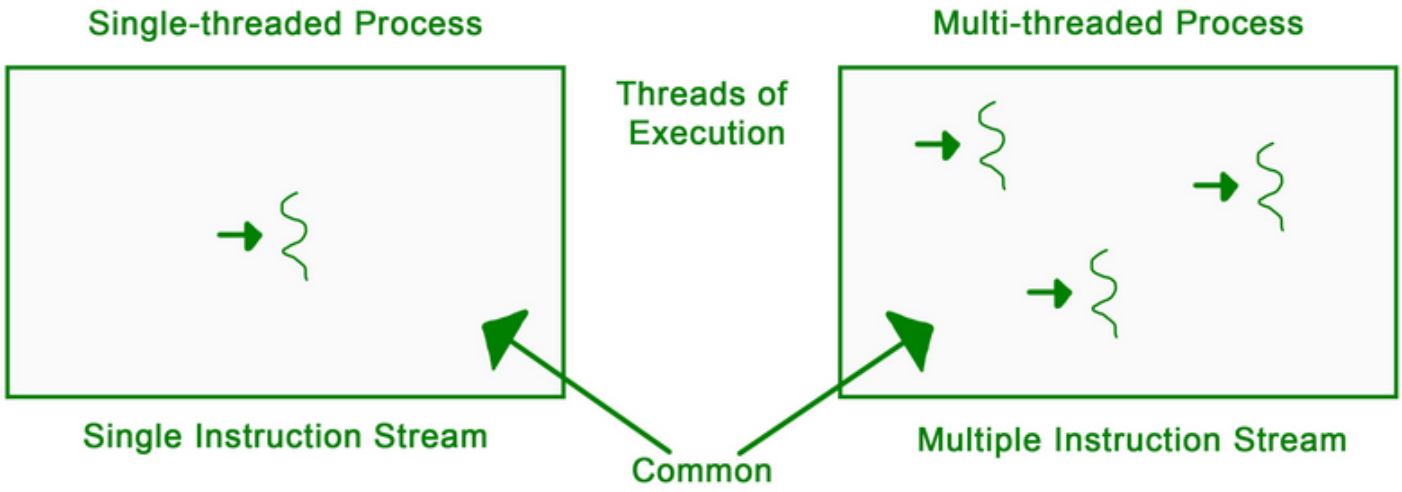
multitasking is of two types :

* processor based (totally managed by os)
* thread based

though the processor based multitasking is managed by os , multitasking through multithreading can be controlled by the programmers to some extend

NOTE :

process is a program being executed
processes are further divided into independent units called threads



Single Thread and Multi Thread Process

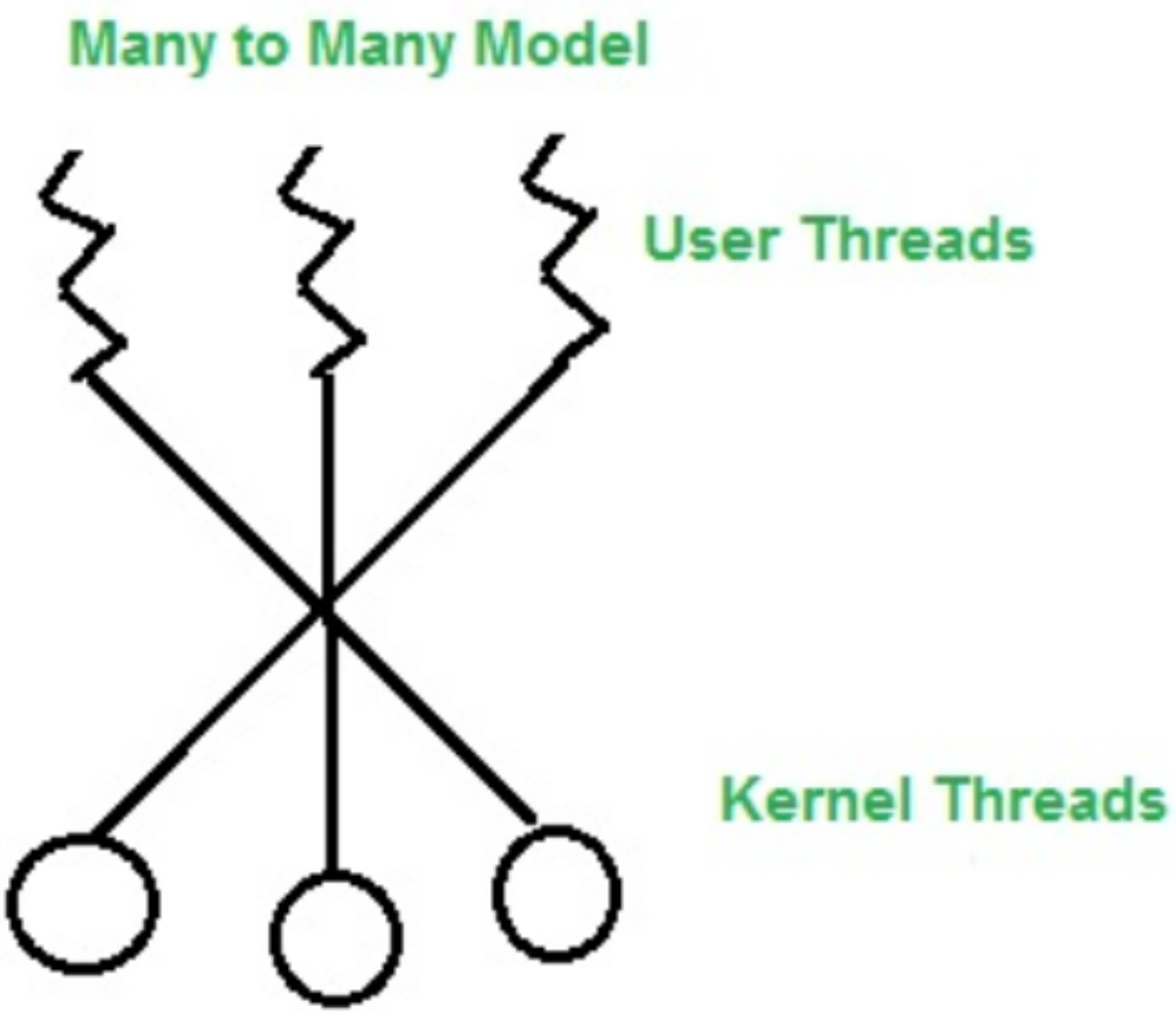
Many os supports kernel thread and user thread in a combined way (ex : solaris)

Multithreading is of 3 types :

* many to many models
* many to one model
* one to one model

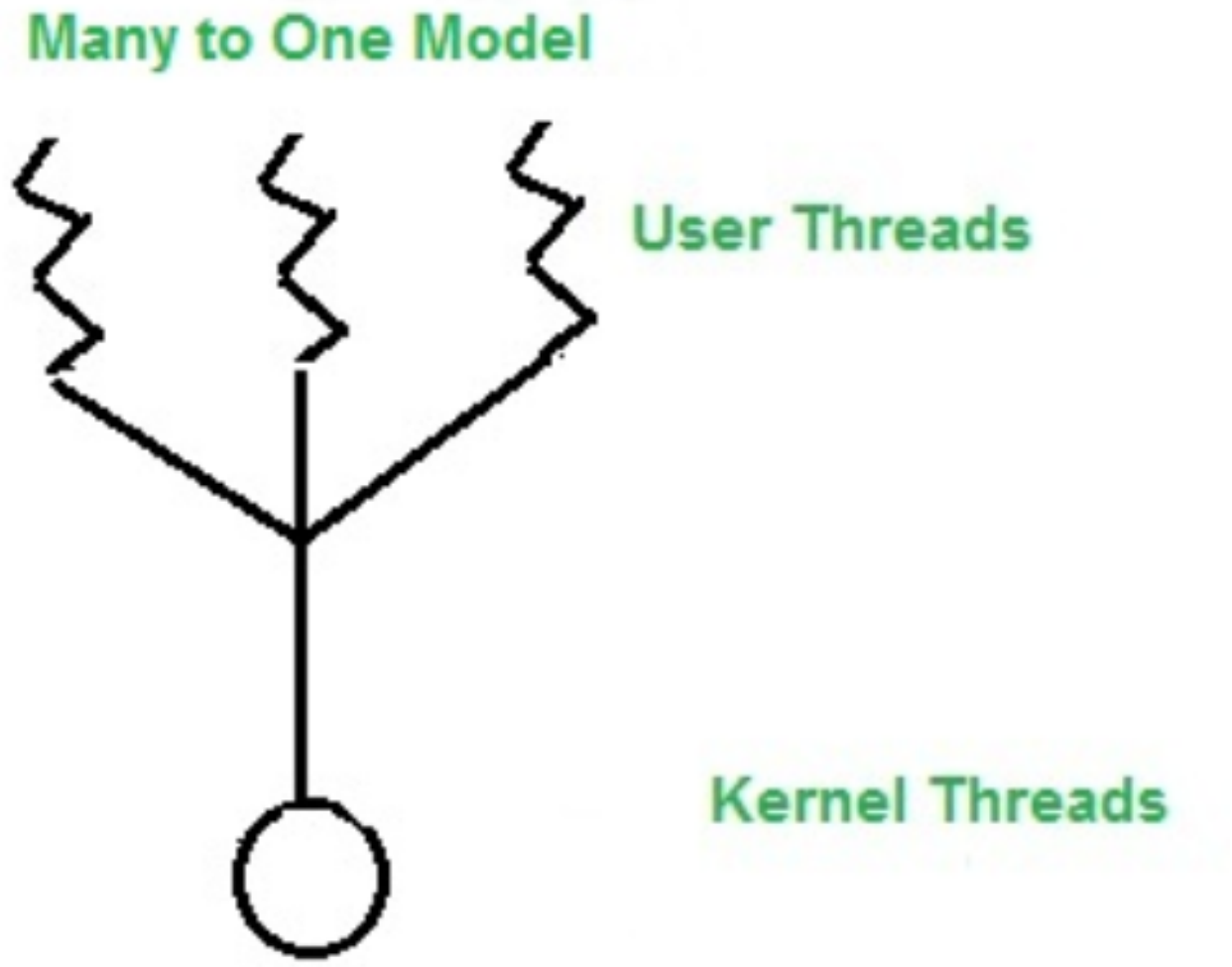
1) Many to many model :

we have multiple user threads multuplex to same or lesser number of kernel level threads.
if a user thread is blocked, we can schedule other user thread to other kernel thread
thus , system doesnt block if a particular thread is blocked



2) Many to one model :

we have multiple user thread mapped to one kernel thread
when a user thread makes a blocking system cal, entire process blocks
since we have only one kernel thread , only one user thread can access kernel at a time

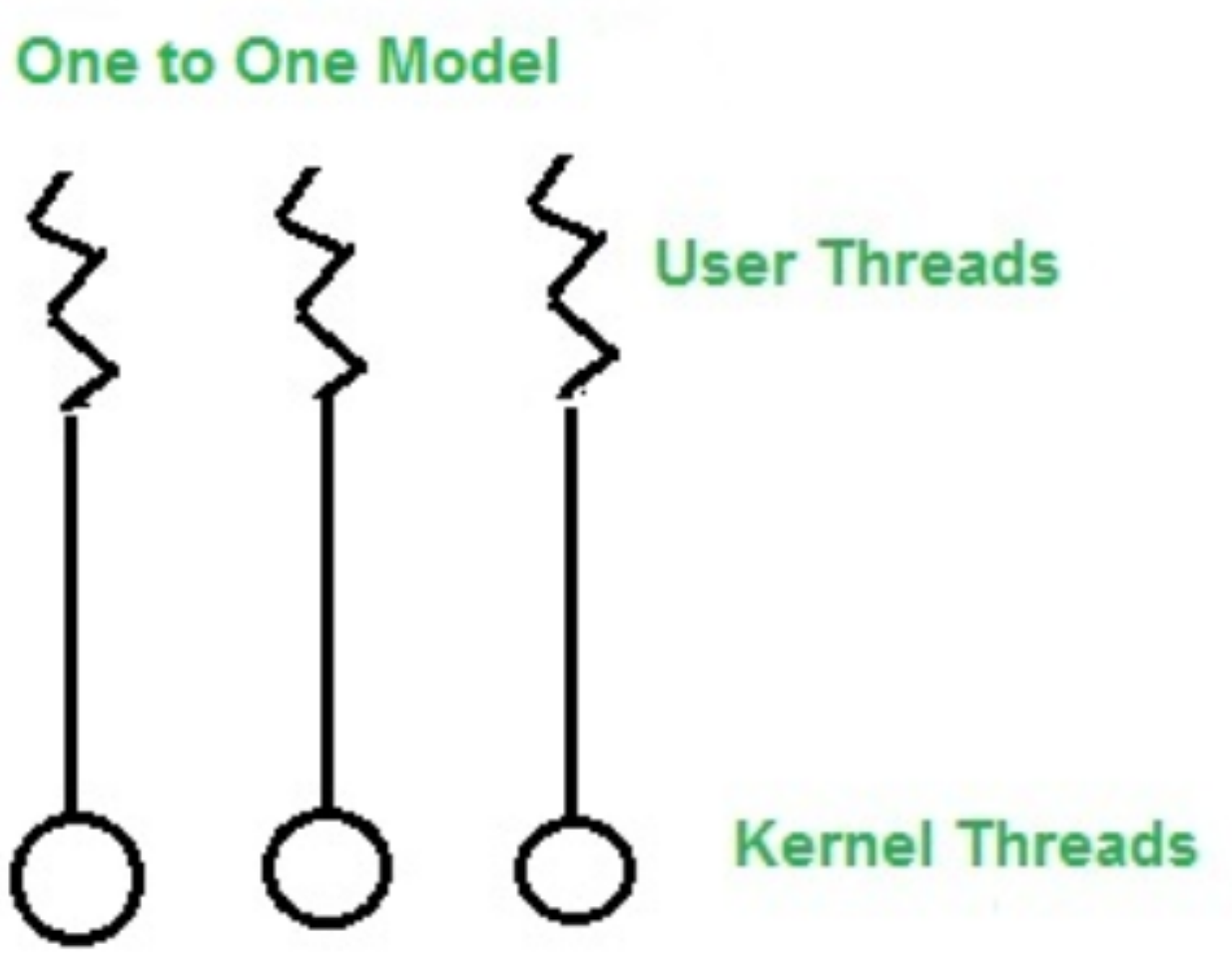


3) One to one model :

multiple threads can run on multiple processors

limitation:

creating a user thread requires the correspondin kernel thread



Benefits of multithreading :

* resource sharing
* responsiveness
* economy
* scalability

Thread libraries (Linux)

there are 3 types of threads libraries used :

* POSIX thread (pthread)
* window thread
* java thread

1) POSIX thread (pthread) :

POSIX standard defines the specification of thread but implementation is not there
solaris , mac osx, thr 64 thread and for windows it is available through public domain shareware
global threads are shared among all threads
one thread can wait for others to rejoin before contiuning
pthread begins executin in a specific function

2) Window thread :

it differs only in syntatic and nomenclature from POSIX windows

3) JAVA thread :

threads are used in all java programs even in common single threaded programms
creation of threads needs Objects that implement the runnable interface
java doesnt suppport global variables , thus a reference must be passed by threads to a shared object in order to share data

Thread scheduling

in multi processor scheduling multiple cpu's are available and hence load sharing becomes possible but it is more complex as compard to single processor scheduling

Approaches to multiple - processor scheduling :

1) Asymmetric multiprocessing (AMP):

all scheduling decisions and I/O processing are handled by a single processor called the master server and the other processors execute only the user code

this is simple and reduces the need for data sharing

2) Symmetric multiprocessing (SMP):

each processor is self scheduling,

all processes may be in a common ready queue or each processor may have its own private queue for ready processes

the scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute

Processor affinity :

process has an affinity for the processor on which it is currently running

when a process runs on a processor , the data recently accessed are populated in cache memory

thus when a process migrates from one process to another , the cache memory is invalidated for the first processor and the cache for second processor must be repopulated

cost of invalidation and repopulating caches are very high

thus SMP systems try to avoid migration of processes

There are two types of processor affinity :

* soft affinity :

an os has to policy to attempt to keep processes on the same processor, but not guranteeing to do so

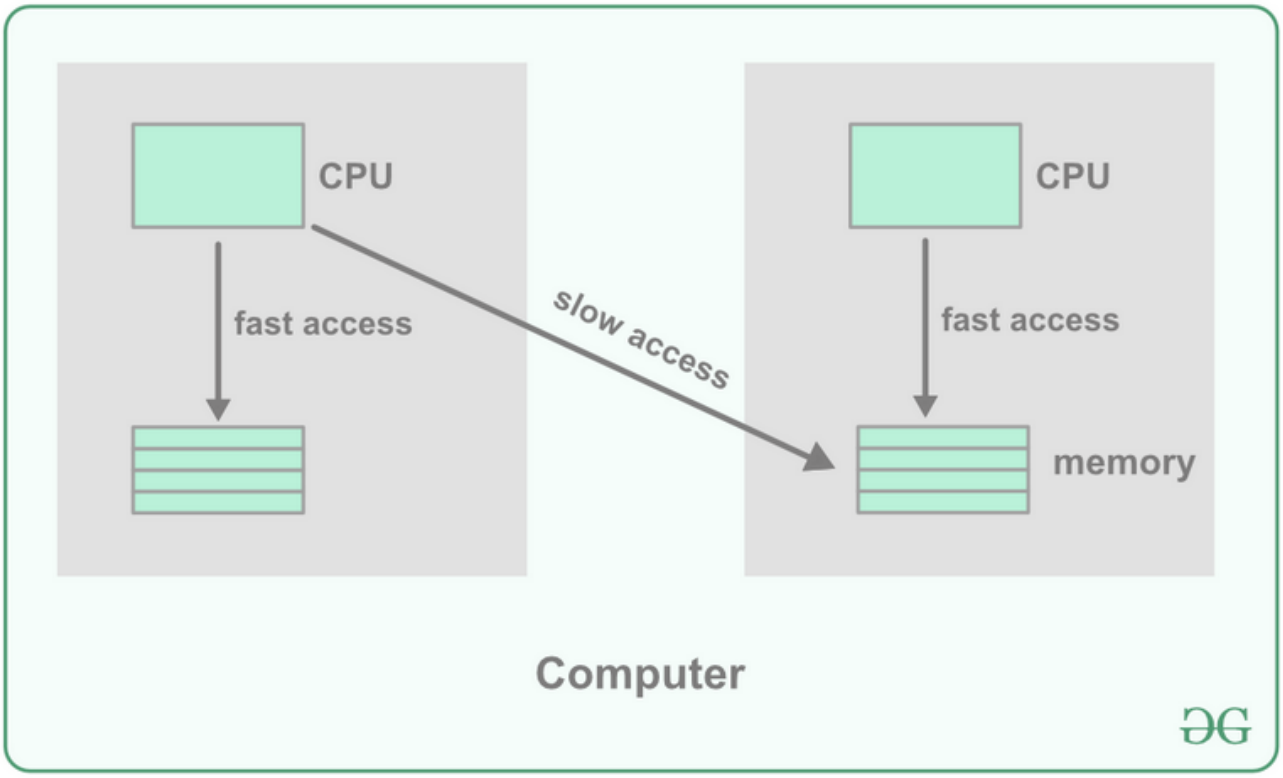
this situation is called soft affinity

* hard affinity :

systems like linux provide some system calls , which allows to migrate process b/w processors

this situation is called hard affinity

the below pic shows non-uniform memory access (ie non uniform access speed)



Load balancing :

load balancing keeps the workload evenly distributed across all processors in an SMP system

it is necessary only in systems where each processor has its own private queue of the process which is eligible to execute. this utilizes the processors efficiently

There are two general approaches to load balancing :

* push migration :

a task routinely checks the load on each processor and if it finds an imbalance then it evenly distributes the load on each processor

* pull migration :

pulll migration occurs when an idle processor pulls a waiting task from a busy processor for its execution

Multicore Processors :

in multicore processors , multiple processor cores are placed on the same physical chip

each core has a register sets to maintain its architectural state and thus appears to the os as a separate physical processors

SMP systems that use multicore processors are fast and consume less power than systems in which each processor has its own physical chip

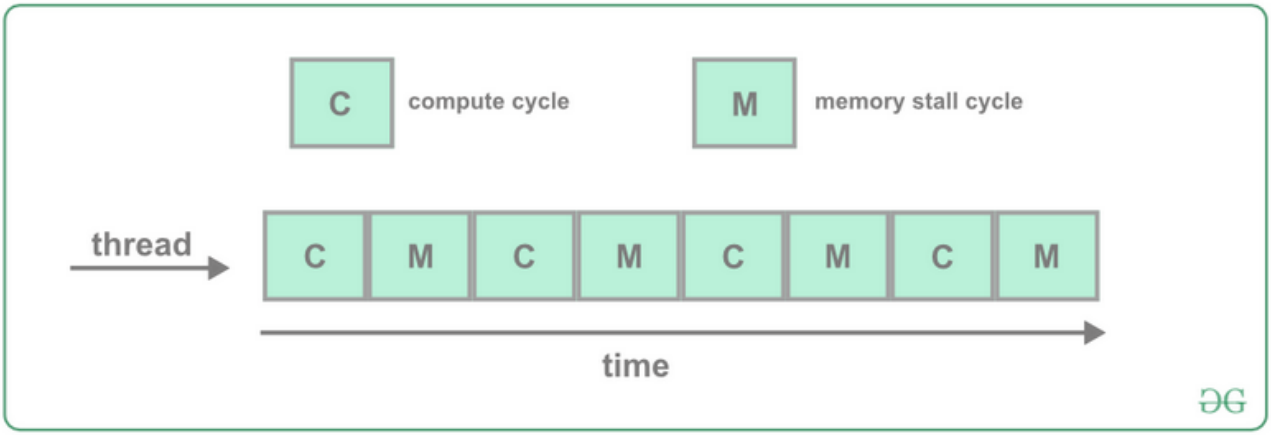
but multicore processors may complicate the scheduling problems

when a processor access memory , it has to wait till the data becomes available

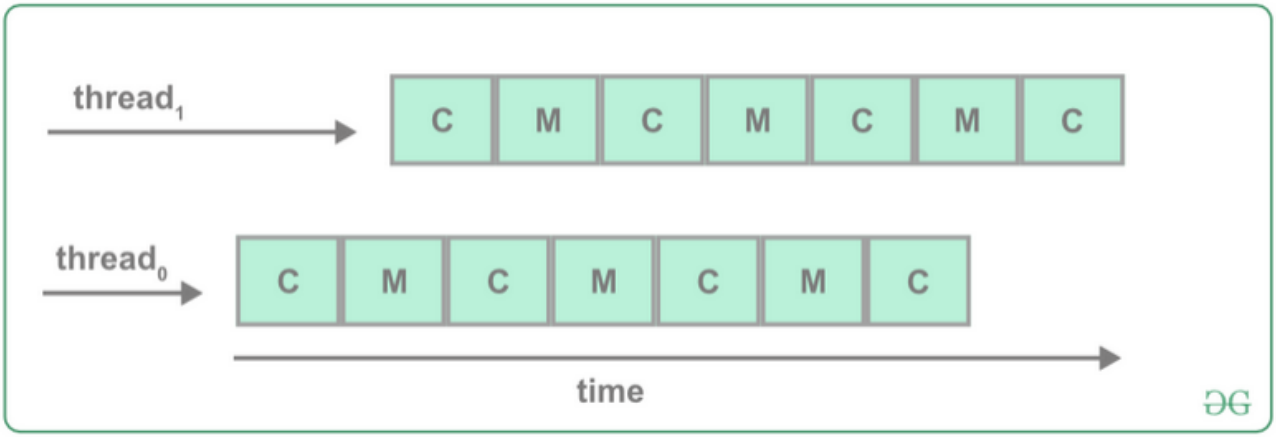
this situation is called memory stall
this may be due to cache miss
in this case , nearly 50% percent of processor's waiting time is wasted

to solve this, processors have implemented multi threaded processor core , in which two or more hardware threads are assigned to each core
therefore if one thread waits for memory access, the core can switch to another thread

The figure below is depicting the memory stall:



The figure below is depicting the multithread multicore system:



There are two ways to mulithreas a processor :

1) Coarse - Grained multi threading :
a thread executes on a processor until along latency event such as a memory stall occurs
because of long wait , the processor must switch to another thread to begin executing
the cost of switching b/w threads is high

2) Fine - Grained multi threading :
the multithreading switches b/w threads at a much finer level mainly at the boundary of an instruction cycle
it's architectural design includes logic for thread switching and as a result , the cost of switching b/w threads is small

Virtualization and multithreading :

depending on how busy the host system is, the time slice may take a second or more which results in very poor response time for users logged into that virtual machine
the individual vm receive only a portion of the available CPU cycles
commonly the time of day clock in vm is incorrect . because timers take no longer to trigger than they would on dedicated cpu's

virtualization can thus undo the good scheduling algo efforts of the os within the vms

Difference b/w multi threading and multi tasking :

* In multi-tasking , the system allows executing multiple programs and tasks at a same time. Whereas , in multi threading , the system executes multiple threads of the different or the same process at the same time

* Multi threading is more granular than multi tasking .
In multi tasking , cpu switches in real time , while in multi threading cpu switches b/w multiple threads of the same program

* Processes are heavyweight as compared to threads , they require their own address space
thus multitasking is heavier than multi threading

System calls

there are 2 modes of operation in an os:

* kernel mode
* user mode

1) Kernel mode :

critical os procedures run in kernel mode

only privileged instructions like :

* handling interrupts
* process management
* I/O managements are allowed to run

when an os boots , it begins execution in kernel mode only

all user applications which are loaded afterwards are run on user mode

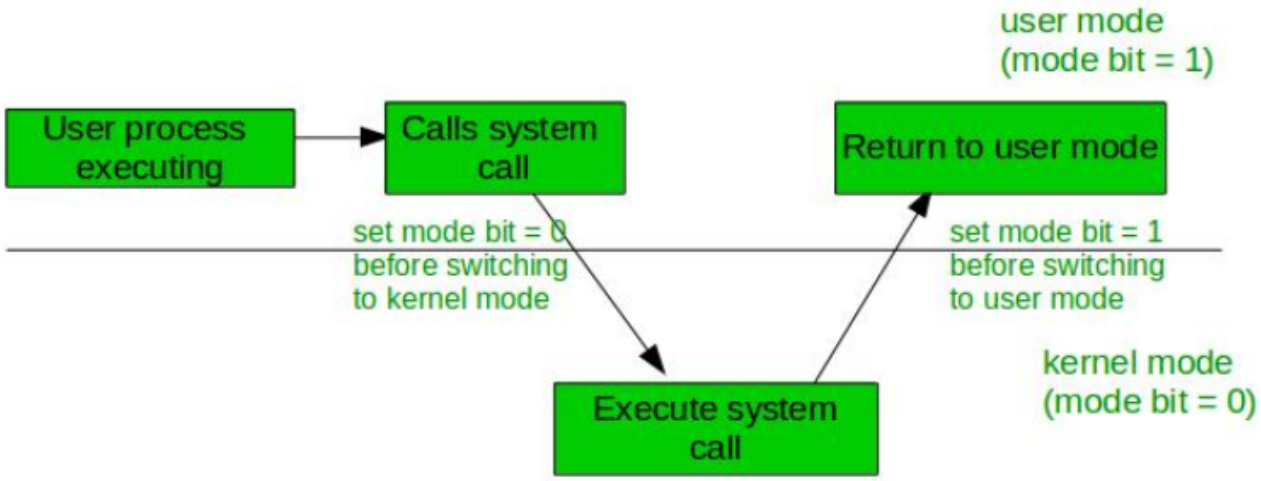
2) User mode :

all user applications and high level programs run in user mode

the dont mess up with critical os procedures

sometims , a user application may require access to low level utilities such as I/O peripheral, file system , etc

so os needs to switch to kernel mode
this is done by system calls



Basic system calls :

1) fork()

it creates a child process
it has a conditional return values:

<0 : unsuccessful child creation
0 : value returned to child process
>0 : value returned to parent process. It is equal to the PID (process Id) of the child process

the same code is executed in both processes
however , everything else (variable, stack , heap , etc) is duplicated

2) exec family :

fork creates a new child process,but has same code executing afterwards
to load a new program into the child process, we can use exec family calls
exec calls replaces the current code with new one
ex : execvp()

3) getpid() :

it returns the process Id (PID) of the currently executing process (program)

4) wait() and exit() :

wait is a blocking call , which prevents the parent process from exiting without properly reaping any child process
exit call teminates the current process. it takes the exit code as an argument , which is returned to the parent