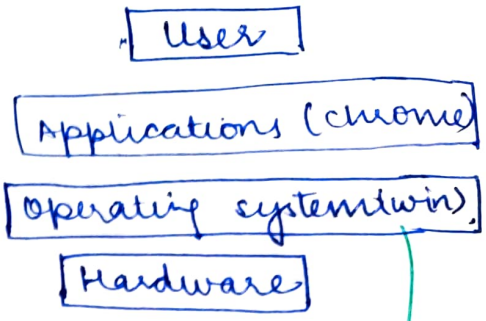
Operating System

- Applications have to interact with operating system to get things done.
- **Services :-**

Resource management
Abstraction
protection

- Abstracts the hardware using system calls
- allocates CPU's rams to different processes.
- One app doesn't impact the other.



windows
linux
macos
Android
ios.

Types of operating system

1) **Single tasking** → MS-DOS (Inefficient)
   - one process is there in a RAM & running

2) **Multiprogramming & Multitasking**
   - during computations process is assigned to the CPU
   - during I/O it's unassigned & assign some other process.

   multitasking is an extension of multiprogramming where process runs in time slots.

3) **Multithreading**
   - using different threads for different process
     - smallest unit of execution.

4) **Multiprocessing**
   - multiple processors ~~on the~~ at available

Multithreading

- Downloading something in browser and browsing
  - multiple things within a process.

multitasking

- listening to music & browsing web } two different processes running

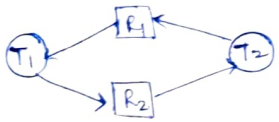
Examples :-

word processors → saving, typing, formatting, spell checking etc.

IDE's, games etc

Disadvantages of multithreading

- Difficulty in writing, testing.
- deadlock and race conditions
  same resource is used by two threads.

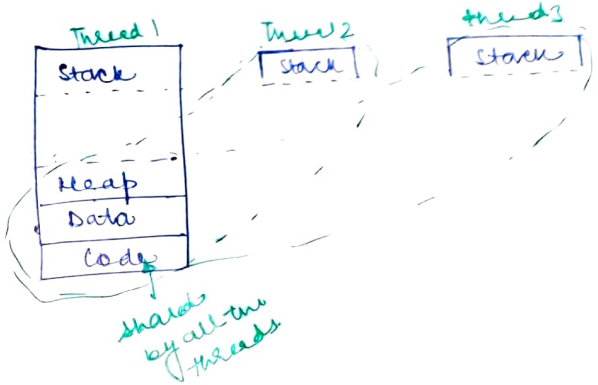


Deadlock.

Threads v/s processes

process → code, data, files, heap, stack etc

thread → stack



Threads are
- faster to create / terminate
- Multiple threads in a process
  - share address space
  - Easier to communicate
  - context switching is equal.
- Threads are lightweight
  - consume less resources.

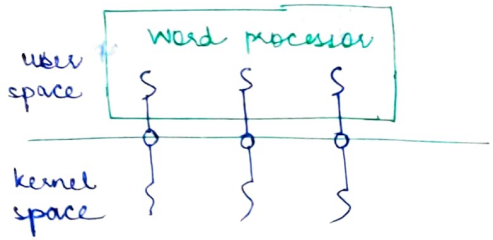
User Threads v/s Kernel Threads

- In user space
- Context switching faster
- one thread might block all other thread.
- Cannot take advantage of multicore systems.
- Creation is fast

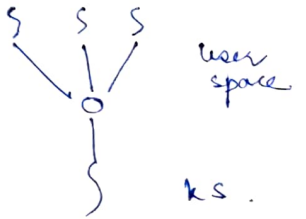
- in kernel space
- context switching slow
- A thread blocks itself only.
- Take full advantages of multicore systems.
- slow.

Mapping of user threads to kernel threads.

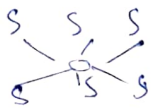
one to one



many to one



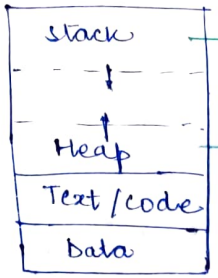
many to many



Program → set of instructions to perform a task

```
int main(){
    cout << "Hello";
}
```
→ compile → Binary program

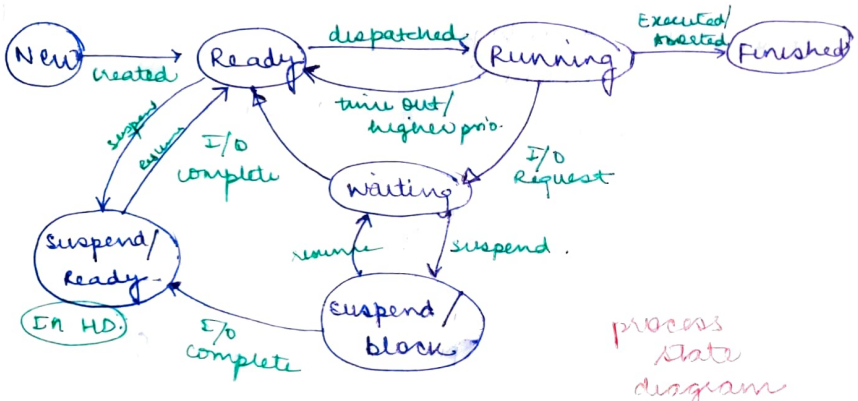
process



Stack → function calls & local fun variables

Heap → dynamically allocated variable

process states



process state diagram

Process Control Block

- OS has to store all the variables and state of process before stopping it

For this we use PCB Process control block.

1) process ID
2) process state
3) CPU registers
4) Accounts information
5) I/O information
6) CPU scheduling information
7) memory information

Process scheduler

1) Long term scheduler → which brings the process from disk to RAM. (To ready state)
2) short term scheduler → moves the process from ready state to running state (Dispatcher)
3) Medium term scheduler

The additional two states were managed by this suspend block & suspend ready.

Background of scheduling algorithm (short term scheduler)

→ Different queues
- ready queue
- job queue

→ Short term scheduler and dispatcher.
- pick one of the process from ready queue
- does context switch and maintains resumed state

→ when is a process picked by the scheduler.

(a) when some other process moves from running to waiting

(b) when some other process from ~~from~~ move from running to ready

(c) when a new / ~~exists~~ existing process move to ready

(d) when process terminates.

- Arrival time
- completion time
- Burst time — time of CPU that process takes
- Turn around time — comp. time − start time
- waiting time → time spent in ready queue
- Response time — diff b/w arrival time & first time it get CPU

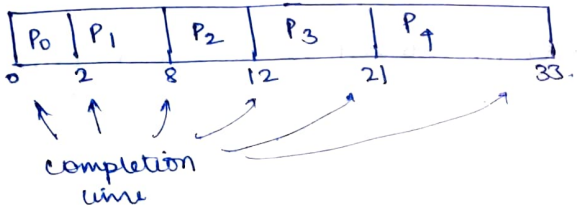
TAT − Burst time

Expectations from scheduling algorithm

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time
- Fair CPU allocation

FCFS scheduling algorithm (Non-preemptive)

process	Arrival time	Burst time
P_0	0	2
P_1	1	6
P_2	2	4
P_3	3	9
P_4	4	12



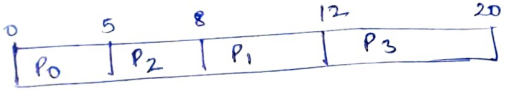
turnaround time → completion time − arrival time

waiting time → turnaround − burst

Convoy effect → one CPU bound process & many I/O bound process → create big waiting queues

Shortest Job First (SJF)

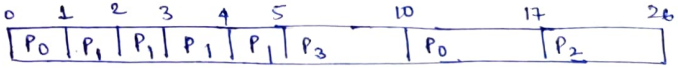
process	Arrival time	Burst time	comp. time	TAT	waiting
P_0	0	5	5	5	0
P_1	1	4	12	11	7
P_2	2	3	8	6	3
P_3	3	8	20	17	9



Non preemptive

Preemptive shortest Job First OR Shortest remaining time first

process	Arrival time	Burst time
P_0	0	~~8~~ 7
P_1	1	~~4~~ 3 ~~2~~ 1
P_2	2	9
P_3	3	5



* minimum average waiting time minimum
* May cause high waiting and response time for CPU bound jobs.

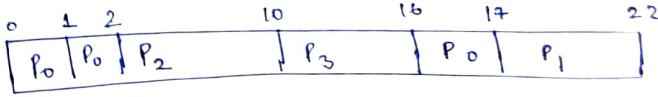
Priority Scheduling

process	Arrival	priority	Burst
P_0	0	5	3
P_1	1	3	5
P_2	2	15 (H)	8
P_3	3	12	6

(Non-preemptive)

PREEMPTIVE

process	arrival	priority	Burst
P_0	0	5	~~8~~ 1
P_1	1	3	5
P_2	2	15	~~8~~
P_3	3	12	~~6~~



preemptive.

Starvation of low priority

↳ ageing → increase the priority with their age

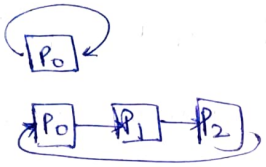
Round Robin Scheduling (Preemptive)

→ Time quantum

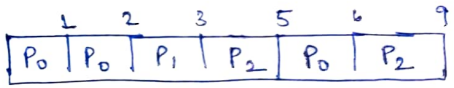
→ circular queue (Ready queue)

→ Avg waiting time can be higher but good response time.

process	Arrival	Burst
P_0	0	~~3~~ ~~0~~ 1
P_1	1	1
P_2	1	~~5~~ ~~3~~



time quantum = 2



Sensitive to time quantum

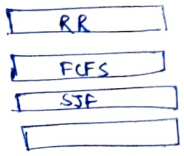
small
- context switch overhead

longer
- Become FCFS

Multilevel Queue Scheduling

↳ have different queues and apply different scheduling algorithms on them

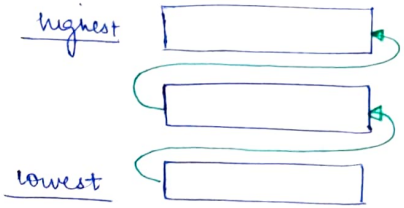
↳ we can have priority queues where CPU is divided among queues on the basis of priority



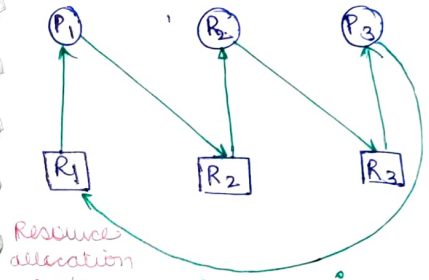
→ Starvation

Multilevel queue with feedback

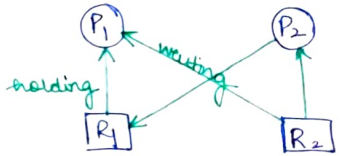
process should be allowed to switch between queues



DEADLOCK → happen when resources are non-sharable.



deadlock in 3 process

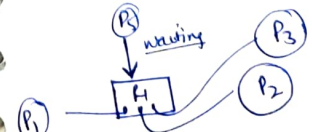


P_1 is holding an non-sharable resource and it waiting for other NS resource

necessary conditions

1) mutual exclusion → resources are non sharable
2) Hold and wait → processes must be in hold & wait i.e holding ~~an~~ a resource & waiting for other
3) circular wait → process are waiting in circular manner
4) No preemption → resources can't be taken back by OS in the middle of process

there might be multiple instances of a resource



Deadlock Prevention methods

1) **Deadlock prevention**
   prevent one of the four necessary condition for deadlock by assigning some rules. OS grant all the request but requests are sent in a way that they never cause deadlock.
2) **Deadlock avoidance**
   OS has to decidod whether a request can be granted or not.
   Ex- banker's Algorithm
3) **Detection and Recovery**
   OS periodically runs a job and check if dL has happened, if it has it is recovered by killing the process.
4) **Ignore the detection**
   ↳ simply ignore the deadlock.

~~Deddo~~ Deadlock Prevention

prevent / eleminate one of the following four

1) Mutual Exclusion
   - **Spooling**
     instead of sending a wait request processes are put in a job queue
2) **Hold and wait**
   - a process can tell in adv. what process it needs (impractical)
   - a process while requesting for a resource must release all the resources held by it
3) **NO preemptive resources**
   impractical as the process might be middle of execution and suddenly OS takes away the resource
4) **Circular wait**
   every process can take resources in increasing order (we number the resources)

Req. Deadlock Avoidance

	Allocated		Max	
	R_0	R_1	R_0	R_1
P_0	0	1	7	5
P_1	2	0	3	2
P_2	3	0	9	0
P_3	1	1	2	2
P_4	0	0	4	3

Total = $\underset{R_0\ R_1}{\langle 10, 5 \rangle}$

$P_3 \rightarrow$ 2 1 ⟸

Available → $\langle 3, 2 \rangle$

to check if safe state is there or not we generate a safe sequence

Consider a scenion where P_3 request for $P_3 \langle 1, 0 \rangle$ should OS grant this request?

Basic check

- total allocation is not more than maximum
- resources are available or not
  Ex → $\langle 4, 3 \rangle$ are available ✗N.
- OS assumes that resources are allocated and check that do y have safe state after allocation

Safe sequence → A sequence where the process will run one after another and all the processes will get resources.

P1 → Request made by Pi can still be satisfied by current available resources + resources held by previous processes.

If there exist even one safe sequence the process will never go in deadlock.

we need to generate need matrix first.

	Allocated		Max		Need	
P_0	0	1	7	5	7	4
P_1	2	0	3	2	1	2
P_2	3	0	9	0	6	0
P_3	2	1	2	2	0	1
P_4	0	0	4	3	4	3

Algorithm

```
safe_seq = { }
while ( All Ps are added ) {
    (a) find Pi such that need_i ≤ available
    (b) if (no such i exist)
            return false.
    else ( we found i ) {
            available += allocated_i
            Add Pi to the safe_seq.
    }.
}
return true;
```

Deadlock detection and Recovery

- Detection
  - all resources have single instances and there is a cycle in Resource allocation graph
  - If there is multiple instances we use Banker's algorithm (modified)
    - we do not include <0,0> while generating safe sequence
- Recovery
  - Kill process
  - Preempt Resource — Impractical

Process synchronization

process
- Independent
- cooperative
  - process may communicate with each other.

(P1)

```
void producer(){
while (true){
    while (count == SIZE) { ; }
    buffer[in] = produceItem();
    in = (in+1) % SIZE;
    count++;
}
```

```
int SIZE = 10
char buffer[SIZE];
int in = 0, out = 0;
int count = 0
```

global variables

(P2)

```
void consumer(){
while(true){
    while (count == 0){
        ;
    }.
    consumeItem(buffer[out])
    out = (out+1) % SIZE
    count--;
}.
```

producer → [ ] ← consumer

out in

(P1)

```
reg = count
reg = reg + 1;
count = reg
```

suppose the process is preempted by OS at this point

RACE cond^n

Execution

before P_1 gets back P_2 get the processor

```
reg = count
reg = reg - 1
count = reg
```

← It uses the wrong count value

Critical section → The section which mandatory have to run as a whole is put in this section

↓

part of program which tries to ~~share~~ access shared resources.

Synchronization mechanism

Disabling Interrupts → process disabling OS interrupts under critical section

- ✗ * allowing a user process to disable interrupts which cause serious issues
- * works only on single processor system

Locks (or Mutex)

- Acquire lock use the var and then release the lock
  - Hardware
  - software
    - Peterson
    - Bakery Algo

Semaphores

- WAIT, SIGNAL — extensions of lock
- Higher level than semaphore

Monitors

Lock for Synchronization

```
int amount = 100;
bool lock = false;
```

```
void deposit (int x) {
    while (lock == true) {
        // waiting
        ;
    }
    lock = true;
    amount = amount + x;
    lock = false;
}
```

```
void withdraw(x){
    while (lock == true)
        ;
    lock = true;
    amount = amount - x;
    lock = false;
}
```

If there is a preemption at this point then both the process will go in ~~mutual~~ to critical state and then mutual exclusion

```
void deposit (x) {
    while (test_and_set (lock)) {;}
    amount = amount + x;
    lock = false;
}
```

```
void withdraw(x) {
    while (test_and_set (lock)){;}
    amount = amount - x;
    lock = false;
}
```

ATOMIC

```
bool test_and_set (bool *ptr) {
    bool old = *ptr;
    *ptr = true;
    return old;
}
```

Semaphore

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

count → number of resources available.

wait()
signal()

wait() → decrease count (1 resource used)

signal() → increase count i.e see the queue if it has process it will assign the resource to process.

In the queue the semaphore stores the PCBs of process

- As any process approach and try to use the resource semaphore reduces the count and check if it's negative
  1) if negative, it stores the process info in queue
  2) if positive allocates the ~~pro~~resource
- If value of count is negative it'll show the number of person waiting in the queue

```
void wait () {
    s.count--;
    if (s.count < 0) {
        ① Add the caller process to q
        ② sleep(p);
    }
}
```

```
void signal () {
    s.count++;
    if (s.count <= 0) {
        ① Remove the process p from q
        ② wakeup(p);
    }
}
```

Binary Semaphore

the count variable is of type bool either 0 or 1.

```
struct BinSem {
    bool val;
    queue q;
}

void wait () {
    if (s.val == 1)
        s.val = 0;
    else {
        1) put this process in queue
        2) sleep (p);
    }
}
```

```
BinSem (true, empty);

void signal () {
    if (q is empty)
        s.val = 1;
    else {
        1) pick a process from q
        2) wakeup(p);
    }
}
```

Monitors → make a class which contains shared variables and functions which will use that are synchronized.

Memory Management in OS

- Memory Hierarchy → fast accessible
  → costing cost low
  → high capacity

CPU

- cache → fast accessible, low capacity
- Main memory → middle of both
- Secondary memory → ~~lowest~~ highest accessible time

Address Binding

↳ Binary of a program contains relocatable addresses.

Executable binary

001	- - -
002	- - -
003	- - -
004	goto 002
005	

- To run the binary file it must be brought in main memory and address in main memory is called physical add.

Address binding → mapping of relocatable address to physical add.

Compile Time

compiler know where the program is to be loaded in phy. mem.

Load time Binding

binary ex. files is mapped against phy. main memory.

Problems: once a prog. is loaded with n.m, it cannot be moved.

Eg: want to shutdown p. to disc while

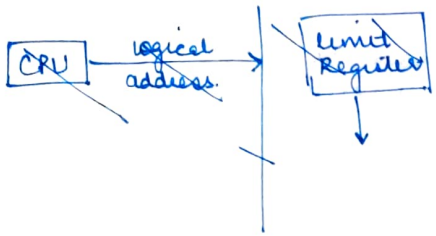
Run time

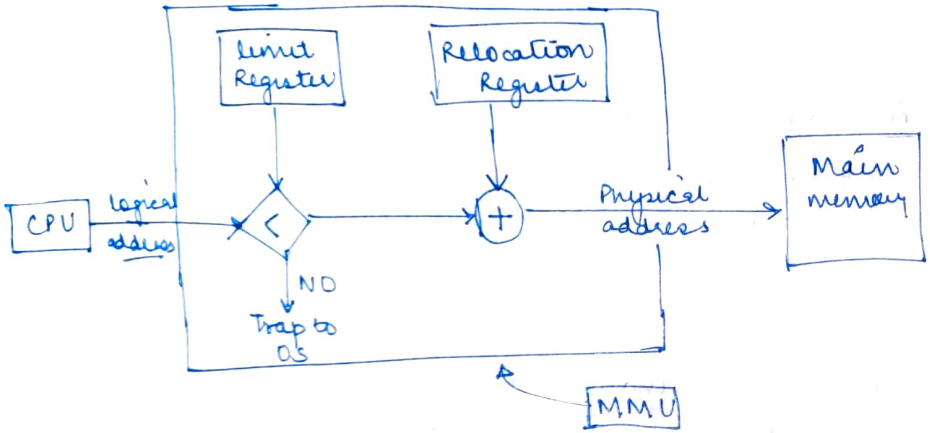
Process can be moved during run time.
→ address generated by CPU all not phy. address; instead logical address is generated.

MMU → convert LA to PA

Run time Binding

- CPU generates logical address.
- happens through hardware
- Hardware → MMU → memory management unit

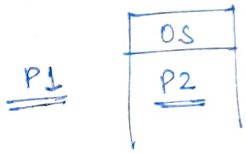




we can implement this using software too, but for every context switch OS has to run process (P) that converts logical A. to physical A. And doing this for every instruction is a costly step.

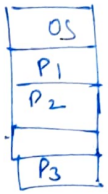
Evolution of memory management

1) Single tasking



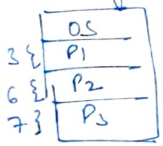
we want memory to hold more than 1 process at a time

2) Multitasking system



memory allocation

- Static
  - Equal size
    - memory loss is huge.
    - External Fragmentation
      - ↓
      - memory is sufficient to hold a process but divided in chunks.
  - Unequal size
    - Internal Fragmentation
- Dynamic
  - (i) contagious
    - allocation process the next av. memory



Problems

suppose P2 leaves the memory now the space left cannot be utilized by another process.

cause cured by ~~contraction~~ compaction

move all process to top

- (ii) Non-contagious
  - ↓
  - (a) paging
  - (b) segmentation
  - (c) paging with segmentations.

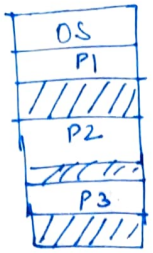
or

defragmentation

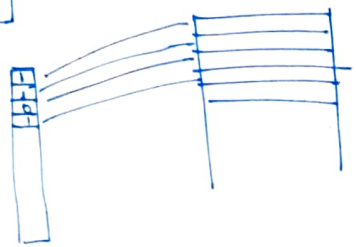
Reorganizing the data stored

Dynamic Partitioning

the holes crated by the leaving of processes has to be managed efficiently



1) Bitmap

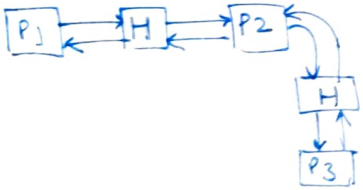


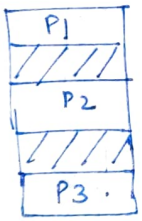
memory is divided into units and then mapped with the corrosponer block in bitmap

Requires lot of space ← Bitmap

2) Linked list

whole block is represented by a block of LL

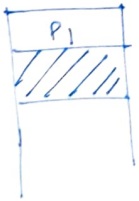




1) First fit
2) Best fit
3) Next fit
4) worst fit

travel whole list to get best fit, creates small holes

search for the first hole size that can accomodate the process.



Next fit → Begin from the place we stopped previously

Worst fit → Always allocate the biggest size slot.

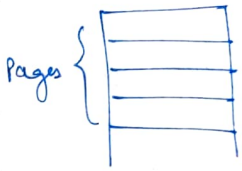
First fit → Best

Paging in memory management

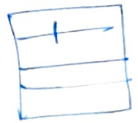
↦ internally program is stored in fragments

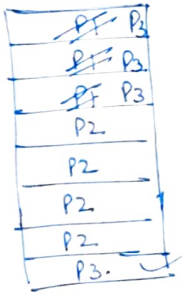
Main memory →

Frames of equal size

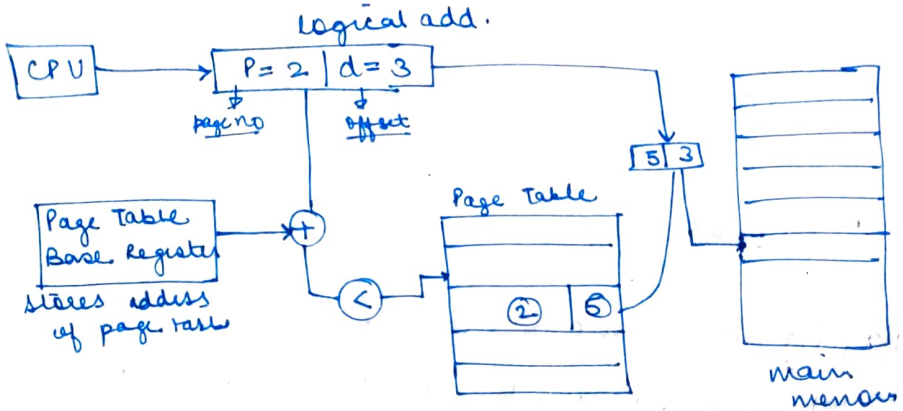


→ when you load a process in memory, its individual pages may be loaded at cont. locations or can not be





not stored contiguously



Page table stores the mapping of logical address pages to mem. frames.

page size

big → fragmentation may happen

small → very big page table

general chosen size → 4kb

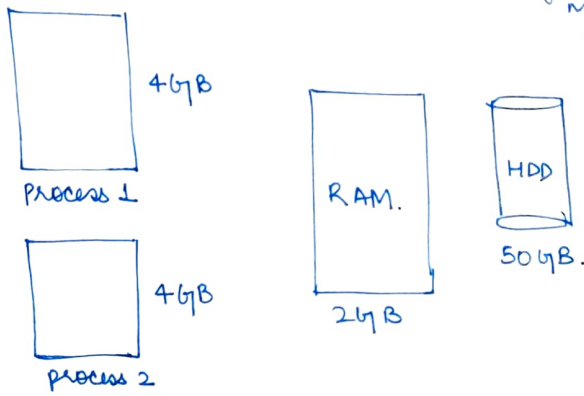
p = 2 | d = 3

↓

access page 2 and within page 2 go to offset 3

* All the pages may not be present in main mem.
  - ↳ Page fault
  - To keep track we have a bit called valid invalid bit which keeps track of the pages (stored or swapped out)
  - → That page has to be called from hard disk

Virtual Memory → some pages of process may be present in memory & some may not.



- Only load the required part of the process in the main memory and if ~~and~~ any required page is not found (page fault), then the page is brought into memory from hard disk

Page fault → costly operation
- ↳ mode switch
- reading data from HD
- ↓ Can increase the access time by a huge ratio.

pure demand paging → load the page from HDD when needed, and only load the required part of process in memory.
- ↳ Increases multitasking & programming.

locality of Reference
- ↳ load some nearby pages also.

TLB → Translational lookaside buffer

↳ CPU generates logical address.
a. LA has to be converted into PA
b. we have to look this PA into page table & this lookup is very frequent and we can optimize it using TLB.

Demand Paging

↳ only key working set of process in memory

thrashing

↳

thrashing is the condition when CPU utilization goes down due to high level of ~~pr~~ multi-programming / page faults

Page Replacement Algo

move existing page from main memory to accomodate the demanded page.

1) FIFO (suffer from Belady's Anomaly)
2) optimal
3) least Recently used.

↓ you remove the page which is going to use later in future (infeasible)

↓ replace the earliest used page in time

↓ If the no of frames is increased, no of page faults inc.

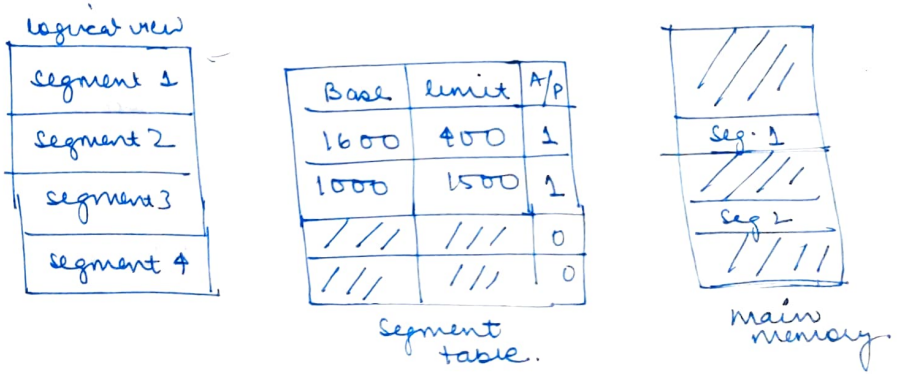
Segmentation in OS

simple / pure virtual

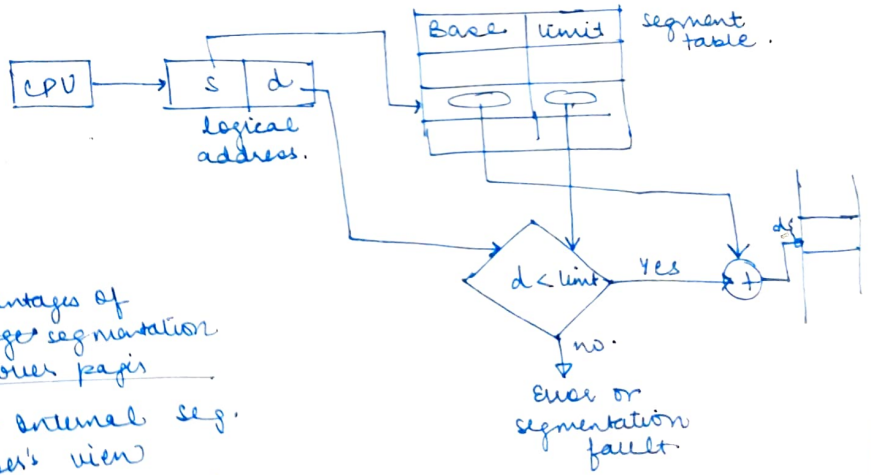
Alternative to paging

↳ brought the related ~~items~~ items together

we divided the process according to user's view, not necessary equal size



virtual memory → segmentation — All segment need not to be present in main memory.

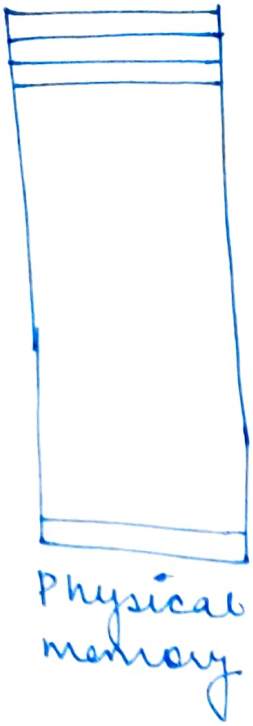


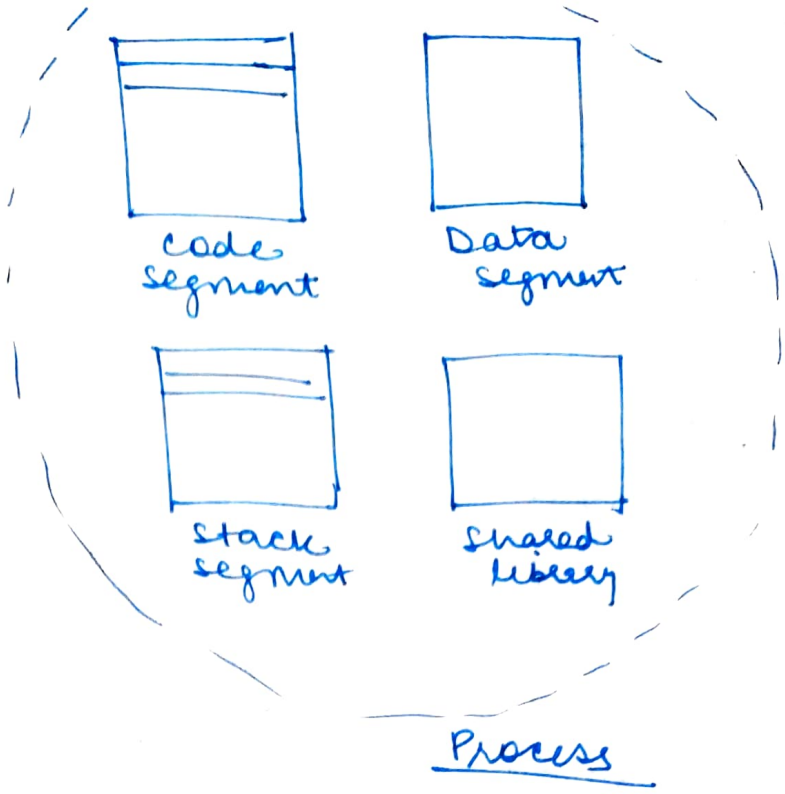
Advantages of page segmentation over pagis

→ No internal seg.
→ user's view
→ Segment table size is smaller than page table size

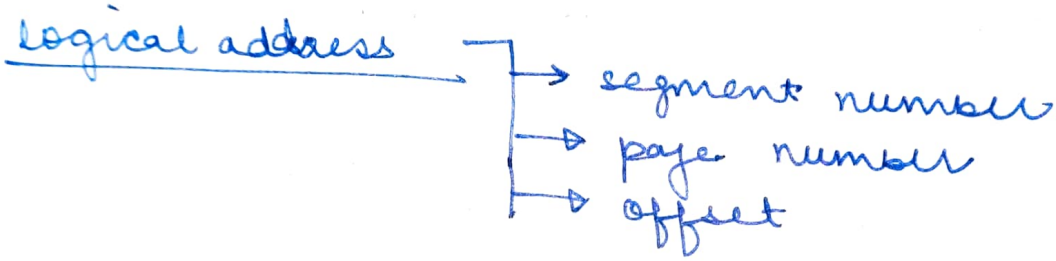
Disadvantages → External Fragmentation

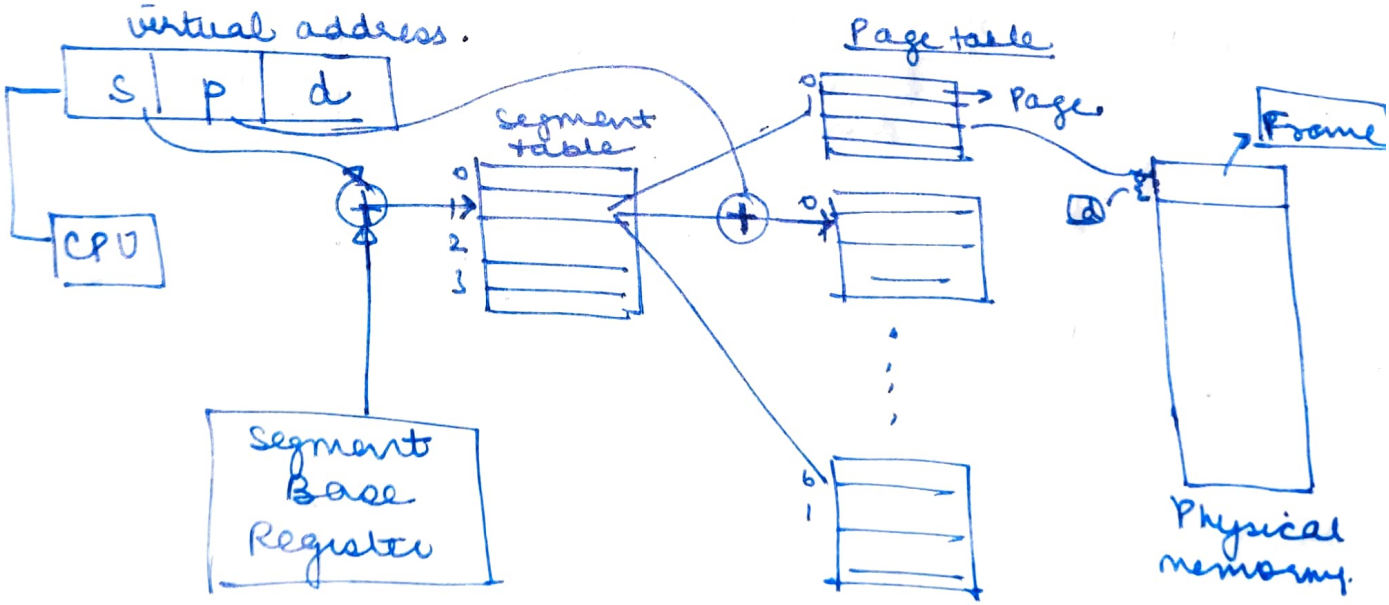
Segmentation with paging





→ Divided into segments and individual segments have pages.





disadvantage → two lookups