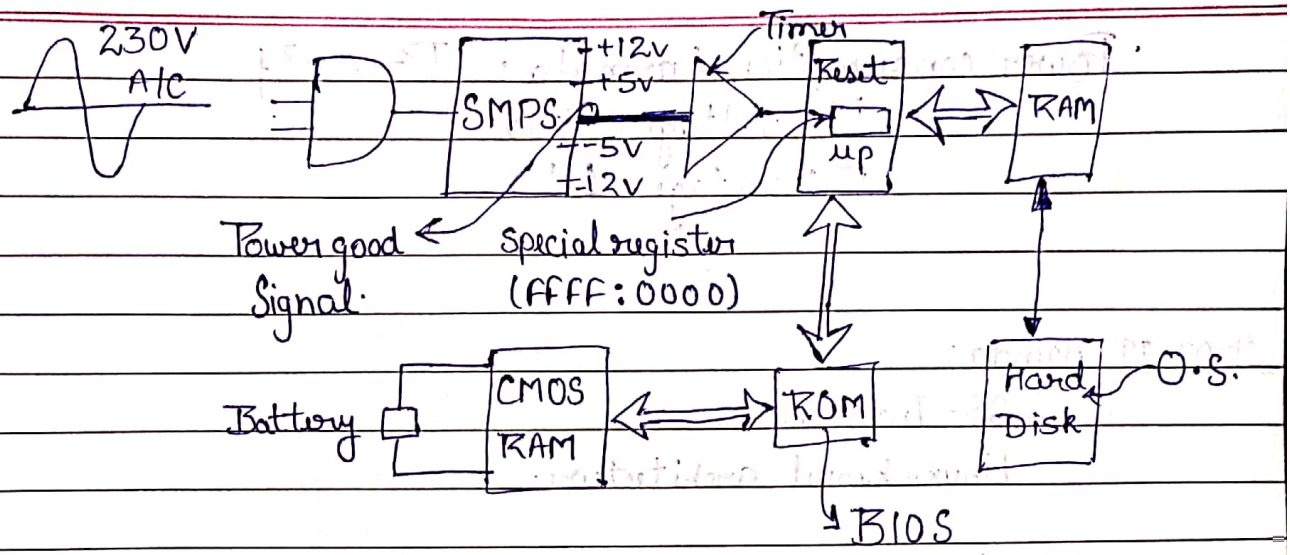


POST: Power-On Self test.

Booting: Transfers OS dependent files from secondary memory to RAM.

* Internals:

- Cold boot VS Warm boot: When we boot a switch off system it is cold boot. But when we are stuck somewhere and we boot the system then it is called warm boot. In warm boot POST will get skiped.

- MBR (Master boot record).
  - ↳ It contains the primary boot loader
  - ↳ It also contains Partition Tables.

- Boot strap loader.

- Boot loader
  - primary
  - secondary (↔ R.B)

* Boot strap loader
  ↓
  Boot loader
  ↓
  Bootable Device.

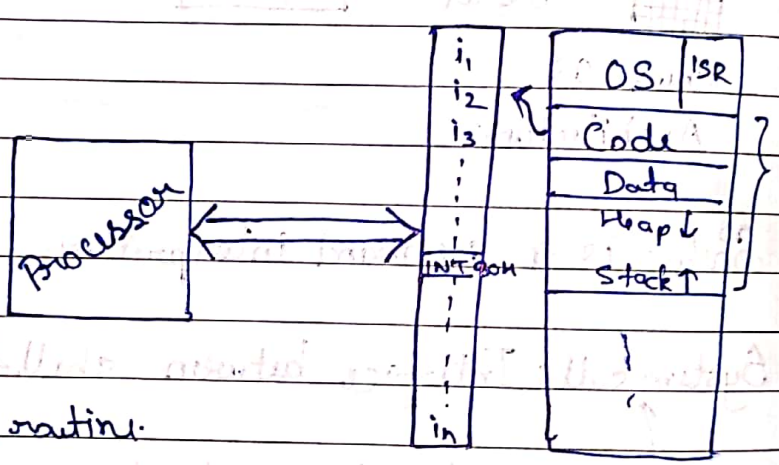
Agenda:
- Linux booting
- Linux OS stack.

ps -ef | grep a.out : It will give lots of information

ppid	pid	----
1	2	term
2	3	a.out

7-8-19 Saidalvi Sir:

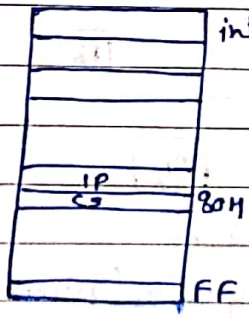
INT 80H
H/W interrupt is active. Exception event occur.



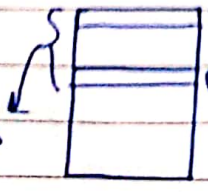
ISR: Interupt system routine.

Interrupt vector table:

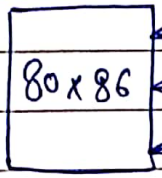
When a interrupt is there we have to update IP & CS or the code segment (CS) for interrupt is different from that of user code segment.



In 80x86



So, the first 1KB of main memory is used to store Interrupt vector table.



Note: If 'I' (Interrupt) flag is disabled INTR is also disabled.

All the user defined software interrupt are INT 33H
|
INT FFH.

When the ~~pgm~~ interrupt occurs we will:
- Save the flag to stack
- Save CS to stack
- save IP to stack
- clean I,T flag.
- load IP from IVT
- load CS.

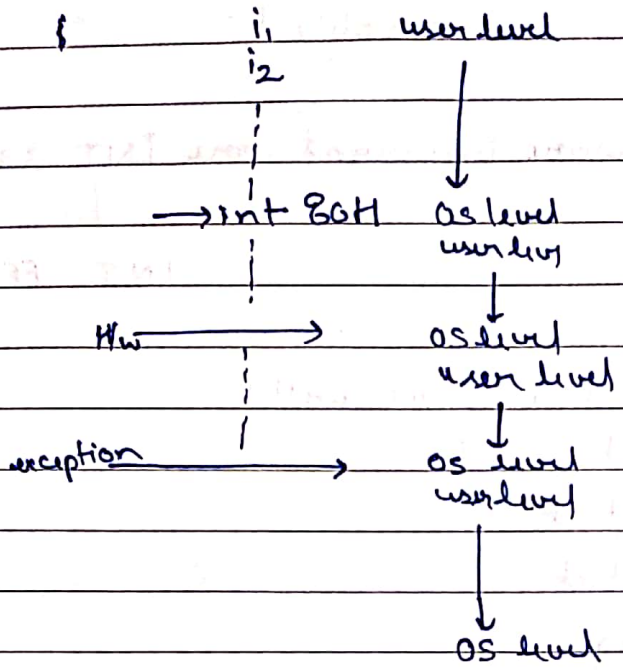
* The last statement of ISR contain IRET which will restore the IP, CS & flag values by poping it from the stack.

* To regain all the values that has been calculated before the interrupt we have two options:

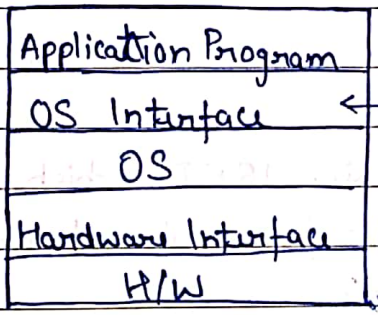
1) User itself use PUSHA before the interrupt and POPA to retrive all the values
2) ISR can initially have PUSHA and just before IRET instruction there must be POPA instruction to retrive all the values.

* If the IVT (Interrupt vector table) has not fixed location in memory then IDTR will contain starting location of IVT.

Process

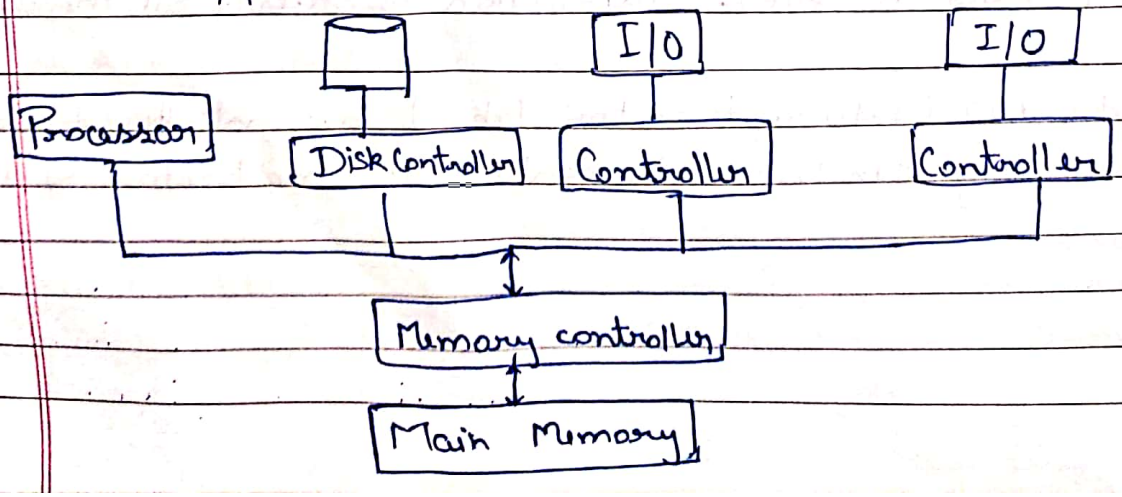


14-8-19

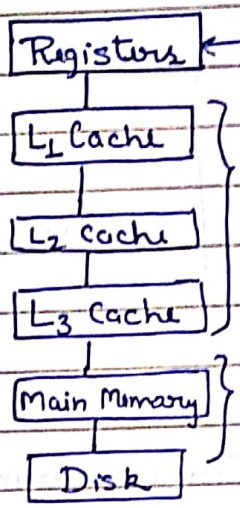


Ring level:

00 ← OS
01
10
11



Memory



← Compiler is manage these.

} Managed by machine/ processor.

} OS manages these.

Polling: For an I/O device processor will always check the buffer and if it is full processor will save the buffer input into main memory and clear the buffer.

Disk to Main memory is DMA (Direct memory access) transfer.

There are 317 system calls in our Linux system.
Command: "NR_syscall".

```
C = C + 10
count = read (fd1, buffer, n);
```

User level:

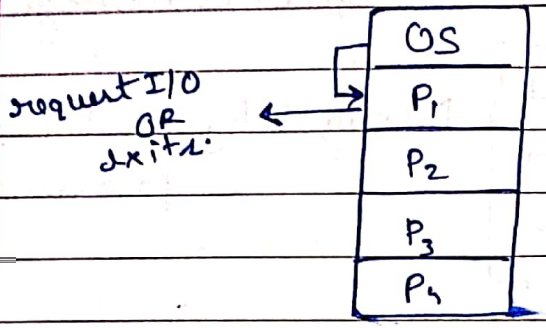
```
LD R2, 0(R1)
ADDIV R3, R2, 10
SD R3, 0(R4)
PUSH N
PUSH buffer
PUSH fd1
read ( )
```

→ Library

```
MOV, EAX, system call No.
INT 80H.
```

If read 0 then library will go to OS level and start take the address of ISR will be there in syscall table and will start the execution from given ISR table.

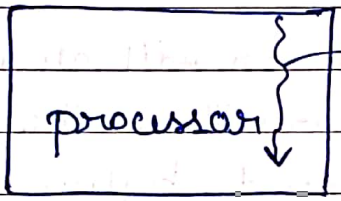
1960's OS looks like:

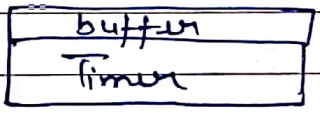


Multiprogramming.

Next level of OS was:

Time sharing.





Jayraj Sir:

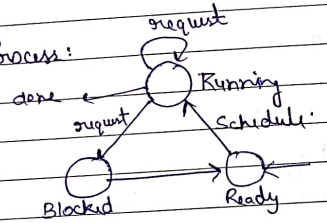
Agenda
- Process
  - Process context
  - Process life cycle.
  - Process creation
    - fork()

Process:
It is a program in execution

- Process context comprised of
  - Text
  - data
  - Stack
  - Other resources.
- Process Control Block (PCB).

* Linux is having only two things - (i) File
  (ii) Process.

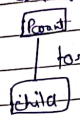
State of a Process:



Process Manager:

Process creation and Execution:

How to Create New Process?



30-8-19

CPU Scheduling:

CPU scheduling refers to the task of managing CPU sharing among a community of ready process.

- Mechanism
  - Hardware, Data structure & Algorithms.
- Policy - strategy.

Part of scheduler:

(i) Enqueuer
(ii) Dispatcher
(iii) Context switcher.

Agenda:

- CPU Scheduling
  - Mechanism
    - Enqueuer
    - Dispatcher
    - Context Switcher.
  - Policy - strategy
    - Non - preemptive scheduler
      - FCFS
      - EDFS
      - SJN → Shortest job next.
      - .
    - Preemptive scheduler.

31-8-19

* Each context switches requires:

$(n+m) * b * k$ time unit

- n: gen. register
- m: status reg.
- b*k: operation

- n=32, m=8, b*k=5ons.

* Pre-emptive & non pre-emptive kernels:
  - A scheduler that uses involuntary CPU sharing is called non-pre-emptive kernels.

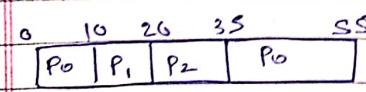
31-8-19 Agenda:

- ☐ Pre-emptive schedulers:
  - Pre-emptive SJN
  - Pre-emptive priority Scheduler
  - Round Robin
  - Multi Level Queue
  - Multilevel Feedback Queue
- ☐ Linux scheduler

Pre-emptive SJN:

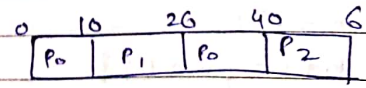
P_i	Service Time	Arrival Time
P_0	30	00:00
P_1	10	00:10
P_2	15	00:15

Gantt Chart



Pre-emptive Priority scheduler:

P_i	$T(P_i)$	Priority	Arrival Time
P_0	30	2	00:00
P_1	10	1	00:10
P_2	20	15	00:15



Linux Kernel Scheduler:

5.2.11

major release ← | → revision

minor release

even: stable / odd: developmental

* Threads, Thread management:

3-9-19

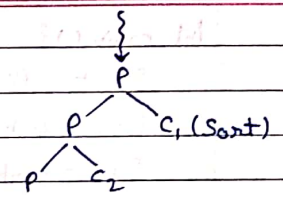
AP
System calls → OS interface
OS
ISA → H/W Interface
H/W

```
Pid_t fork()
void exit(int returnstatus);
Pid_t wait(int *return code);
int execv(char *Pathname, char *argv[])
```

↳ Overwrite the code present in P1 with the code present at the above given pathname and arguments for the program will be passed through argv[].

```
int main() {
    Pid_t Pid1, Pid2, Pid3;
    int status;
    int n, m;
    n = 16;
    Pid1 = fork() {
```

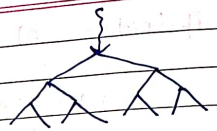
```
if (pid1 == 0) {
    n=n+5;
    printf ("%d", n);
    execv ("Sort", NULL);
    n=n+20;
    printf ("%d", n);
}
else {
    Pid2 = fork();
    n = n+2;
    Pid3 = wait (&Status);
    printf ("%d", n)
}
}

int main () {
  pid_t  Pid1, Pid2, Pid3
  int status;
  int n, m;
  n = 10;
  Pid1 = fork()
  if (pid1 == 0) {
      n = n+5
      printf ("%d", n); ⟶ 15
  }
  else {
     n = n+2;
     Pid3 = wait (& Status);
     printf ("%d", n); ⟶ 12.
    }
```



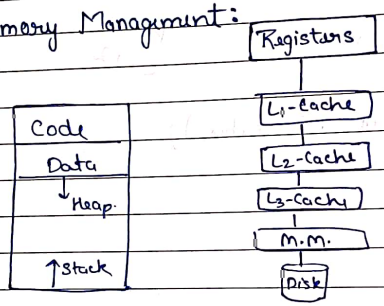
p-12
c-12

Process control block (PCB) will store all the ~~status~~ return status of child till parent also died.

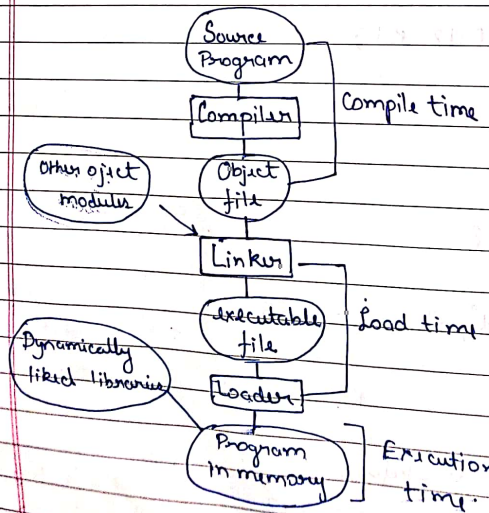
```
int main(){
    Pid_t  Pid1, Pid2, Pid3;
    Pid1 = fork();
    Pid2 = fork();
    Pid3 = fork();
}
```

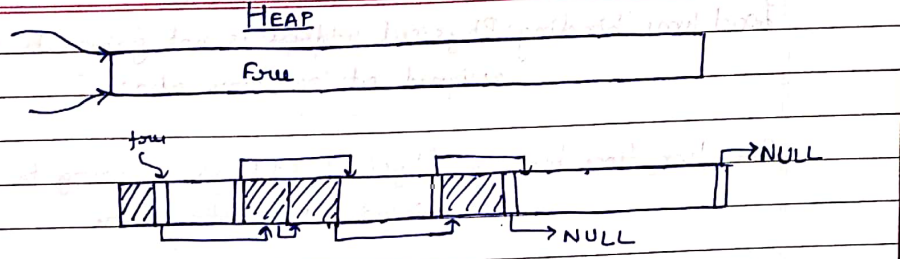


* Memory Managment:



4-9-19



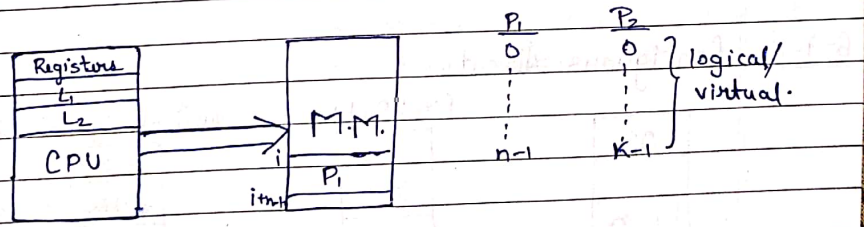


1) First fit: It will search for the first block which can have the size greater than the required size.

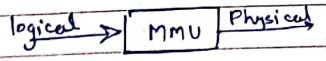
2) Next fit: It will start the search from the block where we have found the required block the next block which is greater than equal to required block size will be allocated.

3) Best fit: It will search for the smallest block which have memory greater than required one.

4) Worst fit: It will allocate the longest free block.



MMU (Memory management unit)



Compile time binding: It will assign the logical address but won't going to bind it with physical address otherwise it cannot change the location in MM at the time of execution.

Load time binding: Physical address is not going to get assigned at load time also.

Execution time binding: Physical address is going to get assigned at execution time.

~~Address~~

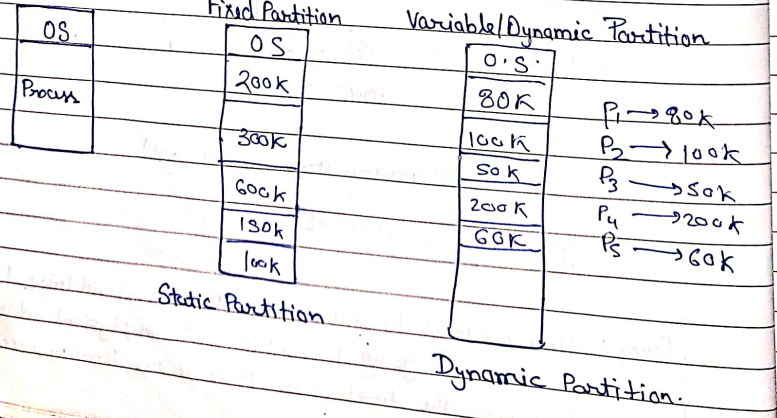
Dynamic loading: Load a module only if required at the time of execution time

Dynamic linking libraries:

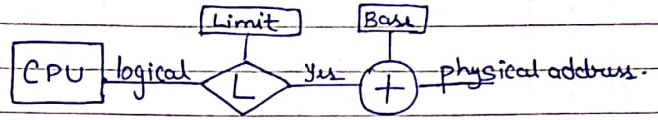
IO program ⟶ strcpy()

In static linking 10 copies of strcpy() is required by in case of dynamic only one copy is required and if the same function called by other function it will check for it MM and if find then it will use that copy without loading it again.

6-9-19 Contiguous Allocation:



Base register → PTBR
Limit → PTLR



Problem with contiguous allocation:

Fragmentation

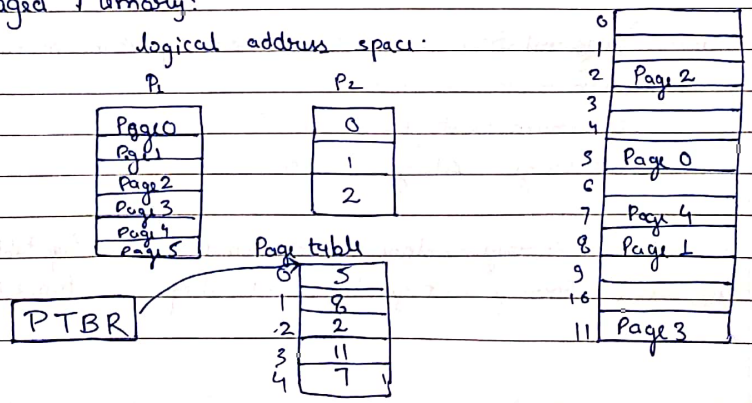
- Internal (Fixed Partition)
- External (We have required memory but the problem is it is scattered in main memory so we cannot use that to load other program) It occurs in both Static as well as dynamic partition.

Solutions for external fragmentation in Dynamic partition:

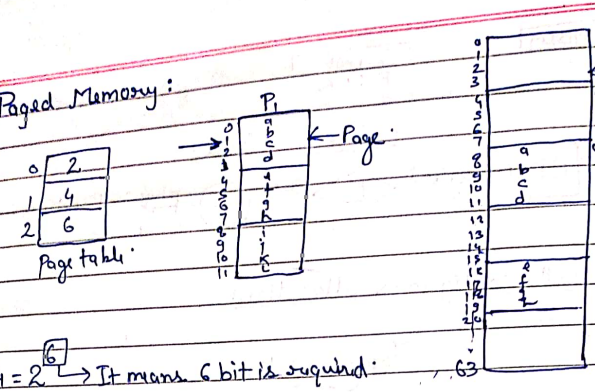
(Compaction)- We can move all the used area block upward so that we can get all free block together at the lower part of the MM.

Non-contiguous allocation: (Paging)

Paged Memory:



13-9-19 Paged Memory:



$64 = 2^{6}$ → It means 6 bit is required.

4	2
P	d

Page No. offset — Logical address

4	2
f	d

frame No. offset — Physical address.

Let change physical address of P1(1) to logical address in above ex.

0000|01

↳ 0.

↳ It contain 2 in page table. ⇒ 0010|01

0010|01

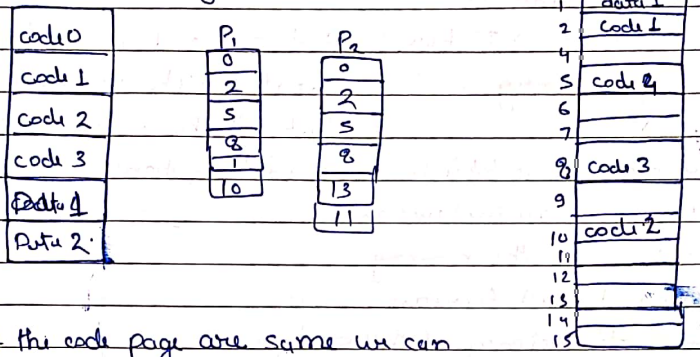
↳ 2 contains 6 in page table ⇒ 0110|01

17-9-19

* Internal fragmentation is possible in case of Paged memory allocation
  Maximum wastage = 4KB − 1
  Minimum wastage = 0KB
  ⇒ Average wastage = 2KB.

* Page size ↑ Average wastage due to Internal Fragⁿ ↑ Page table size ↓
  Page size ↓ Average wastage due to internal Fragⁿ ↓ Page table size ↑

* Shared Memory:

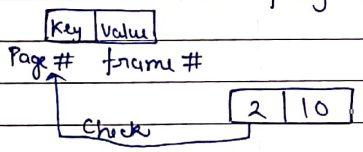


If the code page are same we can improve the performance.

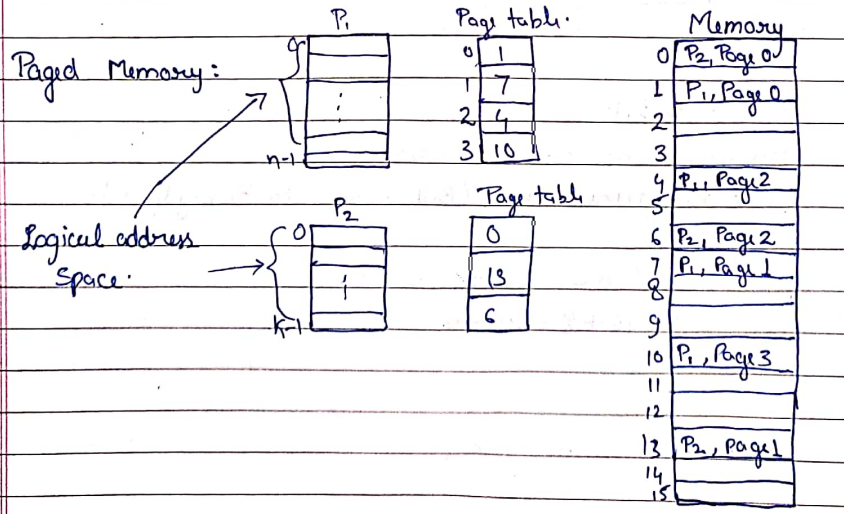
10 ⟶ 60 (if we are loading all the pages)

10×2+4 = 24 (if we are using shared memory mechanism)

* We have a hardware cache for page table i.e. TLB
Translation lookaside buffer.
(It is a fully associative cache).



17-9-19 Paged Memory:



Stored page table in:

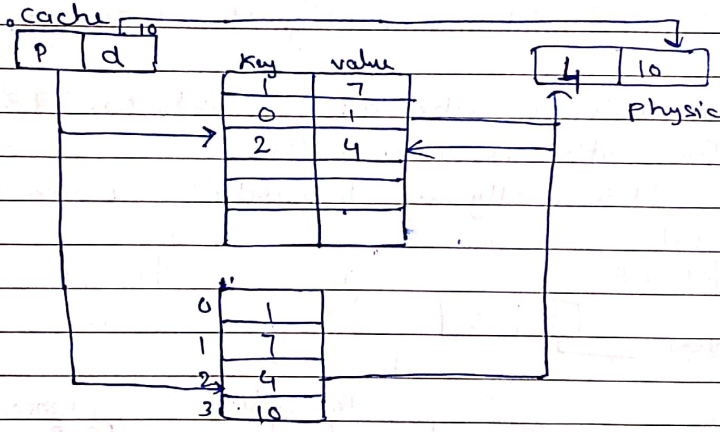
(1) In registers ⟶ It have limited size and at the time of context swich we have to flush all the value which is a costly action.

(2) In main memory

↓

If we store page table in mm for a single instruction we have the access the memory twice which will decrease the ~~efficient~~ efficiency of the system.

(3) In hardware cache i.e. TLB (Translation lookaside buffer).



80% TLB hit

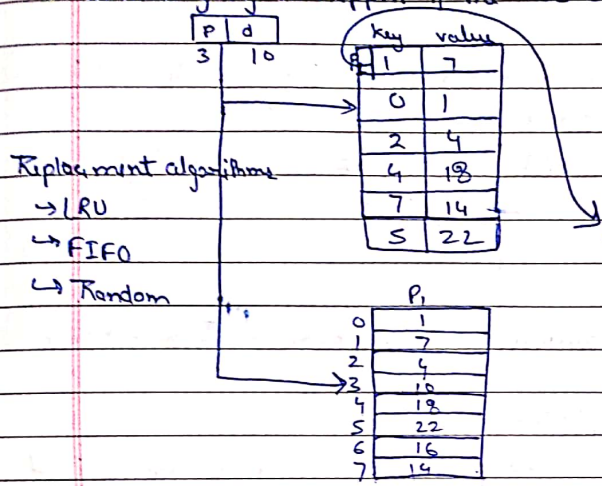
Memory access time = 20 nsec

TLB hit time = 1 nsec (But assume to be negligible).

AMAT = 0.80 × 20 + 0.20 × 40

(Average memory access time) = 16 + 8 = 24 nsec.

* What is going to happen if the TLB size is smaller than page table?



Replacement algorithms
→ LRU
→ FIFO
→ Random

* At the time of context switch we must flush the TLB.

ASID: Address space identifier

* Protection:

* We have one more table along with page table i.e. Frame table which keep track ~~the~~ of frame whether a frame is free or allocated if allocated then to which process and much more details.

There are space in Frame table for these entries also:

1 — read only / read-write
1 - valid / invalid
1 — Dirty
1 - reference.

PTLR: It will show the size of Page table.

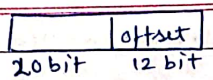
18-9-19

① Structural page table:

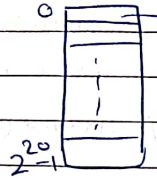
1. Hierarchical Page table:

IA32 —— 2^{32} ⟶ 32 bit logical address
32 bit physical address
Page size 4 KB.

F



In the above case, maximum entries in a page table will be-



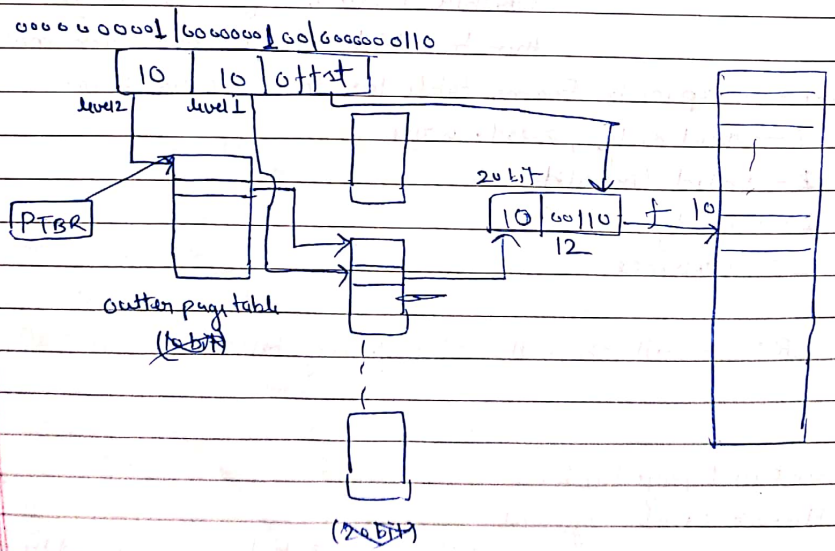
And each entry will be of size 4KB (Size of the logical address)

∴ Total memory required of storing page table in that case will be equal to:

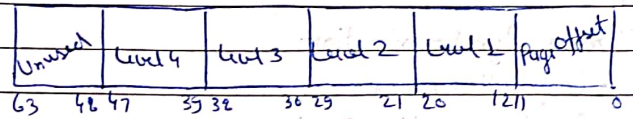
$$2^{20} \times 4 = 4MB$$

Where as page size is 4KB ⇒ 1024 (2^{10}) pages are required to store the page table.

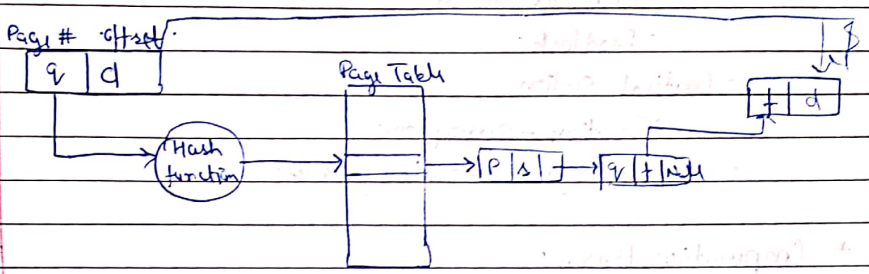
This is the case when we are allocating contigous memory. So we will go to the page table (which will vania all our paging concept) So we will go for other location:



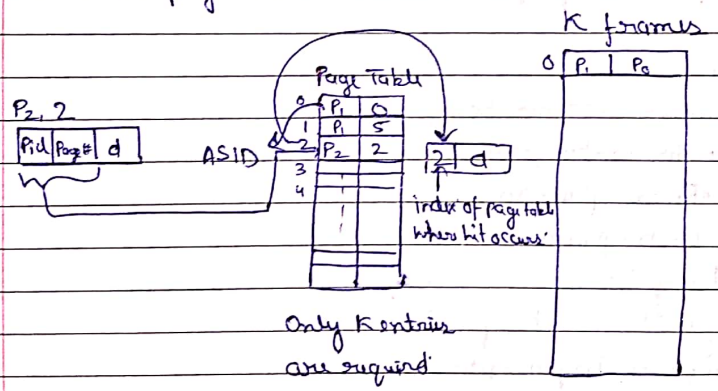
In intel 64 bit:



② Page # offset



③ Inverted page table:



The main problem with above method was is we have to search for all the elements in page table to get the required frame number. Second problem is we can't use shared pages.

Operating System

Modern perspective

classmate
Date
Page

18-9-19

Agenda:

- Process Synchronization
- Cooperative process
- Synchronization problem
- Critical Section
- Deadlock
- Critical Section
- Solution - Semaphore
- Classical synchronization problems

* Cooperative Process:

Process cooperating to work each other to accomplish a specific task

20-9-19 Agenda

- Classical synchronization problem
- Reader writer problem
- Reader precedence
- Writer precedence
- Dining Philosophy Problem
- Sleeping barber
- Cigarette smoker problem
- Semaphore Implementation
- Test & Set instruction