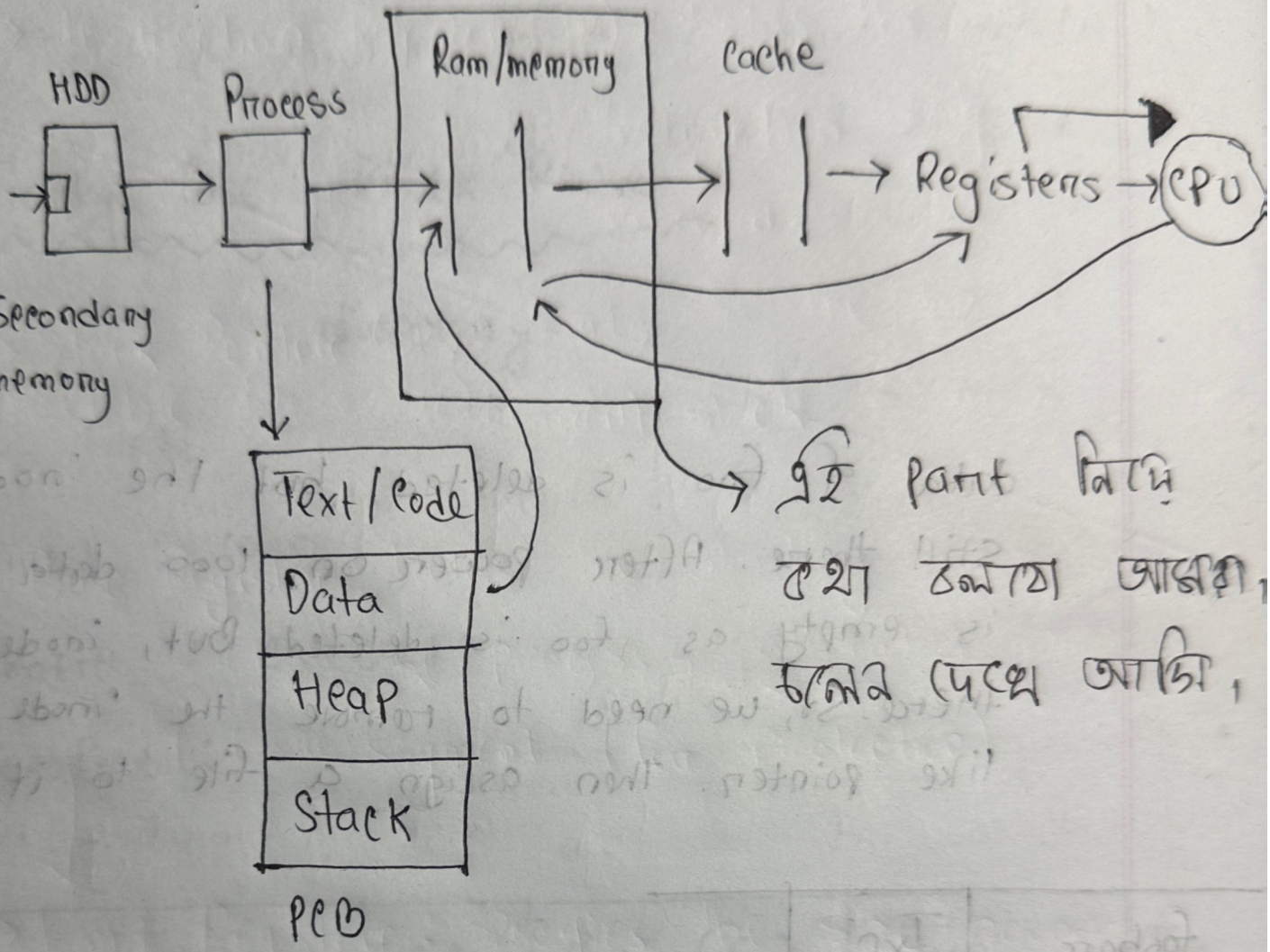
Video : 01

Memory Management

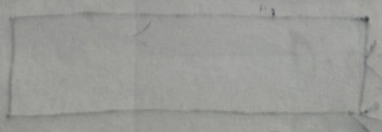


এই part নিয়ে কথা বলার আগে, চলো দেখে আসি,

memory থেকে RAM faster.

আবার RAM থেকে Register faster.

RAM থেকে আবার cache faster.



* How Main Memory & CPU registers interact and run a process by CPU

Register

A register is a small, high speed storage location inside the CPU. Much faster than the main memory (RAM). Stores temporary data the CPU is currently working on. Size is usually in bits or words. 32 bit or 4 byte.

Some types of register includes

i) General Purpose Registers used to hold data, operands or intermediate results.
Examples: R0, R1, R2 .... in simple CPUs

ii) Special Purpose Register. These control

→ Program Counter (PC): Holds the address of the next instruction to be executed.

(xxx)

→ Instruction Register (IR): Holds the current instruction being decoded/executed.

→ Stack Pointer (SP): Points to the top of the current stack in memory. ~~Base~~

Base Register: Holds the starting address of the process's allocated memory segment. (for protection)

→ Limit Register: Specifies the size (boundary) of the process memory segment.

→ Processor Status Word (PSW): Holds status bits (eg. mode bit → user/kernel condition flag like zero, carry, sign, overflow).

iii) I/O and Control Registers.

Main Memory

→ main memory is the large storage area directly accessible by the CPU.

→ It stores both program instructions and data while a process is running.

→ It is volatile → contents are lost when power is off

→ example : RAM

→ stores active programs and the data they are working on.

→ each program loaded from secondary storage (disk) into main memory before execution.

→ CPU fetches instructions and operands direct from main memory.

We will be focusing on the main memory since that is where the main information are stored and CPU registers just fetch from them and run.

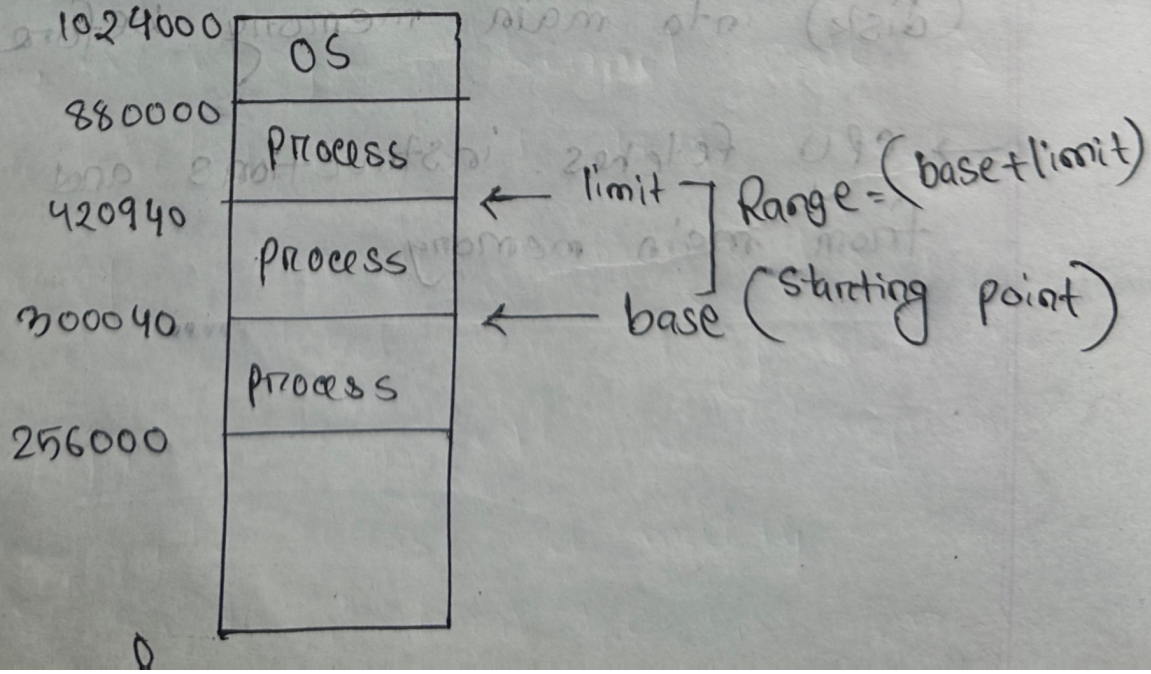
executable file (inode) → called exec() → loader

↓

memory space allocation → PCB created → PCB added to ready queue

↓

Scheduler dispatch → CPU execute.



→ each process has a separate memory space. Separate per process memory ~~state~~ space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution. To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a base and a limit.

→ Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other ~~part~~ users' memory results in a trap to the operating system, which treats the attempt as a fatal error.

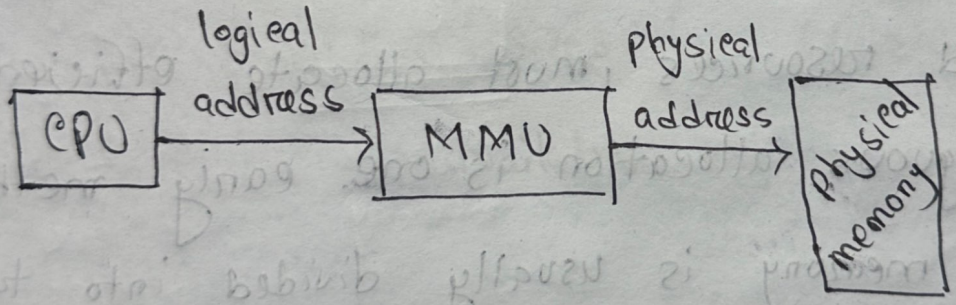
→ The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers. This scheme allows the operating system to change the value of the registers but prevents users programs from changing the registers contents.

→ The OS, executing in kernel mode is given unrestricted access to both operating-system memory and users memory. This provision allows the OS to load users programs into users memory, to dump out those programs in case of errors, to access and modify parameters of system calls, to perform I/O and from user

memory and to provide many other services.

→ Usually a program resides on a disk as a binary executable file. To run the program must be brought into memory and placed within the content of process, where it becomes eligible for execution on an available CPU.

Logical vs Physical address Space



→ Binding addresses at either compile or load time generates identical logical and physical addresses. However, the execution time address-binding scheme results in differing logical & physical addresses. We usually refer the L.A as Virtual address. The user program never accesses the real physical addresses.

Contiguous Memory Allocation

→ যেই যেই গুলা allocate হয়া, No fragmentation.

Non Contiguous memory Allocation : PAGING

The basic method for implementing paging

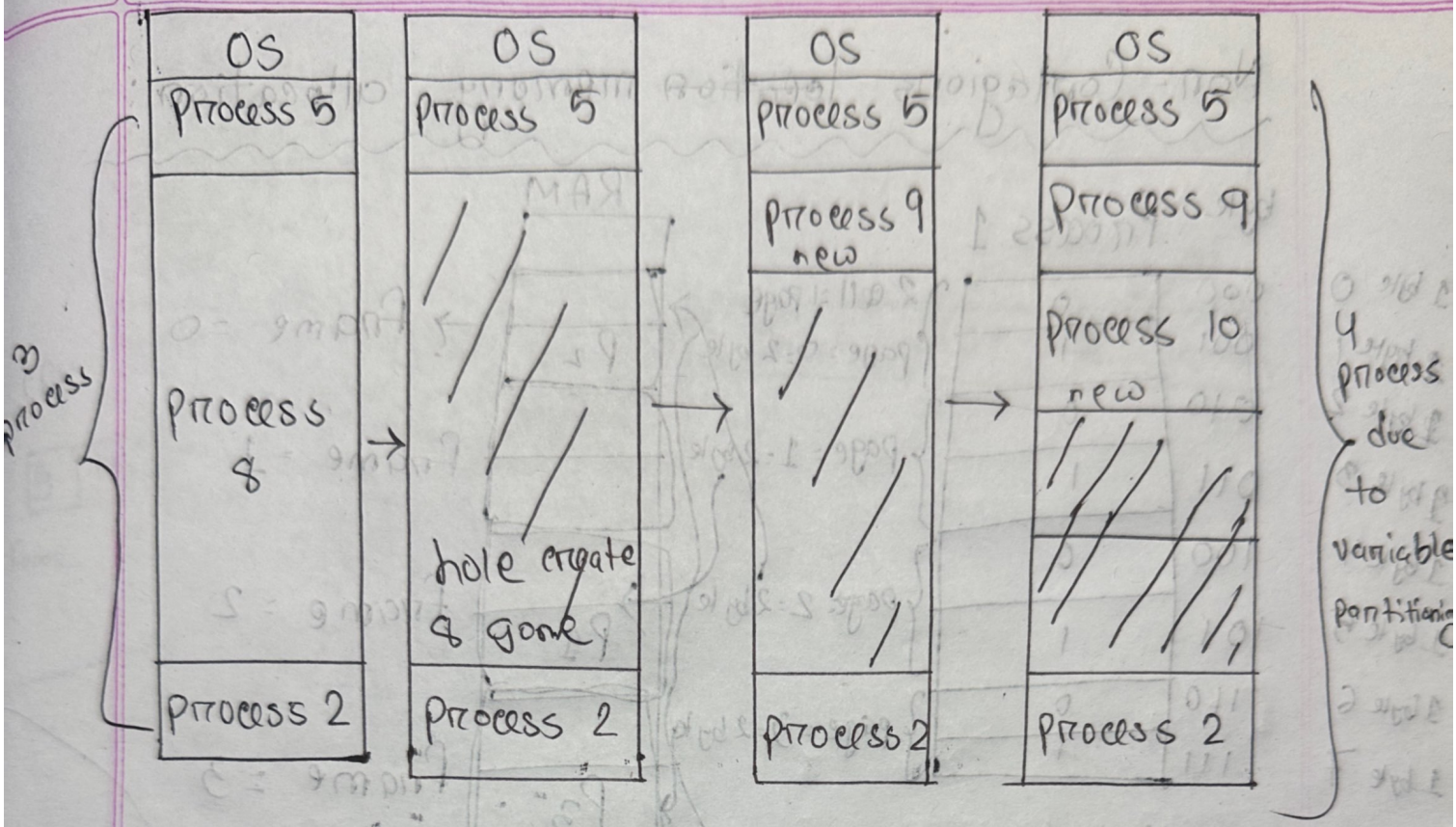
→ main memory must accommodate both OS and user processes.

→ limited resources, must allocate efficiently.

→ contiguous allocation is one early method

→ main memory is usually divided into two partitions:

* resident OS, usually held in low memory with interrupt vector
* user processes then held in high memory
* each process contained in single contiguous section of memory.



fixed partition = loss

Variable partition = sizes for efficiency

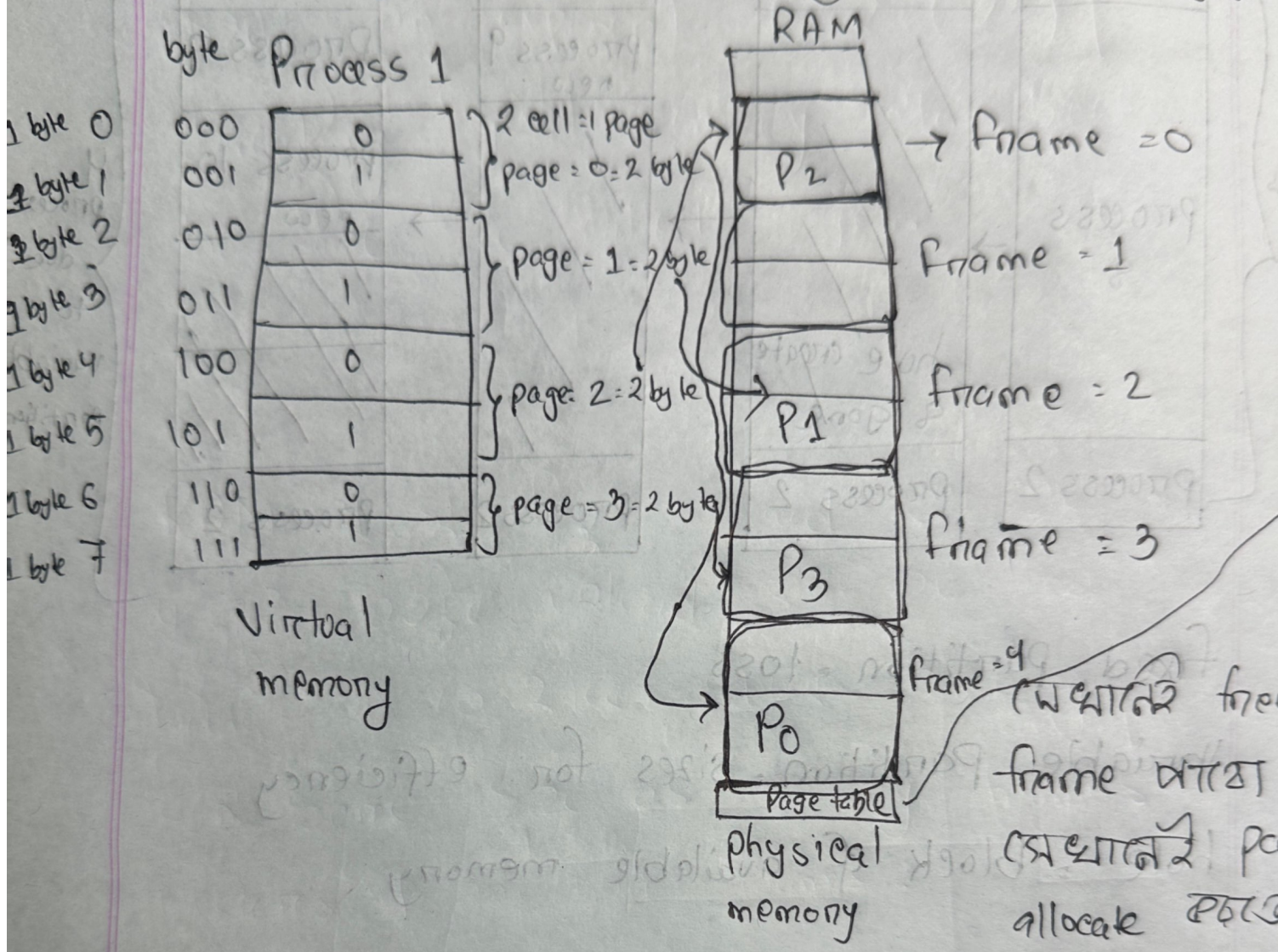
hole = block of available memory.

best fit : speed slow but memory optimized

first fit = Average efficiency but good speed.

worst fit = আকার আকার বড় longest contagious block নেয়া

Non-Contagious ~~location~~ memory allocation:



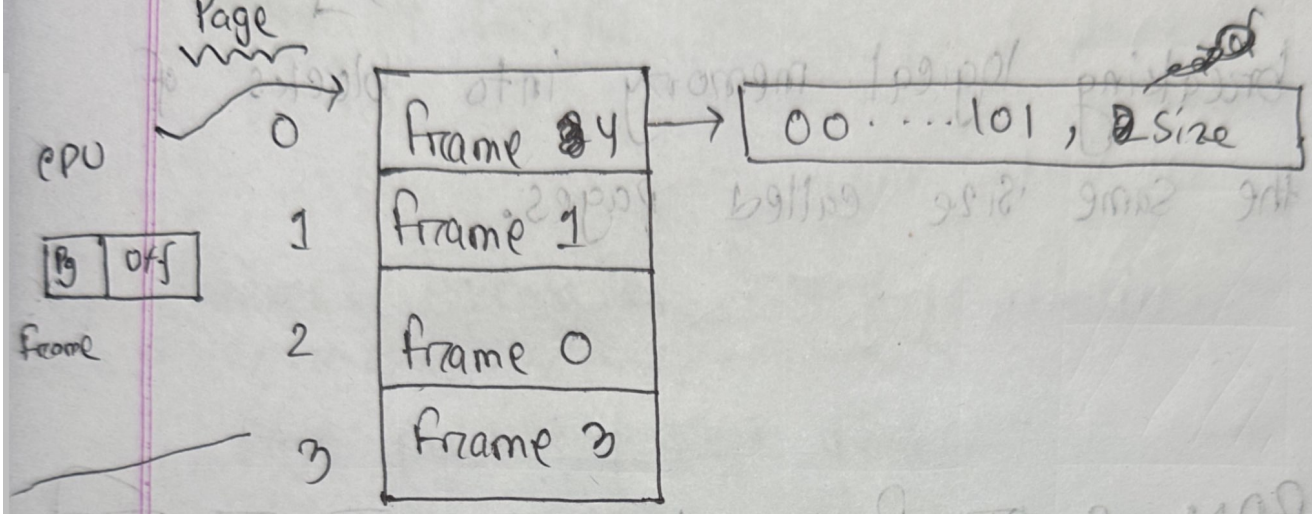
যেখানে free frame পাবো সেখানেই page allocate হতে পারবে, No need contagious allocation.

Every page has 2 info

i) Page no = (Page 0, page 1 etc)

ii) Page offset = Page এর মধ্যে index to identify cell.

Page Table



Page table is saved in RAM. Named PTBR

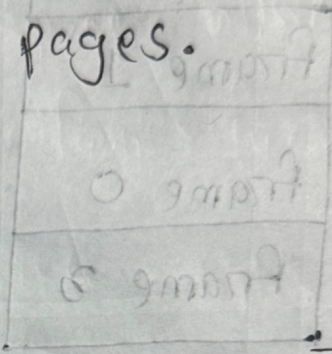
↳ Page table base register also has a size.

PCB ৩ RAM ৭ store করে ২৭।

PAGING

The basic method for implementing paging involves breaking physical memory into fixed sized blocks called frames [general convention is

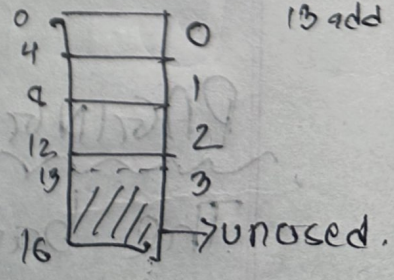
divide it in a way that each frame size is an exponent of 2 (2^n)] and breaking logical memory into blocks of the same size called pages.



RAM এ সব কিছু upload করা হয় না, যতটুক প্রয়োজন ততটুক upload করা হয় and বাকি কিছু HDD তে থাকে, এটাকে backing store বলে। Same size as memory frames or clusters of multiple frames.

PAGING এ

→ Internal fragmentation হতে পারে



13 add

How paging works in a simplier explanation:

আগে না লিখছি তাই।

Address Translation Scheme

Page number	Page offset
m-n	n.

$2^1 = n = 1$.

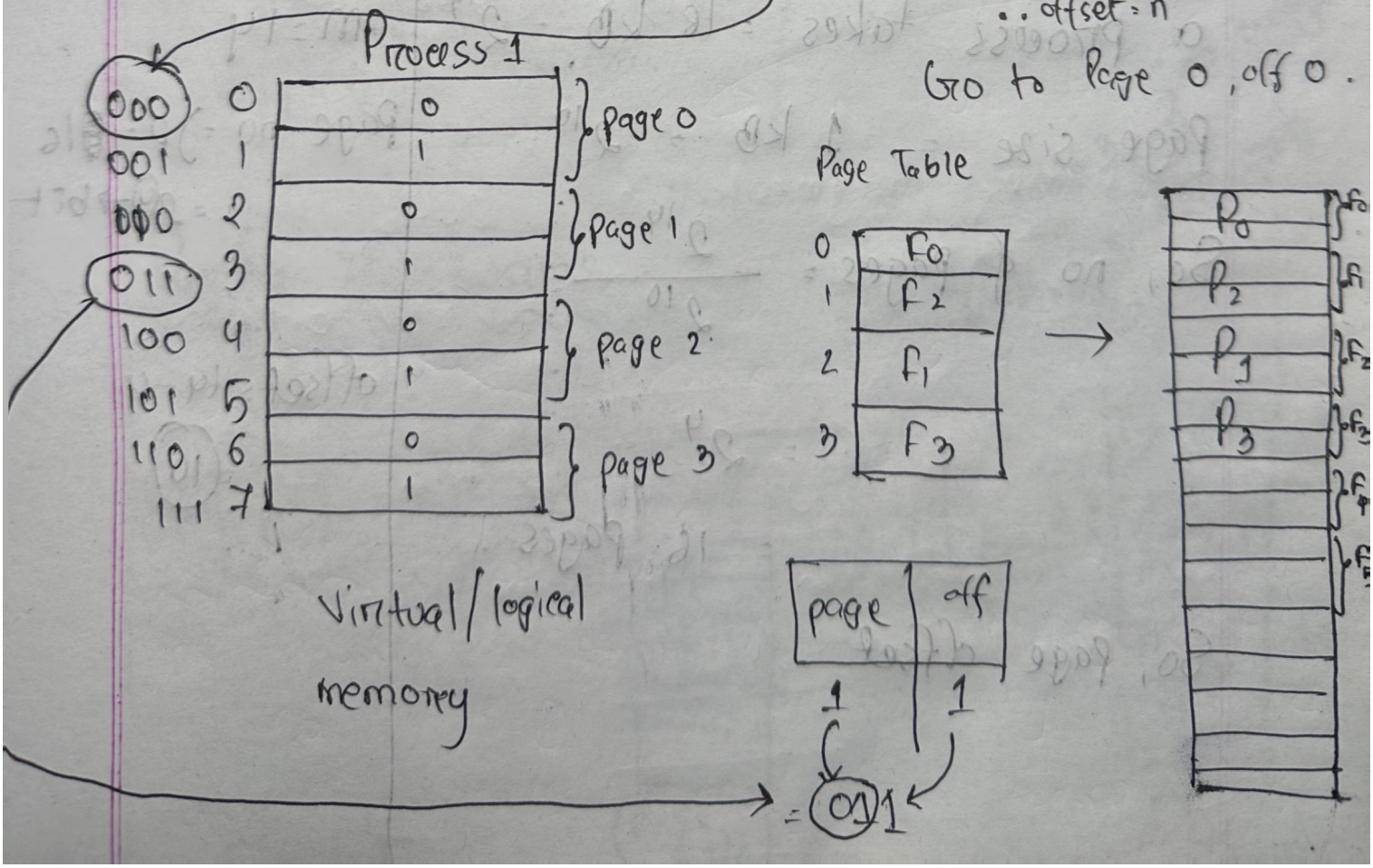
offset - 0 & 1

So, value 2

Virtual address space :) 8 :) 2^3 :) 3 bit :) m

∴ offset = n

Go to Page 0, off 0.



Different value দিয়ে দিয়ে and কত্তে to find

i) total address bit দিয়ে দিতে পারে?

ii) Virtual address space কতগুলো

iii) Per page size দিয়ে দিতে পারে।

---

* Maths

a process takes = 16 kB = 2^{14}

Page size = 1 kB = 2^{10}

So, no of pages = $\frac{2^{14}}{2^{10}}$

$= 2^4$

= 16 pages.

So, Page offset

m = 14

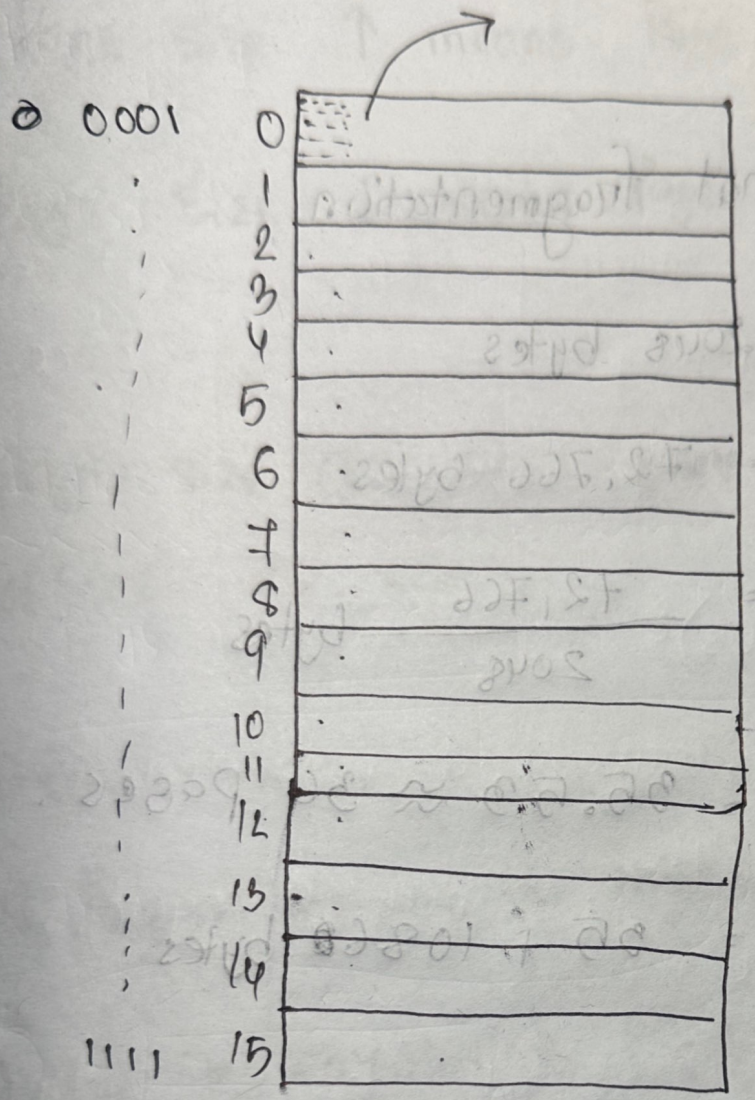
Page no = p = 16

= 4 → bit

offset = 14 - 4

= 10

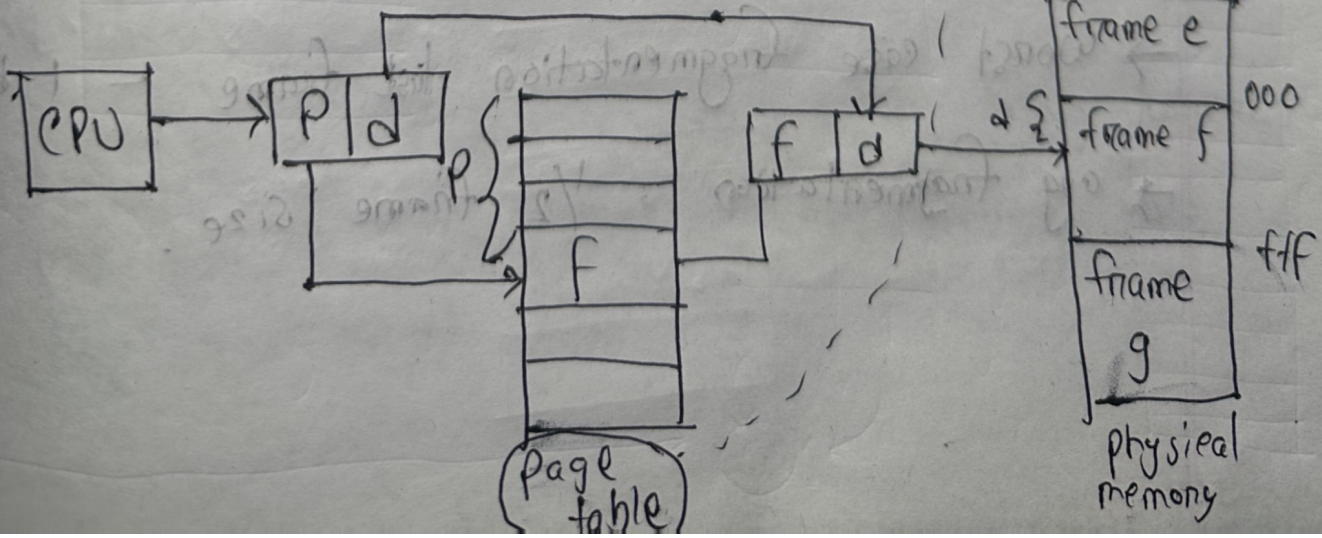
প্রত্যেক তার ভিতরে 14 টি করে
ঘর আছে। So, 14 bit
করে represent. And,
সেখানে 4 bit page no
and 10 bit offset.



its contagious

page table 3;
RAM এ save
করা আছে

Paging Handware



Size of Page

- Calculating internal fragmentation

→ page size = 2048 bytes

→ process size = 72,766 bytes

→ no of pages = $\frac{72,766}{2048}$ bytes

= 35.53 ≈ 36 pages.

= 35 + 1086 bytes

→ internal fragmentation = 2048 − 1086

= 962 bytes.

→ worst case fragmentation = last frame − 1 byte

→ avg fragmentation = 1/2 frame size.

Page size ↑ means Page bit ↓ offset ↑

Page size ↓ means Page bit ↑ offset ↓

Page size ছোট মানে অনেক page লাগবে আর offset কম লাগবে।

After this.

↳ logical memory & physical memory are different

↳ by implementation process can only access its own memory.

Free frames

OS এর মধ্যে Frame Table থাকে from where we can get to know the which frames are occupied and which are free.

Video: 02

Some additional Details

When a computer boots, part of the OS is loaded into main memory (RAM) so it can execute. Not everything is loaded at once. The main parts typically include:

Kernel (core of OS):

→ manages CPU, memory, devices and processes

→ Process schedulers, memory manager, interrupt handlers, Device drivers

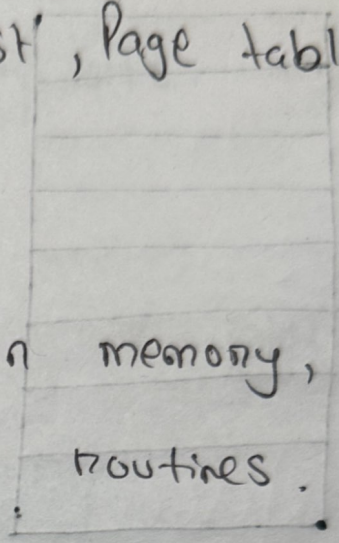
→ loaded in a protected area of RAM (kernel space)

System Programs / Utilities (Sometimes loaded as needed):

→ File system management routines, I/O routines, Network Services.

Data Structures for OS management:

→ PCB, free frame / block list, Page table, I/O buffer



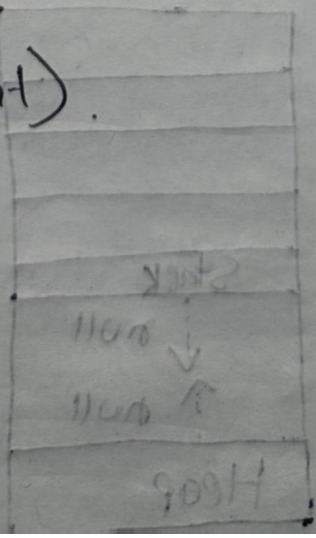
Interrupt Vector Table

→ located at a fixed place in memory, holds addresses of interrupt service routines.

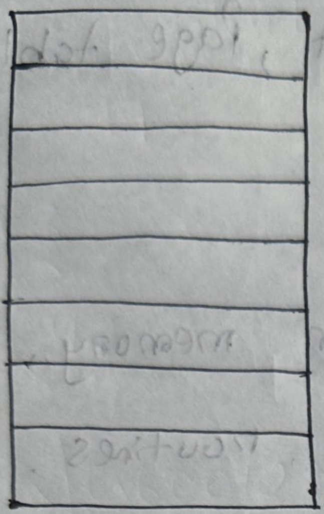
Modern OS (Paged kernel)

→ The OS kernel itself can be paged like a regular process.

→ Its code and data are divided into pages, which can be swapped in/out of main memory if needed (though some critical kernel pages always stay resident).

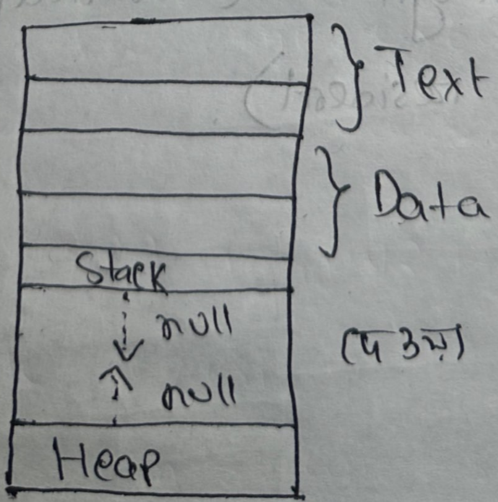


PCB:



How the page table differentiate between each part of the process. ⟶ one page, only 1 segment

How does the dynamic allocation for heap and growing stack is allocated inside the page table.



(ফাঁকা) থাকে so valid, invalid.

Page table Entries (PTEs) and Flags

each page table entry has flag bits:

R/W bit → is it writable?

X bit (Nx or executable) → can it be executed?

U/S bit → user vs kernel accessible

P(present) bit → is it in RAM?

The OS sets these flags differently for different regions:

Text (code): read + execute, not write

Data/heap/stack: read + write, not execute

Guard pages (like stack limit): not present → page fault if touched.

Now lets look at how a dynamic memory allocation for a instruction inside your code may look like:

code:

*p = 42

↳ [0x600123], eax → heap address

CPU: "Need VA 0x600123."

MMU: split → page = 0x600

offset = 0x123

TLB ~~Miss~~ → page table walk.

PTE says: VA page 0x600 → PFN 42, RW permission

MMU: Translate → PA = from 42 + 0x123 = 0xA5123 ↓ actual RAM byte address

CPU: Send write request to PA = 0xA5123

Cache / memory system → writes value

Execution continues.

Implementation of Page Table

→ Page table is kept in main memory.

→ Page table base register (PTBR) points to the page table.

→ Page table length register (PTLR) indicates size of the page table.

→ The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative memory or translation lookaside buffers (TLBs).

Effective Access Time

Hit ratio = α

miss ratio = $1-\alpha$

$\alpha = 80\%$.

ε = 20 ns

↳ TLB search

memory access = 100 ns

TLB ← 20+100 ← RAM

EAT = .80 × 120 + .20 × 220

hit ↲ miss ↘ 20+100+100

TLB RAM RAM

= 140 ns.

* $\alpha = 99$

$\varepsilon = 20$ ns (TLB search)

memory access = 100 ns.

EAT = .99 x 120 + .01 x 220

= 121 ns.

Memory Protection

Valid and invalid bit attached to each entry in the page table:

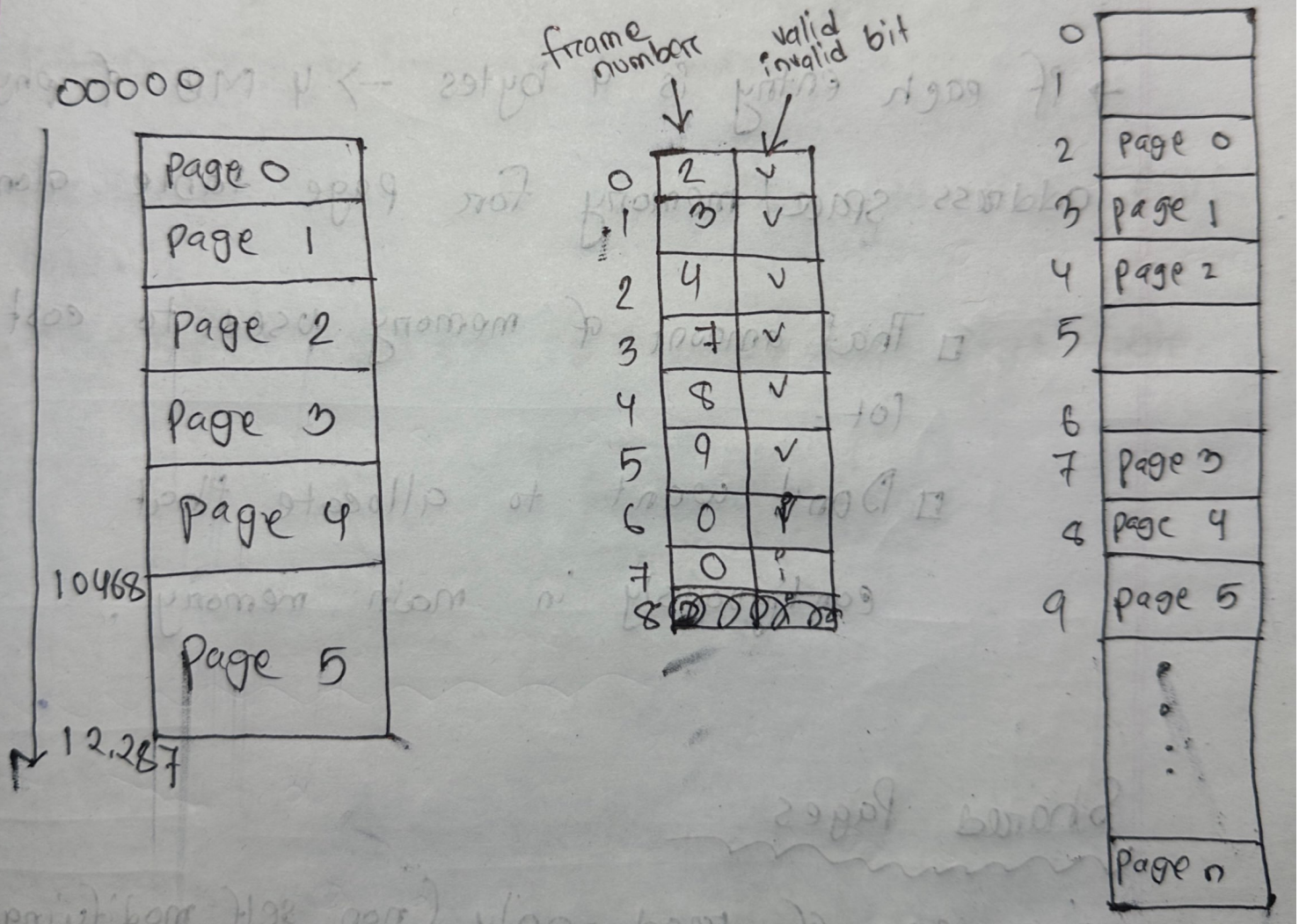
→ valid indicates that the associated page is in the process logical address space, and is thus a legal page

→ invalid indicates that the page is not in the

Process' logical address space.

→ Or use page-table length register (PTLR).

Any violations result in a trap to the kernel.



Structure of Page Table

→ logical address 32 bit

→ page size 4 kB (2^{12})

→ page table would have 1 million entries ($2^{32}/2^{12}$)

→ if each entry is 4 bytes → 4 MB of physical address space / memory for page table alone.

- That amount of memory used to cost a lot.
- Don't want to allocate that contiguously in main memory.

Shared Pages

→ one copy of read-only (non self modifying) code shared among processes.

→ Similarly to multiple threads sharing the same process space.

→ also useful for inter-process communication if sharing of read-write pages is allowed.

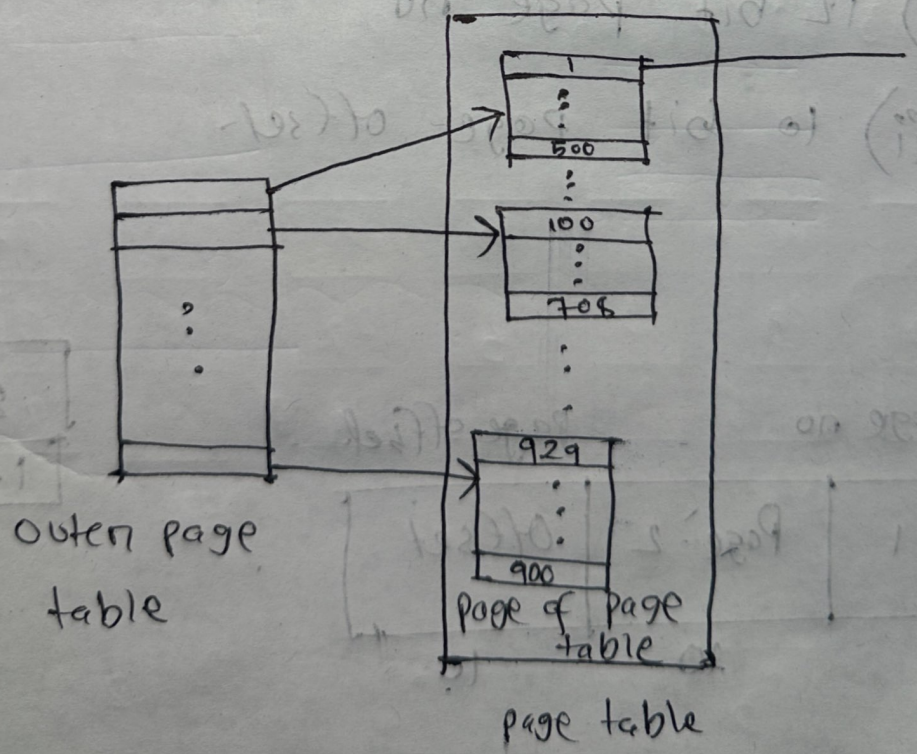
Hierarchical Page Table

~~→ Memory structures for paging can get huge using straight forward methods.~~

→ Break up the L.A space into multiple page table.

→ a simple technique is a two-level page table.

→



Two level paging

A logical address of 32 bit machine with 1 K page size) is divided into.

→ a page of 22 bits

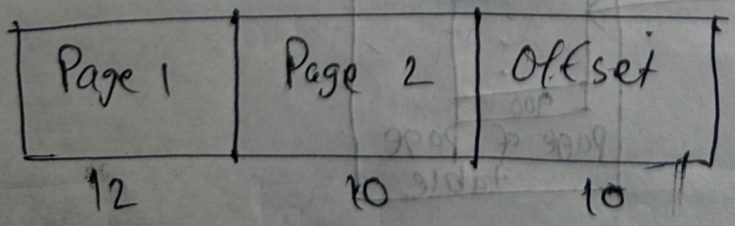
→ offset of 10 bits.

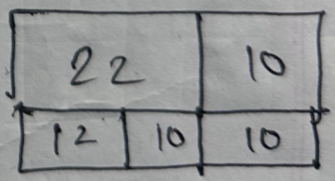
→ Page is again divided into

i) 12 bit page no

ii) 10 bit page offset.

Thus.





Video 03

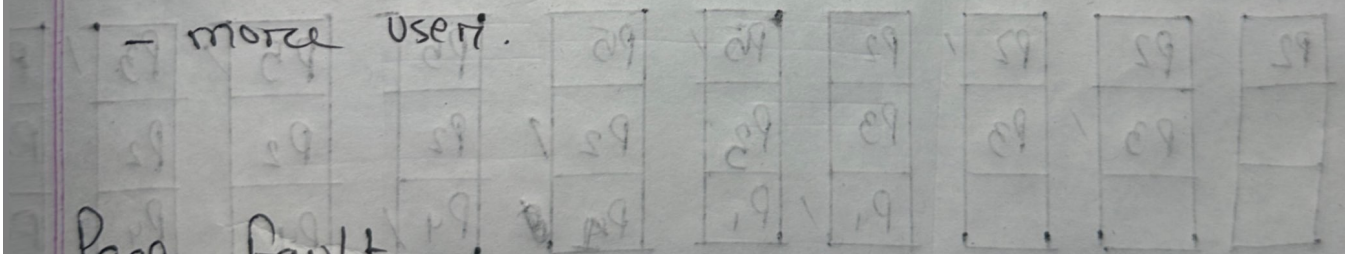
Virtual memory

The term virtual memory refers to something which appears to be present but actually it is not.

Demand Paging

Bring a page into memory only when it is needed

- less I/O needed
- less memory needed
- faster response
- more users.



Page Fault

A page fault occurs when a process tries to access a page that is not currently in main memory.

Cause: demand paging, page replacement, invalid access / protection

Page Replacement

3 Algo.

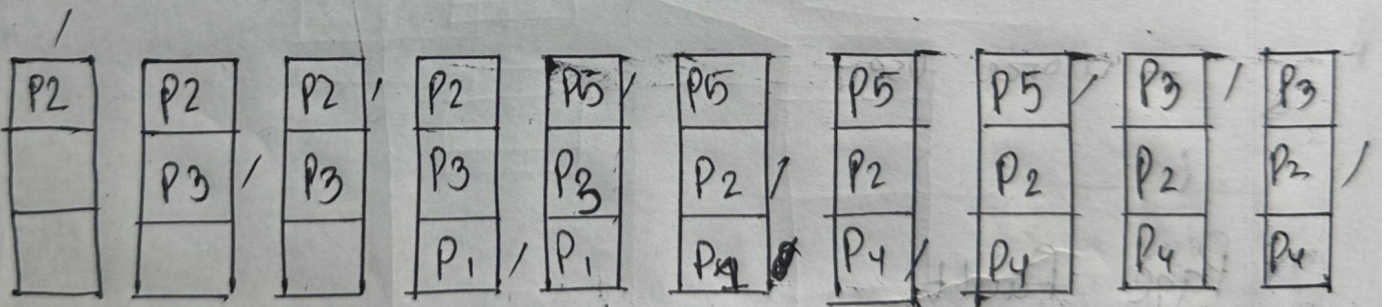
i) First in first out (FIFO)

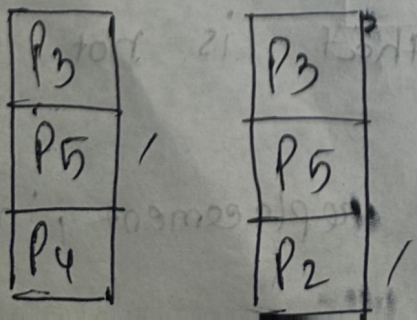
ii) LRU (Least Recently Used)

iii) OPT (optimal)

FIFO

time	1	2	3	4	5	6	7	8	9	10	11	12
page	P2	P3	P2	P1	P5	P2	P4	P5	P3	P2	P5	P2

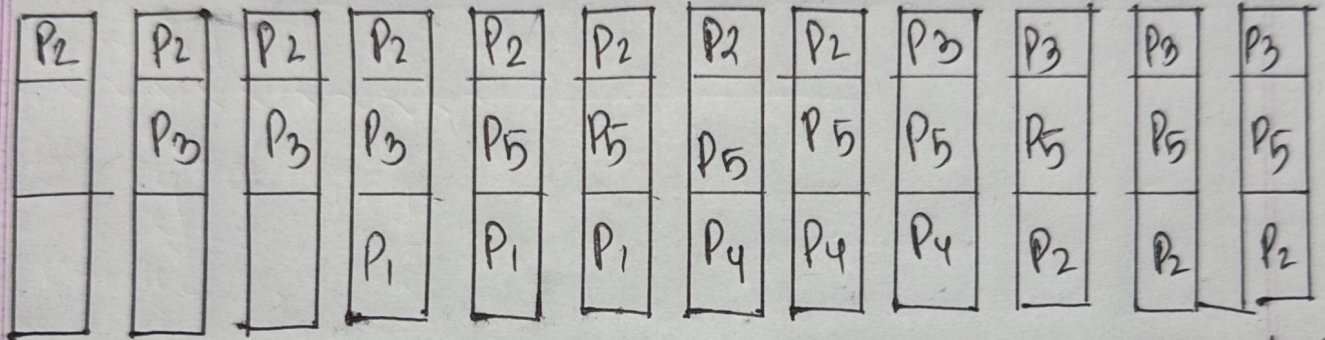




hit = 3/12.

LRU

time	1	2	3	4	5	6	7	8	9	10	11	12
page	P2	P3	P2	P1	P5	P2	P4	P5	P3	P2	P5	P2



recently যেটা কম use হয়েছে।

hit ratio = 5/12. > FIFO.

Optimal

time	1	2	3	4	5	6	7	8	9	10	11	12
page	P2	P3	P2	P1	P5	P2	P4	P5	P3	P2	P5	P2

replace the page that will be accessed at the farthest point in time.