Operating Systems

Unit - 3

Memory Management

* Registers vs. Main memory access

→ Registers built into the CPU are accessible within one clock cycle.

→ However, the main memory is accessed via transactions on the memory bus, which may take several clock cycles.

→ In such cases, the processor normally needs to stall, since it does not have the data required to complete the instruction that it is executing.

Remedy: Introduce cache between main memory & CPU registers

* Base and Limit Registers

→ Each process must have its own memory space.

→ To separate memory spaces, the range of legal addresses for that process must be determined.

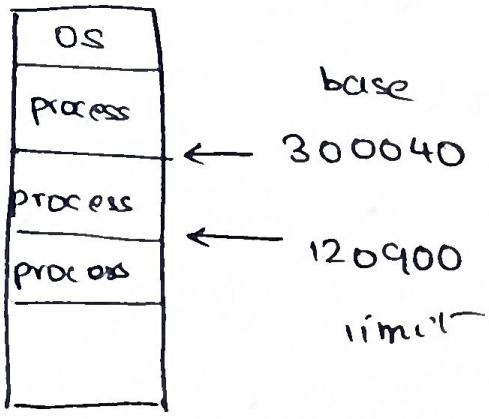
→ Done using 2 registers - base and limit register

Base Register: holds smallest legal physical memory address

Limit Register: Specifies ranges

* Hardware Address Protection

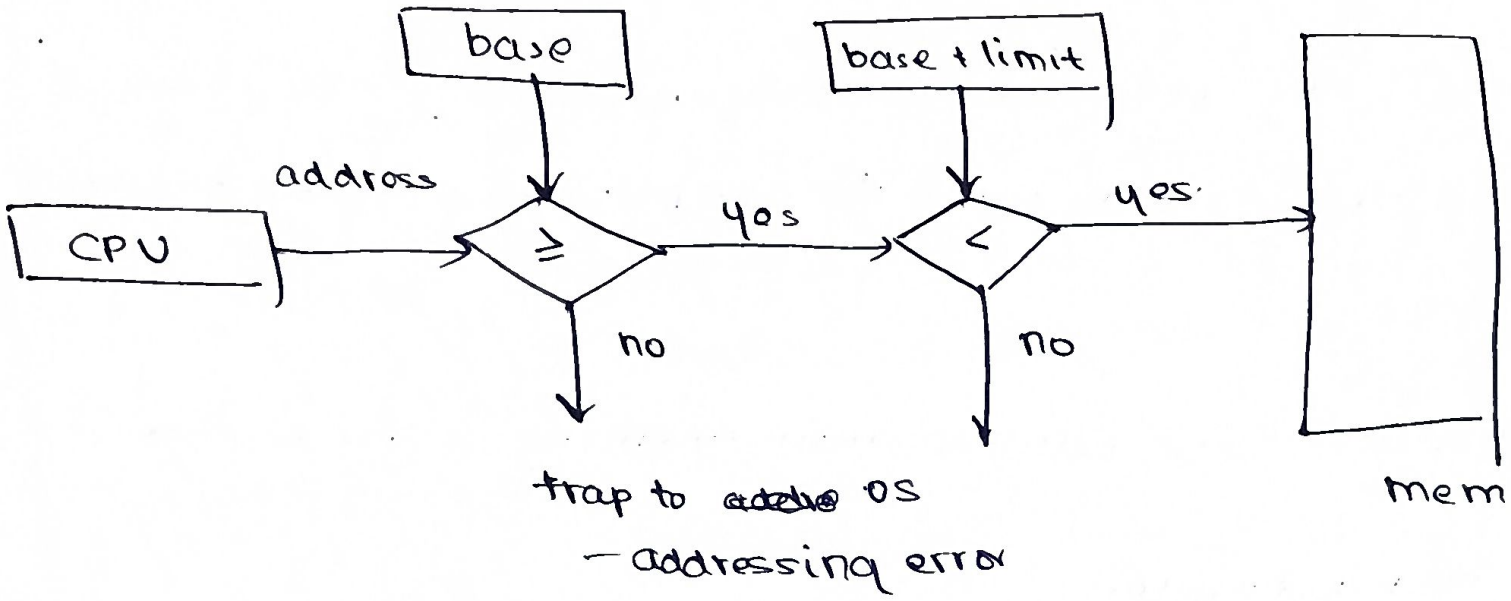
→ Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode w/ the registers.



→ An attempt by a user program to access OS memory or other user's memory results in a <u>trap</u> to the OS.

→ This scheme prevents the user program from accidentally or deliberately modifying the code or data structures of either the OS or other users.

→ The base and limit registers can be accessed only using a special priveleged instruction ⇒ only the OS can load the base & limit registers



* Address Binding

→ Programs on the disk are brought into the memory to execute from an input queue.

→ A user program goes through several steps, before being executed. Addresses are represented in different ways during these steps.

→ Initially, addresses in the source program are <u>symbolic</u>.

→ A compiler <u>binds symbolic addresses</u> to <u>relocatable addresses</u>

→ The <u>linkage editor</u> or loader binds the relocatable addresses to the <u>absolute address</u>.

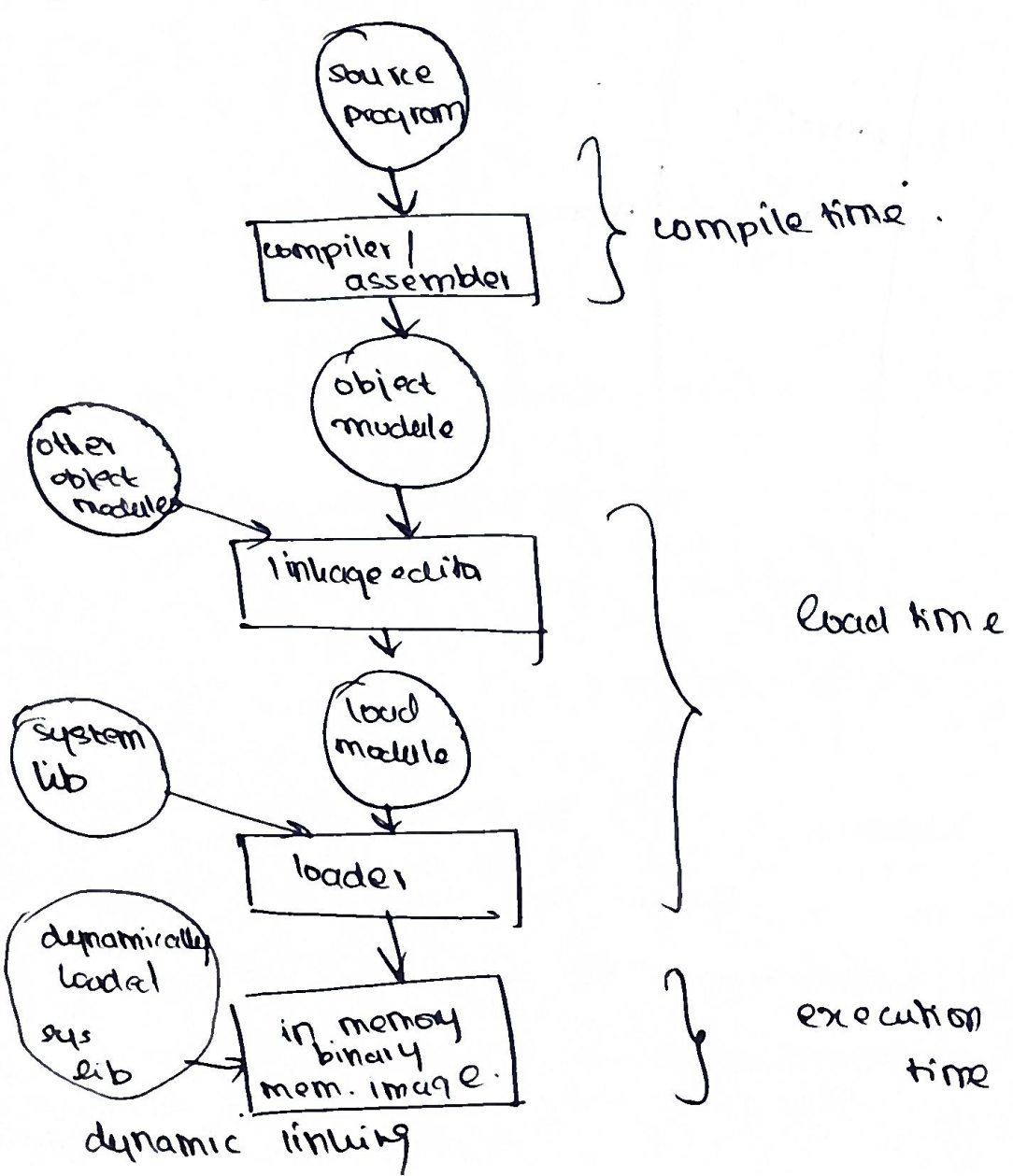
→ The binding of instructions is a mapping from one address pace to another.

→ Address binding of instructions and data to memory addresses can happen at 3 different stages.

(i) Compile Time : If the memory location is known a priori, absolute code can be generated, must recompile code if starting location changes.

(ii) Load Time: If it is not known at compile time where the process will reside in the memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time.

(iii) Execution Time: Binding delayed until runtime if the process can be moved during its execution from one memory segment to another (Need special hardware for this scheme to work)



* Logical vs. Physical Address Space

Logical address - address generated by the CPU

Physical address - address loaded into the memory address register

Logical address space - set of all logical addresses generated by a program

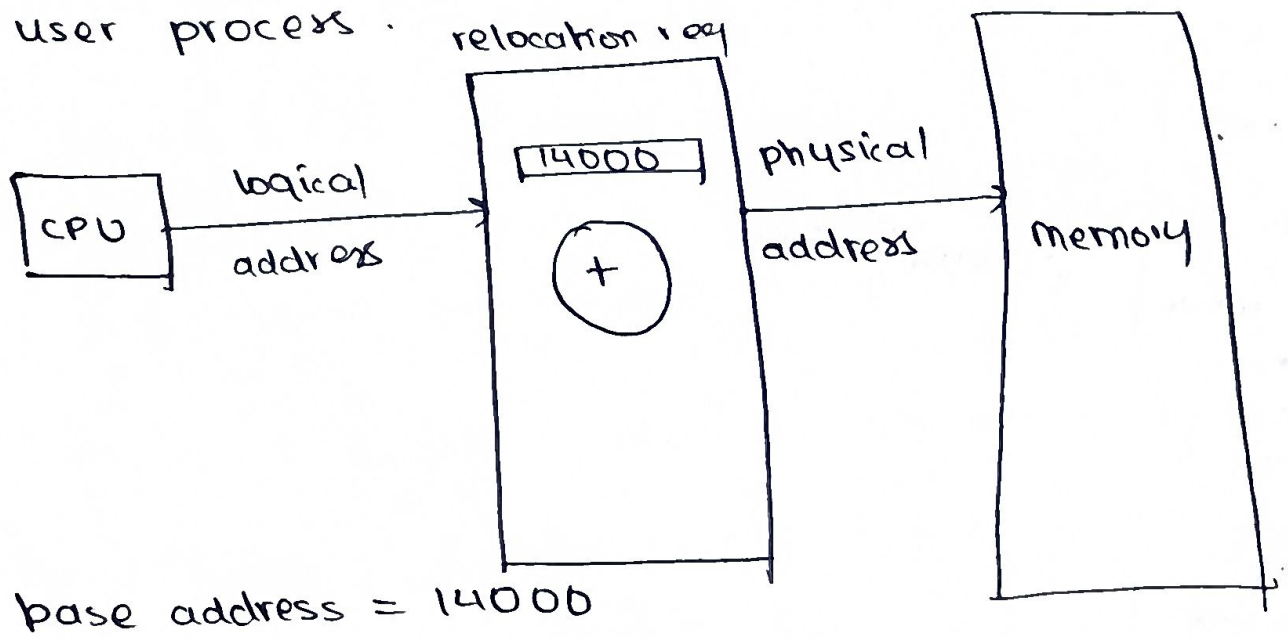
Physical address space - set of all physical addresses generated by a program

* Memory Management Unit (MMU)

→ run-time mapping from virtual to physical addresses done using MMU.

→ base register now called a relocation register

→ value in the relocation register added to every address generated by a user process.



eg. base address = 14000

Logical address = 346

physical address = 14346

→ In general : Logical address = 0 to max

physical addresses : R+0 to max+R

for base R.

* Dynamic loading

→ a routine is not loaded until it is called

→ all routines are kept on the disk in a relocatable load format.

→ The main program is loaded into the memory & executed.

→ When a routine needs to call another routine, the calling routine checks to see whether the other routine has been loaded.

→ If it has not, the relocatable linking loader loads the desired routine & updates the program's address.

→ does not require special support from the OS.

Advantage: a routine is loaded only when it is needed

→ useful when large amounts of code are needed to handle infrequently occurring cases.

* Dynamic linking

Static linking: system libraries and program code combined by the loader into the binary program image

dynamic linking: linking postponed until execution time

→ a small piece of code - stub - used to locate the appropriate memory resident library routine

→ stub checks if the routine is already in memory or otherwise loads it

→ stub replaces itself w/ the address of the routine & executes the routine

→ useful for using latest library versions - called shared libraries

* Swapping

→ A process can be swapped temporarily out of the memory to a backing store, and then be brought back into the memory.

→ Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system.

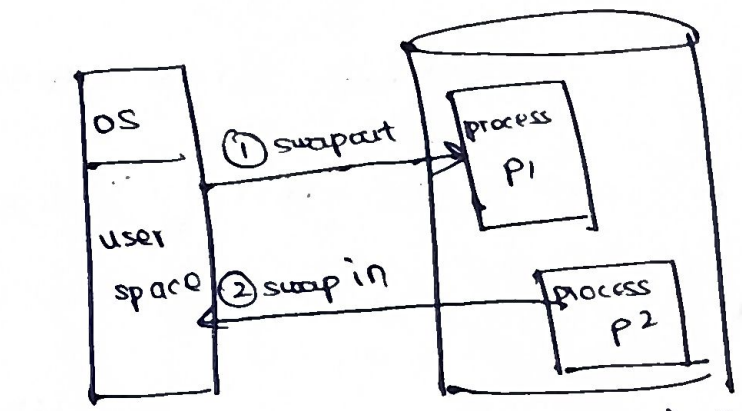
A. Standard Swapping

→ move processes between the main memory & backing store (backing store = a fast disk, large enough to accommodate copies of all memory images for all users).

→ system maintains a ready queue w/ all processes whose memory images are in the backing store.

→ context-switch time in standard swapping is very high

→ can reduce if the size of memory swapped is less (by knowing how much memory really is being used)



system calls to inform OS of memory use
request-memory()
release-memory()

* Constraints on Swapping

→ can swap only completely idle processes

→ must be careful when one wants to swap a process that is pending I/O

(can't swap as I/O would occur to wrong process)

Solutions: (i) never swap process w/ pending I/O

(ii) execute I/O operations only on to operating system buffers. Transfers between OS buffers & process memory happen when the process is to be swapped in.

→ called double buffering - adds overhead, since the data has to be copied from the kernel memory to user memory, before the user process can access it.

* Swapping in Modern Operating Systems

methods used:

(i) disable swapping

- starts only if the amount of free memory falls below a certain threshold.

- halted once free memory increases

(ii) Swap portions of processes - rather than entire processes, to decrease swap time.

* Swapping on Mobile Systems

→ Mobile systems do not support swapping in any form.

→ They use flash memory rather than spacious hard disks.

→ When free memory falls below a certain threshold, iOS asks applications to voluntarily relinquish allocated memory.

→ The read only data is removed from the system & late reloaded

→ Failure to free memory may result in application termination.

Android : adopts a strategy similar to iOS.

→ before terminating a process, Android writes its application state to the flash memory so that it can be quickly restarted

* Contiguous Memory Allocation

→ main memory must accomodate both the OS and the various user processes

→ Main memory must be allocated as efficiently as possible.

→ In contiguous memory allocation, the memory is divided into 2 partitions.

(i) The resident OS, is usually held in low memory along w/ the interrupt vector

(ii) User processes are held in high memory.

→ Each process is contained in a single contiguous section of memory.

* Memory Protection

→ use both a relocation register and limit registers.

→ Relocation register contains the value of the smallest physical address.

→ The limit register contains the range of logical addresses.

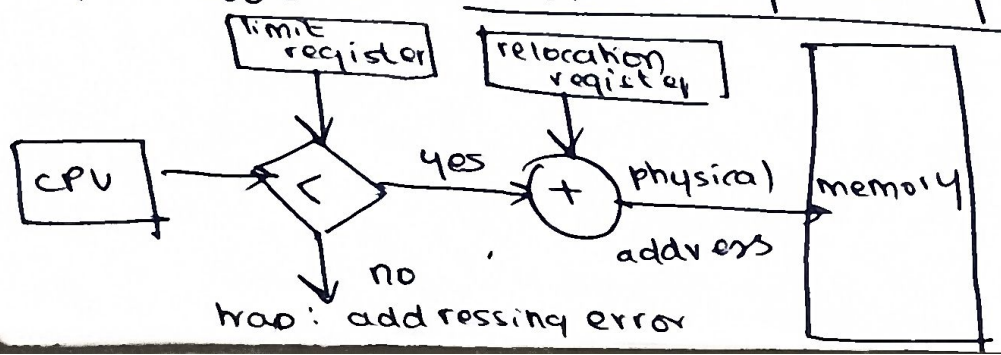
→ The MMU dynamically maps the logical address.

→ This allows the size of the OS to change dynamically.

→ The OS contains code & buffer space for drivers.

→ If a device driver is not commonly used, the space can be used for other purposes.

Such code = Transient operating system code



* Multiple-Partition Allocation

A. Fixed-Size Partitioning

→ memory divided into several fixed-size partitions

→ each partition may contain exactly one process.

→ degree of multiprogramming bound by the no. of partitions

B. Variable-Partitioning

→ The OS keeps a table indicating which parts of memory are available and which are occupied.

→ hole: a block of available memory, holes of various sizes are scattered throughout memory.

→ When a process arrives, and needs memory, the system searches in the set for a whole that is large enough for this process.

→ If the hole is too large, it is split again.

→ Adjacent holes are merged to form one larger whole.

→ The system continually checks for processes waiting for memory and whether the free/ recombined memory could satisfy the demands of any of the waiting processes.

* Dynamic Storage-Allocation Problem

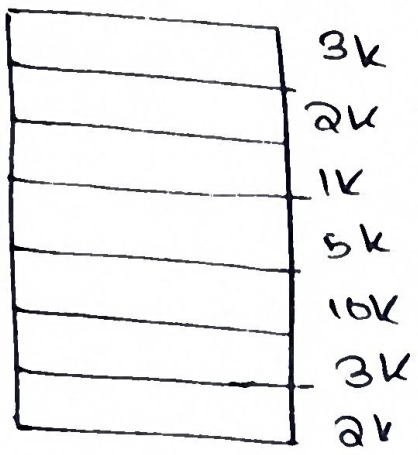
→ concerns how to satisfy a request of size n from a list of free holes

next-fit - like First Fit, but check from last allotted spot

Methods

A. First Fit - allocate first hole that is big enough
B. Best Fit - allocate the smallest hole that is big enough
C. Worst-fit - allocate the largest list, search entire list

Example : Allocate the following request using first fit, best fit, worst fit and next fit



A = 6K

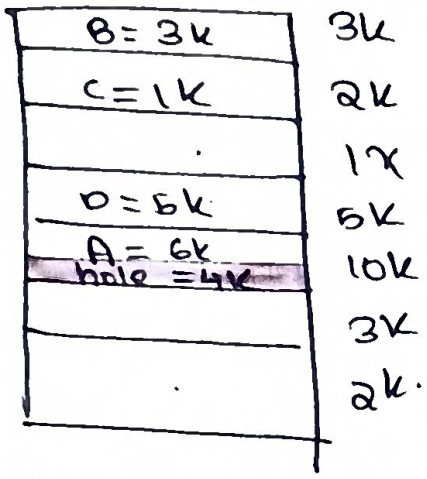
B = 3K

C = 1K

D = 5K

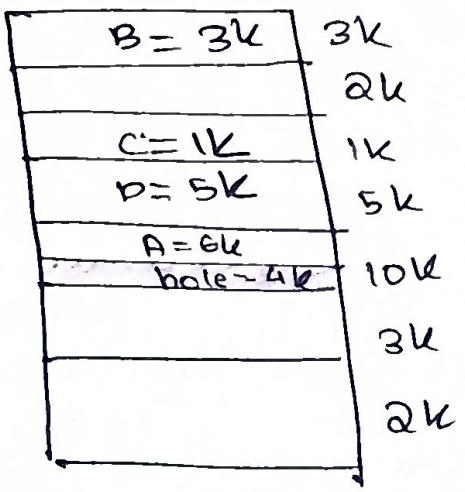
E = 8K

Ans FirstFit



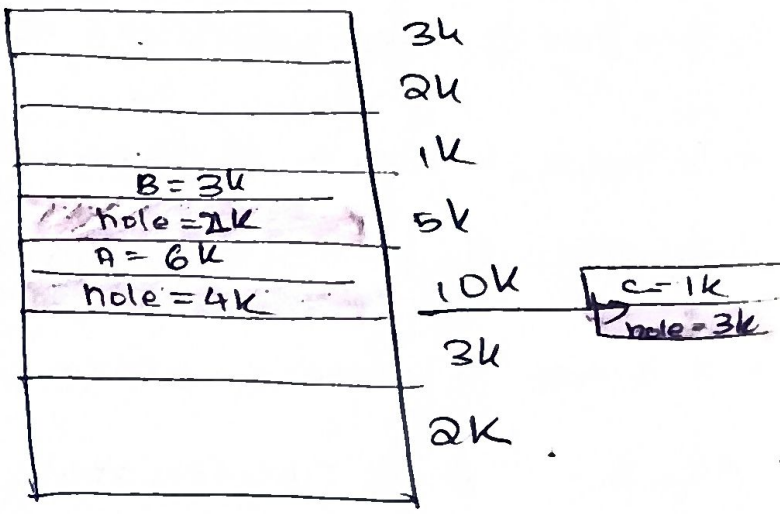
E not allocated

Best Fit



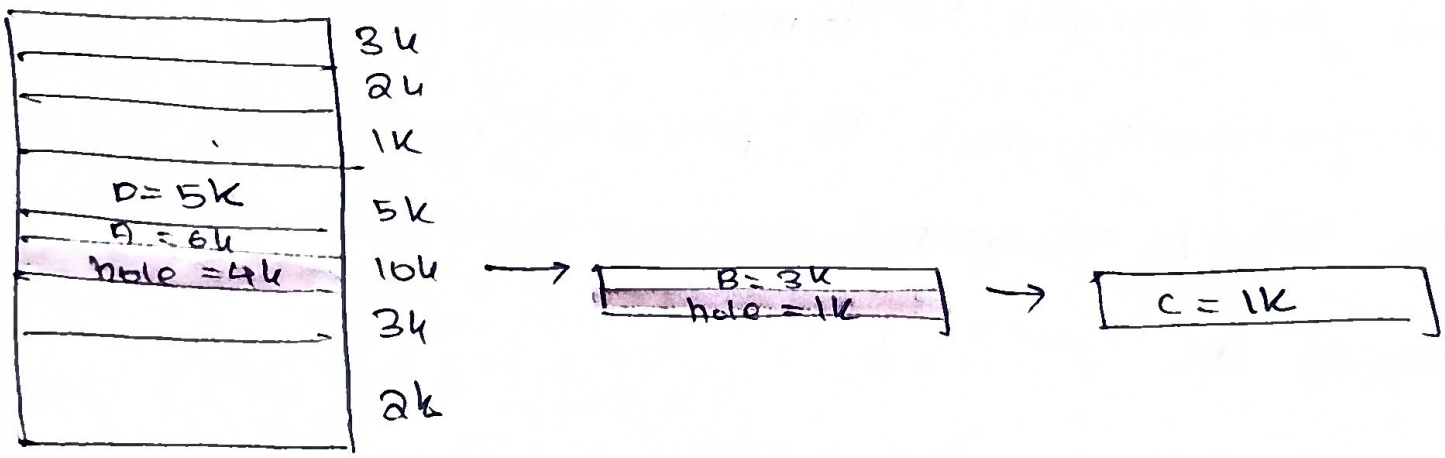
E not allocated

Worst Fit



D & E not allocated

Next fit



E not allocated.

* Fragmentation

A. External Fragmentation - happens when there is enough total memory to satisfy a request, but the available spaces are not contiguous.

* Internal Fragmentation - allocated memory may be slightly larger than requested memory, this size diff. is internal to a partition

50% Rule: Analysis of first fit reveals that even col some optimization, given N allocated blocks, another 0.5 N blocks will be lost to fragmentation. 1/3rd of memory be unusable. This property is known as the 50% rule.

* Compaction

→ can be used to reduce external fragmentation

→ shuffle memory contents to place all free memory together in one large block

→ compaction is possible only if relocation is dynamic, and is done at execution time.

→ If allowed, then move program and data, and change the base register to reflect the new address

→ simplest compaction algorithm - move all processes to one end of the memory, and all holes to the other end, producing one large hole of memory. This scheme is expensive

* Segmentation

→ a memory management scheme that supports the programmer view of memory

→ A logical address space is a collection of segments. Each segment has a name and a length

→ The addresses specify the segment name and the offset

→ The logical address is a tuple with 2 entries

<segment - no, offset>

Segmentation Architecture

* segment table - maps two-dimensional physical addresses. Each table entry has:

(i) base - contains the starting physical address where the segments reside in the memory

(ii) limit - specifies the length of the segment

* segment table base register (STBR) : points to the segment table's location in memory

* segment table length register (STLR) : indicates number of segments used by a program

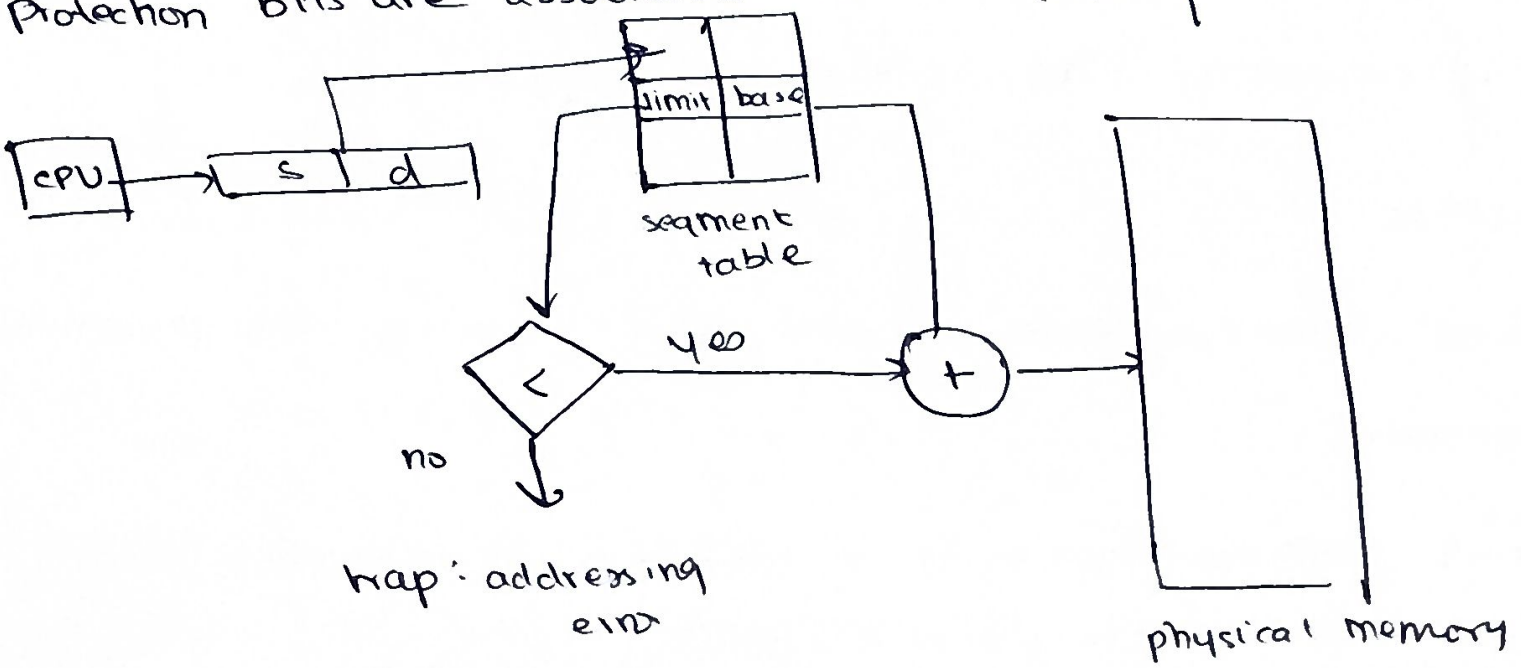
* protection

with each entry in the segment table associate:

(i) validation bit = 0 ⇒ illegal segment

(ii) read/write/execute privileges

→ Protection bits are associated with segments



Example: Consider the following segments along with their base and length values.

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

(a) 0, 430

physical addr = 219 + 430

= 649

(b) 1, 10

physical addr = 10 + 2300

= 2310

(c) 2, 500

physical addr = 90 + 500

= 590 > length + base

⇒ segmentation fault

(d) 3, 400

physical address = 1327 + 400 = 1727

(e) 4, 112

physical address = 1952 + 112 ≥ length + base

⇒ segmentation fault

* Paging

→ a method of non-contiguous memory allocation

→ The process is divided into a no. of partitions called pages

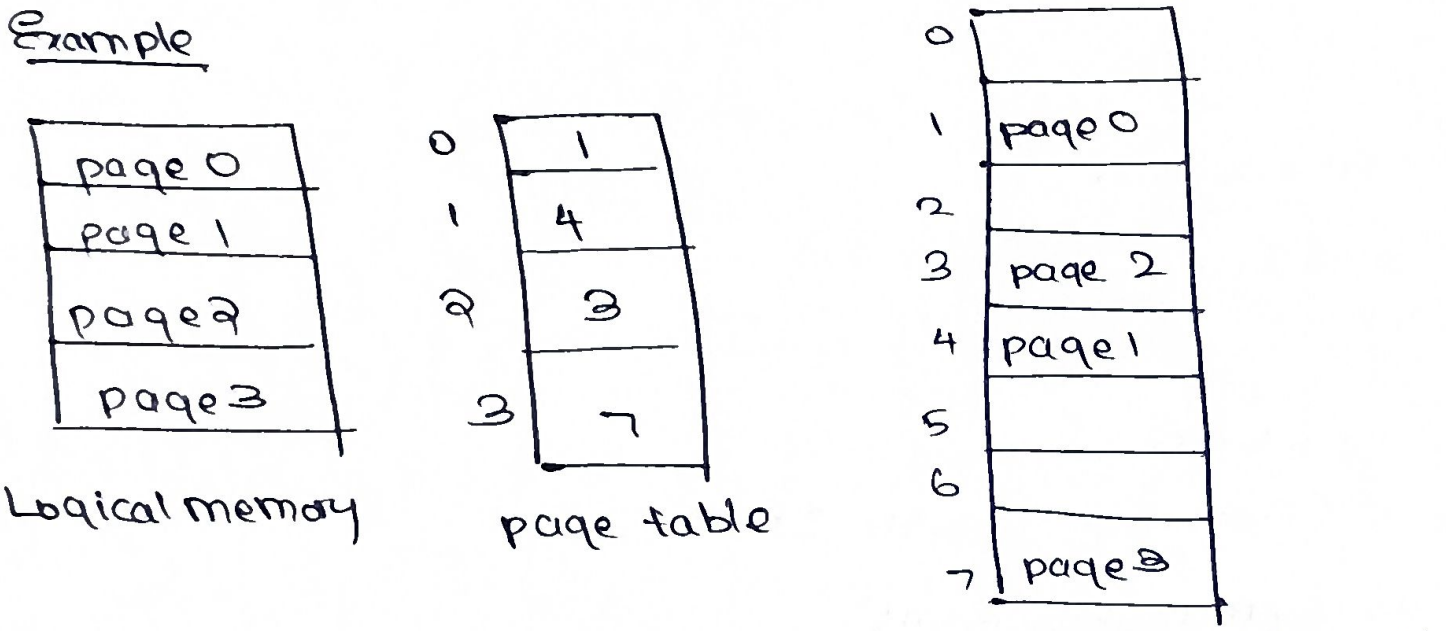
The size of each page must be the same in the logical memory

→ In the main memory (physical memory), the division is in the form of frames. size of each frame.

→ Page table helps link the logical memory and physical memory.

→ has 2 parts <pageno, frame no>

Example

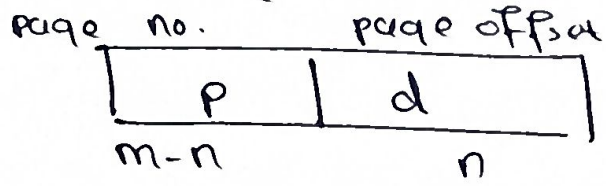


Address Translation Scheme

The logical address generated by the CPU is divided into:

Page no (p) - indexes into a page table, which contains base address of each page in physical memory

page offset (d) - combined with base address to define the physical memory address that is sent to the memory unit



for
logical address space: 2^m
page size = 2^n

* Paging Hardware

→ CPU generates a logical address for the instruction to be executed.

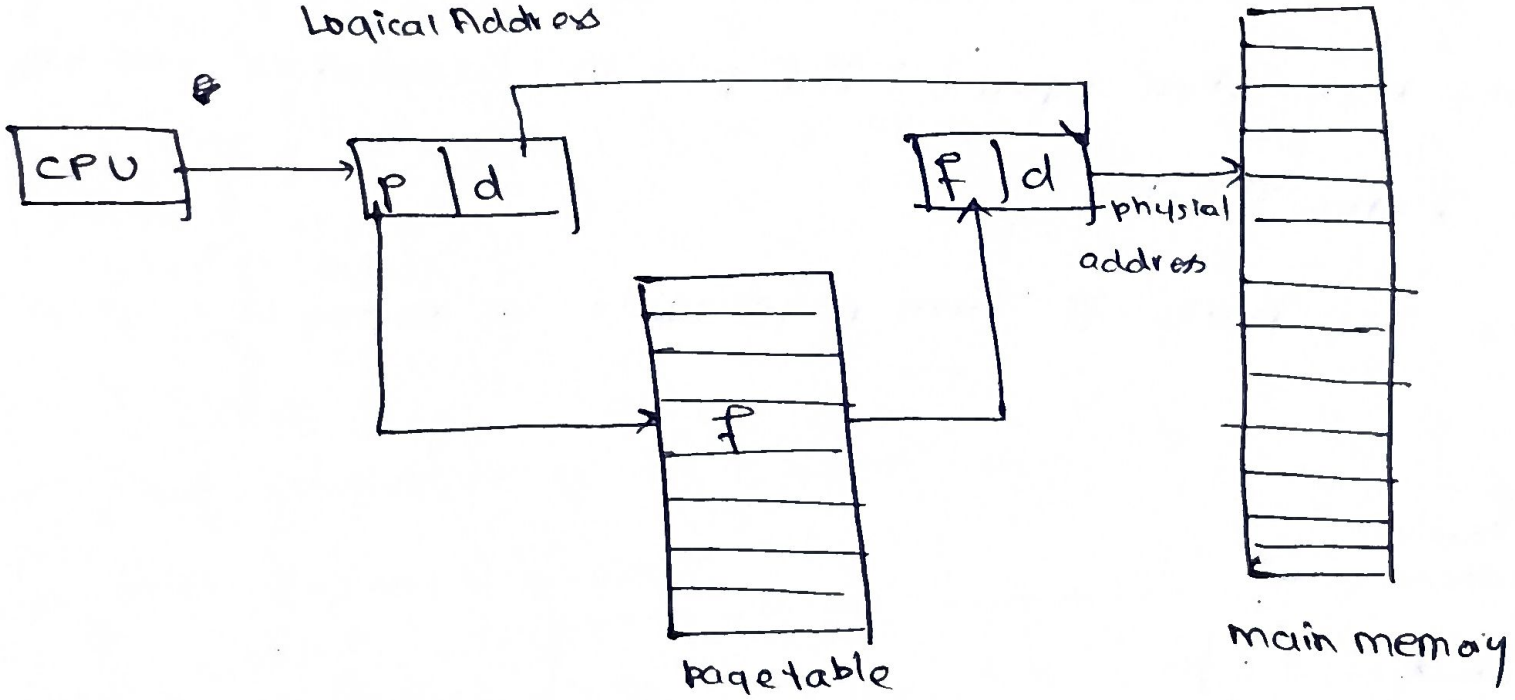
→ logical address is divided, | p | d |, where

p = page no.

d = displacement.

→ The page no is given as input to the page table. The page table produces the corresponding frame no

→ If the frame no is combined with the offset d, it gives the exact physical address



* Implementation of Page Table

→ The page table is kept in the main memory,

→ The page-table base register (PTBR) points to the page table

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

→ In this scheme, every data / instruction access requires 2 memory accesses - one for the page table and one for data/instruction.

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

→ Some TLBs store address-space identifiers (ASIDs) in each TLB entry - uniquely identifies each process to provide address-space protection for that process

→ TLBs are very small, on a TLB miss, a value is loaded into the TLB for faster access next time.

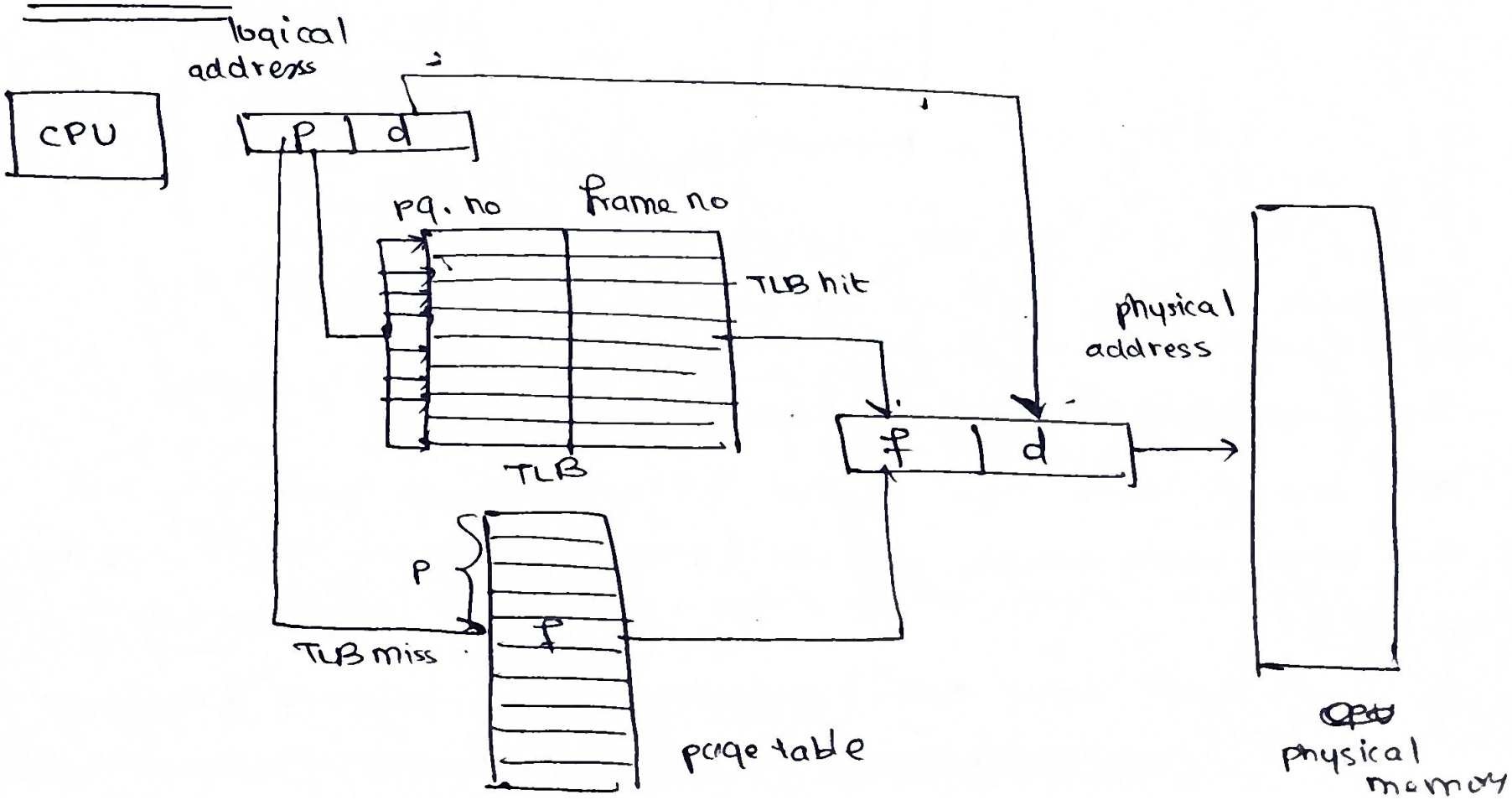
Associative memory

→ The TLB has 2 parts: page # & frame #

→ Address translation (p,d): If p is in associative register get the frame # out.

→ Otherwise get frame # from page table in memory.

Hardware



* Effective Access Time

→ associative lookup = ε time unit

→ hit ratio = α

↳ defines the percentage of times that a page no. is found in the associative registers

→ To find the effective memory-access time, weight each case by its probability.

EAT = Hit(TLB + MM) + miss(TLB + PT + MM)
HIM

Example1 : $\alpha = 80\%$, $\varepsilon = 20$ns for TLB search, 100 ns for memory access.

EAT = 0.8 * (20 + 100) + 0.2 * (100 + 20 + 100)

= 0.8 * 120 + 0.2 * 220

= 140ns

Example2: Consider a hit ratio of 99%, 20ns for TLB search and 100ns for memory access

EAT = Hit(TLB + MM) + miss(TLB + PT + MM)

= 0.99 (20 + 100) + 0.1 (20 + 100 + 100)

= 0.99 * 120 + 0.01 * 220

= 121 ns

Example3: Consider a paging system with the page table stored in the memory. If a memory reference takes 50 nanoseconds how long does a paged memory reference take? If we add TLBs and

75%. of all page-table references are found in the TLBs, what is the effective memory reference time? Assume that finding a page-table entry in the TLBs takes 2ns, if the entry is present.

Solution

Time for paged memory reference = time to access page table

+ time to access main memory

= 50+50 = 100ns

α = 75%.

ε_p = 2ns

EAT = hit (TLB + MM) + miss (TLB + PT + MM)

= 0.75 (2 + 50) + 0.25 (2 + 100)

= 0.75 * 52 + 0.25 * 102

= 64.5 nanoseconds

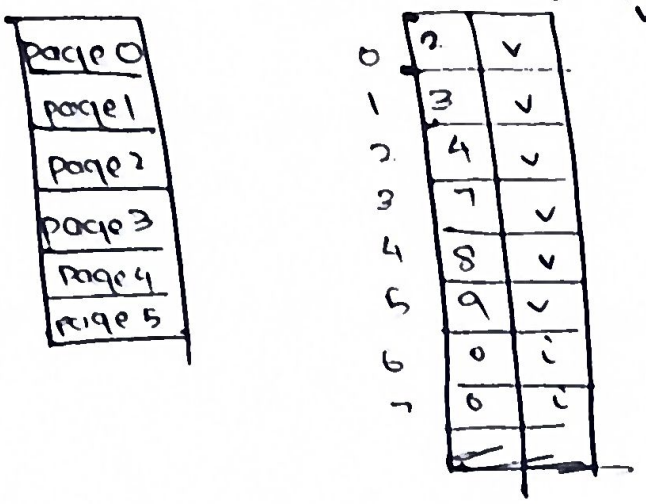
* Memory Protection

→ Is implemented by associating a protection bit with each frame to indicate if read-only or read-write access is allowed

→ a valid - invalid bit is attached to each entry in the page table

(i) 'valid' indicates that the associated page is in the process' logical address space, and is thus a legal page

(ii) 'invalid' indicates that the page is not in the process' logical address space.



* Shared Pages

→ Paging can help share common code.

→ Reentrant / pure code can be shared

↓

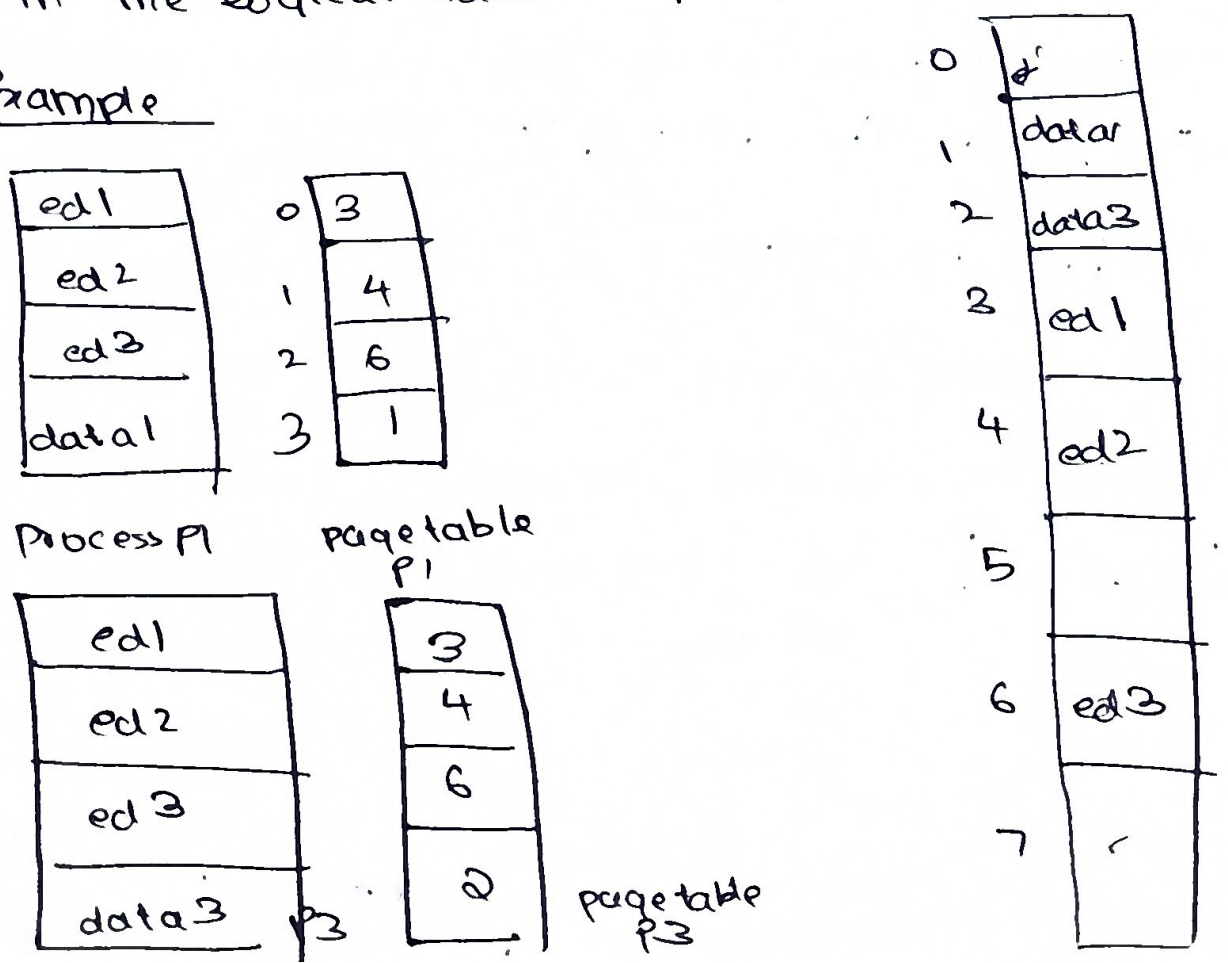
non-self-modifying code - it never changes during execution.

→ Thus, 2 or more processes can execute the same code at the same time.

→ Each process has its own copy of registers & data storage to hold data for the process' execution.

→ The pages for the private code and data can appear anywhere in the logical address space

Example



* Structuring the page table

→ memory structure for paging can get huge using straight forward methods

→ It is not desirable to allocate such large blocks contiguously in the main memory.

→ Common techniques used:

(i) Hierarchical paging

(ii) Hashed page tables

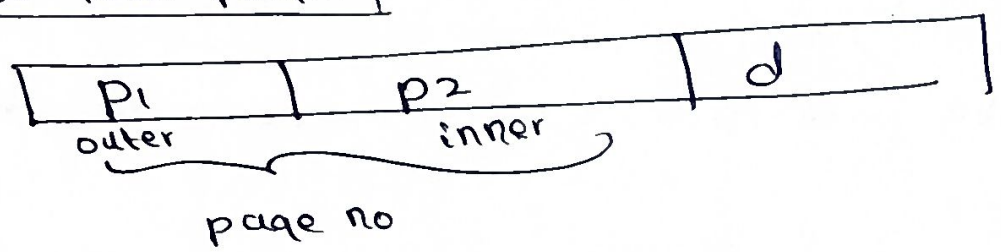
(iii) Inverted page tables

A. Hierarchical Paging

→ The logical address space is broken up into multible page tables.

→ The page table itself is paged.

Two-level paging

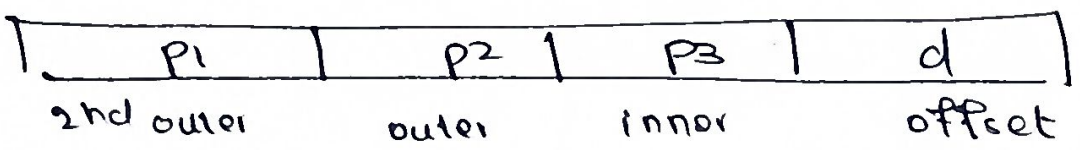


→ p1 is an index to the outer page no

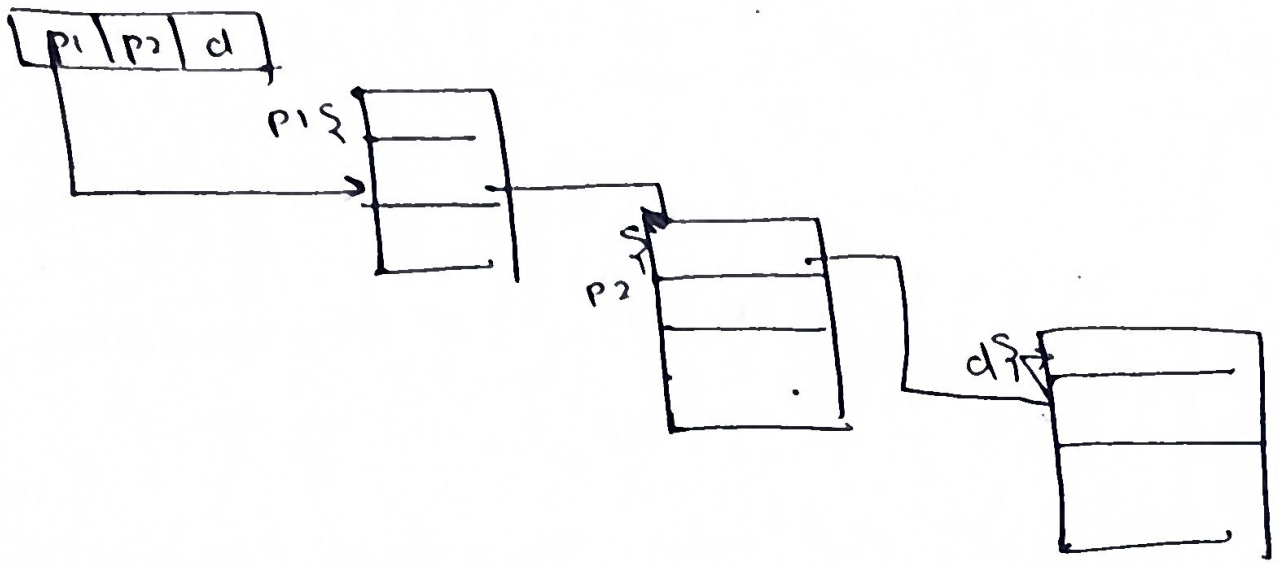
→ p2 is the displacement within the page of the inner page table

→ Known as forward-mapped page table.

Three-level paging



* Address Translation Scheme for Hierarchical Paging



B. Hashed Page Tables

→ CPU generates a logical address - w/ page no. & offset

→ The page no. is given as i/p to the hash function

→ The hash fn. produces a hash value, that is mapped to a location in the hash table.

→ Each entry in hash table is a linked list (separate chaining)

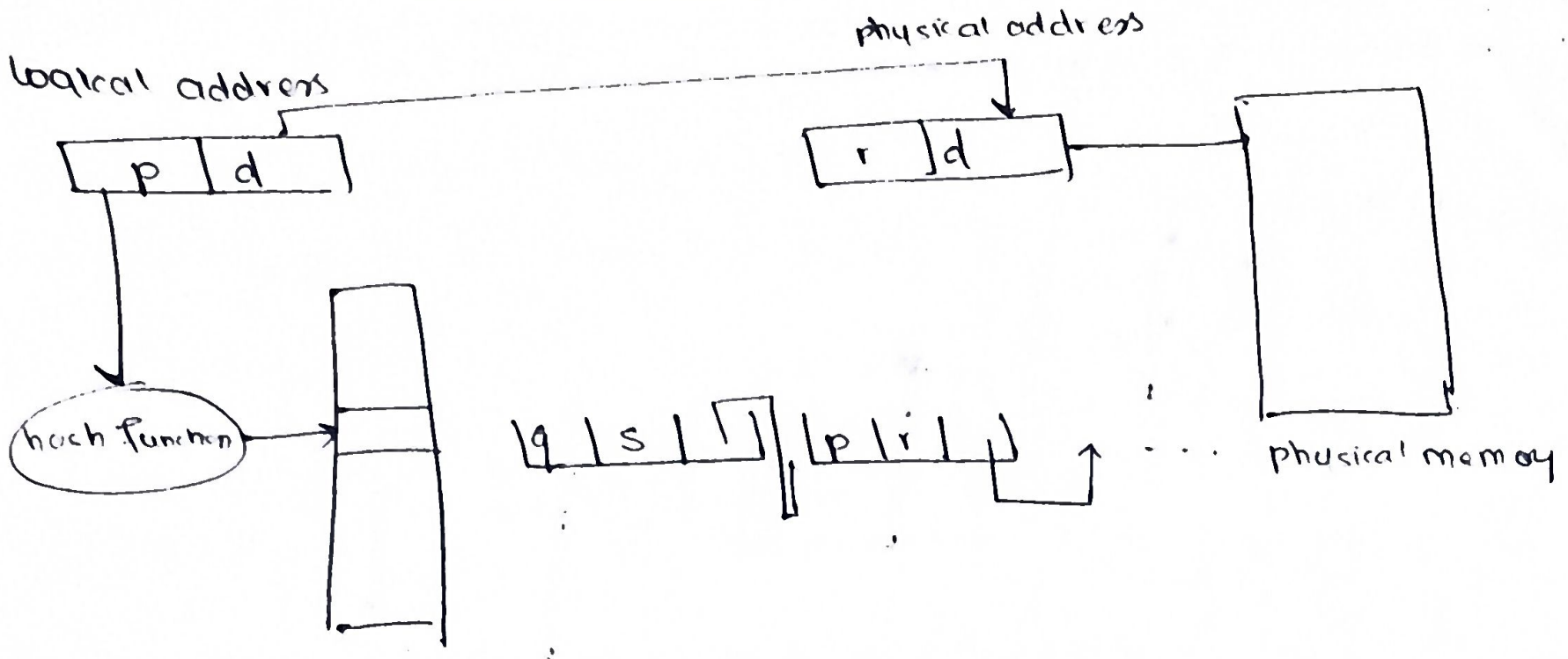
→ Each entry of the linked list has the following structure:

page no.	frame no	nextaddr

→ The virtual page no. is compared with field1 in the first element of the linked list.

→ If there is a match, the corresponding page frame (field2) is to form the desired physical address.

→ If there is no match, subsequent entries in the linked list are searched for a matching virtual page no



* clustered page tables

→ a variation of hashed page tables

→ each entry in the hash table refers to several pages, rather than a single page.

→ ∴, a single page-table entry can store the mappings for multiple physical page-frames.

→ Clustered page tables are useful for sparse address spaces where memory references are non-contiguous & scattered throughout the address space

* Inverted Page Tables

→ Usually, each process has an associated page table.

→ A drawback of this method is that each page table may consist of millions of entries - may consume large amounts of physical memory

→ Inverted page table - has one entry for each real page/frame

→ Each entry consists of the virtual address of the page, along with the information about the process that owns the page.

→ Inverted page tables have an address-space identifier. This allows a logical page of a particular process to be mapped to the corresponding physical page frame.

→ The virtual address is in the form of:

<process-id, page-number, offset>

→ Each inverted page table entry is a pair <process-id, page-no>

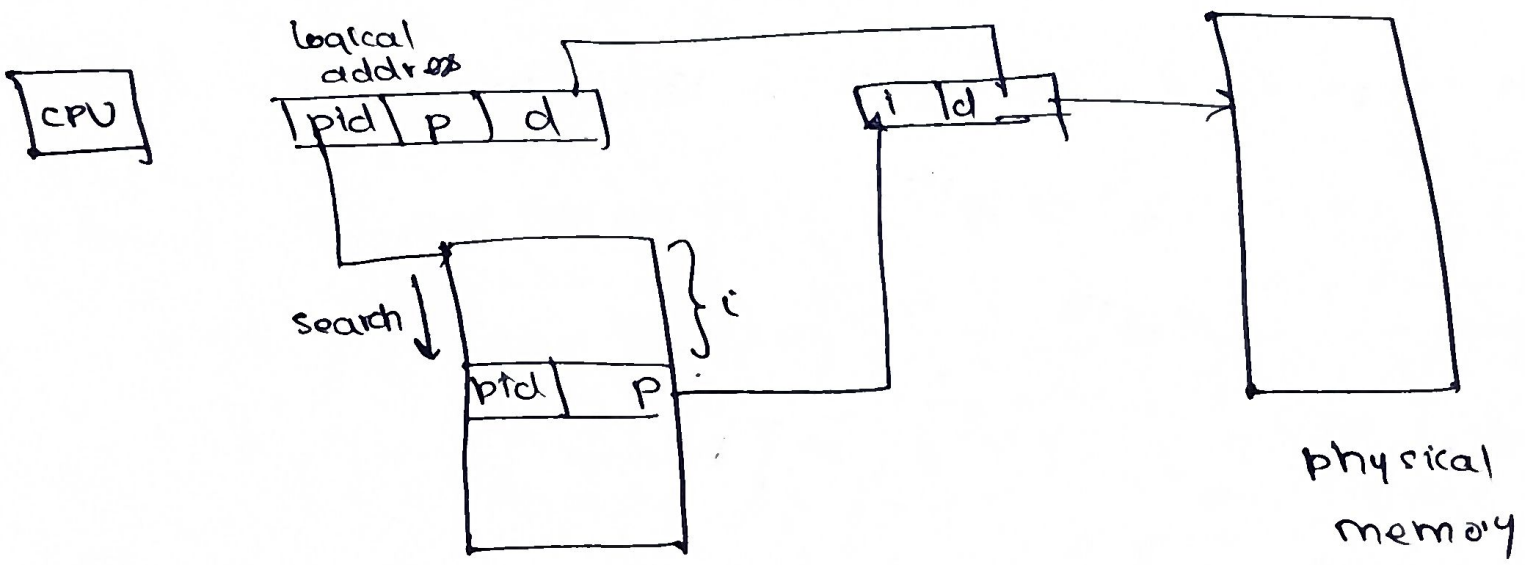
→ When a memory reference occurs, part of the virtual address is presented to the memory subsystem.

→ The inverted page table is then searched for a match, then the corresponding physical address <i, offset> is generated.

Disadvantage

→ increased time to search table (use hash tables)

→ difficult to implement shared memory (have one mapping of virtual address to shared physical address)



Numericals

Points to Note:

- m-bit processor ⇒ logical address = m bits long
- size of logical address space = 2^m

  page size = 2^n (bytes or words)

- page no = m-n
- page offset = n

Q1. Consider the logical address space of 8 pages of 1024 words mapped onto the physical memory of 32 frames

(i) How many bits are there in the physical address?

(ii) How many bits are there in the logical address?

Ans: page size = 1024 words

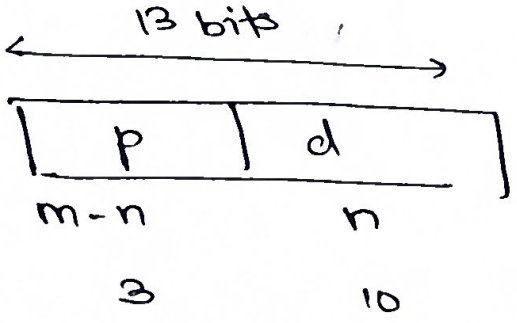
$= 2^{10}$ ⇒ $n = 10$

logical address space = 8 pages × 1024 words

$= 2^3 \times 2^{10}$

$= 2^{13}$ ⇒ $2^m = 2^{13}$

⇒ $m = 13$



size of physical address space:

$= 32 \times 2^{10}$

$= 2^5 \times 2^{10}$

$= 2^{15}$

$\Rightarrow$ logical address space = 13 bits

physical address space = 15 bits

Q) Given a 32 bit processor, with a page size of 1024 bytes. Find

(i) size of logical address

(ii) no. of bits to represent page no. & offset

(iii) max. size of logical address space

(iv) max. no. of pages in logical address space

(v) max length of page table of a process

Ans: (i) size of logical address space = 2^m

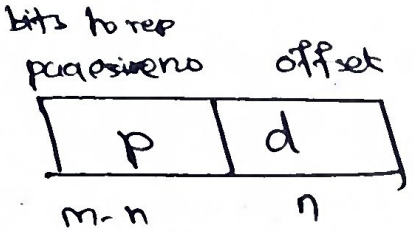
32 bit processor = size of logical address space

where $\boxed{m = 32}$

(ii) no. of bits used to represent page no. and offset

page size = 1024 bytes

$= 2^{10}$ bytes $\Rightarrow$ $\boxed{n = 10}$



bits to represent pg. no = m-n = 22 bits

bits to represent offset = n = 10 bits

(iii) max. size of logical address space

$= 2^{32}$ bytes

$= 2^2 \times 2^{30}$

$= \underline{4G \text{ bytes}}$

$= \underline{4GB}$

$2^{10} = 1kB$

$2^{20} = 1MB$

$2^{30} = 1GB$

(iv) max no. of pages in logical address space =

2^{m-n}

$= 2^{32-10} = 2^{22}$

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

$= 4MB$

(v) max length of page table

= max no. of pages = 4M entries

= 4 million entries

Q3 Consider a machine with 64 MB physical memory & a 32 bit virtual memory. If the page size is 4kB, what is the approximate size of the page table?

Ans: physical memory = 64MB

Logical address = 32 bits long

page size = 4kB

Convert into bytes first

physical memory = 64MB = $2^{20} \times 2^6$

$= 2^{26}$ bytes

page size = $2^2 \times 2^{10}$ bytes

$= 2^{12}$ bytes $\quad \boxed{n=12}$

virtual address space = 32 bits long $\Rightarrow$ 2^{32} bytes

$\boxed{m=32}$

p	d
m-n	n

size of logical address space = pages x words (page size)

2^{32} = pages x 2^{12}

pages = $\frac{2^{32}}{2^{12}}$ = $\boxed{2^{20}}$ entries

~~Total no of frames = page size x~~

physical memory size = page size x no. of frames

frame size = $\frac{2^{26}}{2^{12}}$ = 2^{14} frames

page table size = 2^{20} entries x 14 bytes

$\approx 2^{20} \times$ 2 bytes = 2MB

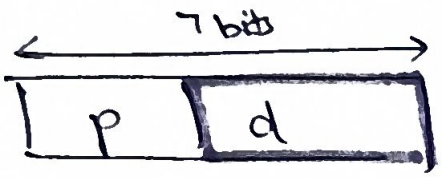
Q4 Consider a system which has logical address = 7 bits, physical address = 6 bits, page size = 8 words. Calculate the no. of pages and no. of frames

page size
page offset are
the same thing

Ans:

logical address = 7 bits

logical address size = 2^7 $\boxed{m = 7}$



pagesize = 8 words

= 8 bytes

= 2^3 bytes

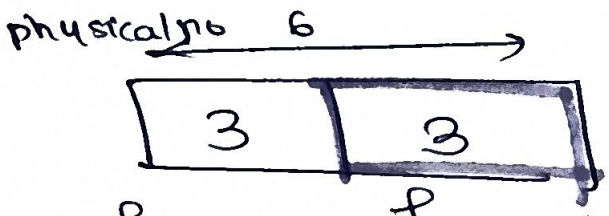
$\boxed{n = 3}$

no. of bits used to represent page no = 4 bits

no. of page = $2^4 = 16$

physical address = 6 bits

total size = 2^6 = 64 bytes

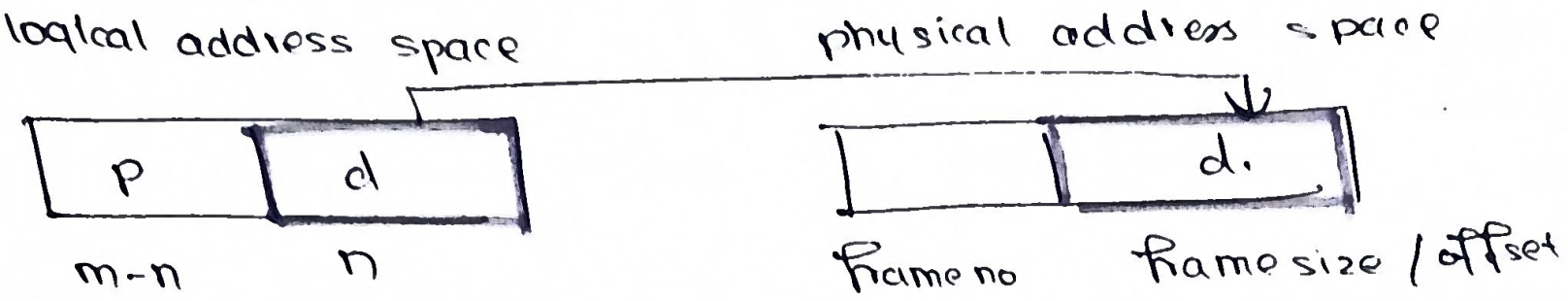


frame size = page size

bits used to represent frames = 3

total no. of frames = $2^3 = 8$

Consolidated Formulae



page no { page size / offset } represented in words somehow

page size / offset = frame size / offset

logical address = m bits
is represented in

Total logical address size = 2^m

no. of words = no. of bits used to represent a page = n

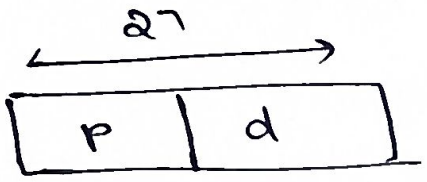
size of page = 2^n

size of logical address = pages x page size

size of physical address = frames x frame size

Q5. Consider a system in which the logical address is 27 bits and the physical address requires 21 bits. The page size is 4K words. Calculate the no. of pages and no. of ~~words~~ frames

Ans logical address = 27 bits $\boxed{m = 27}$



page size = 4k words
$= 2^2 \times 2^{10} = 2^{12}$ bytes

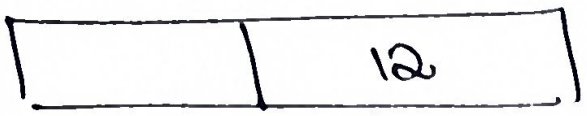
$\boxed{n = 12}$

no. of bits used to rep pages = m-n = 15

$\boxed{\text{total no. of pages} = 2^{15}}$

no. of frames = ?

Physical memory



frame no. frame size / offset

representation of physical address is in 21 bits

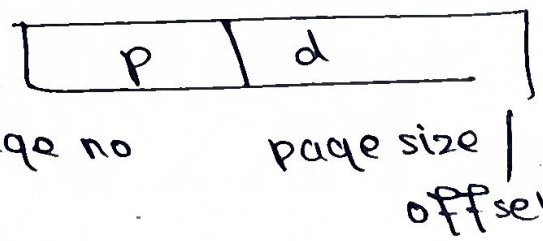
no. of frames = 21 - 12 = 9 bits

Total frame size = 2^9

= 512 frames

Q6 Consider a system having a page table of 4k entries, logical address of 29 bits. How many bits does the physical address require if the system is of 512 frames ?

Logical address



no. of bits used to represent logical address = 29 bits

$m = 29$

no. of entries ~~= no. of words~~ = no. of pages = $4k = 2^2 \times 2^{10}$

$= 2^{12}$

$m - n = 12$

page size = $n = 29 - 12 = 17$

Physical address

9	17

frame no. frame size

no. of frames = 512 = 2^9 => 9 frames

frame size = page size = 17

=> Total bits in physical address = 26 bits

Q7 Consider a logical address space of 64 pages of 1024 words each mapped onto a physical memory of 32 frames. How many bits are there in the logical address & physical address?

Logical address

p	d

no. of pages — page size/ offset

Words = page size

m-n = no. of pages

$2^{m-n} = 2^6$

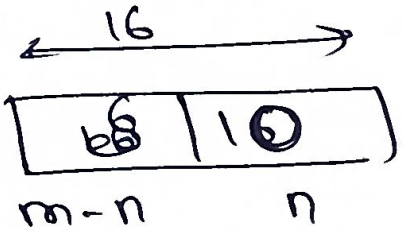
m-n = 6

no. of words = page size

= 1024 words

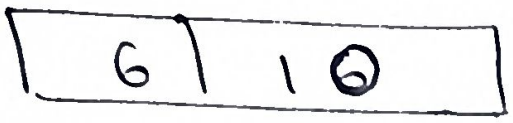
$= 2^{10}$

=> 10 bits n=10



=> Total logical address = 16 bits

Physical memory



frame no. frame size

physical frames = 32

$= 2^5$ frames

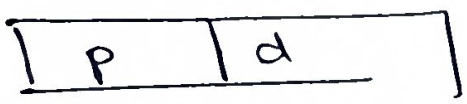
no. of bits to represent frames = 5

Total no. of bits in physical address = 15

Q8 Consider a memory system w/. 128 kB of logical address space and 512 kB of physical address, the page size is 16 kB Calculate the no. of bits needed to represent the logical address and the no. of bits needed to represent the physical address

Calculate the no. of pages and the no. of frames

Logical address



page no. page size offset

no. of bits in logical addr = m

$= 128kB = 2^7 \times 2^{10}$

$= 2^{17}$

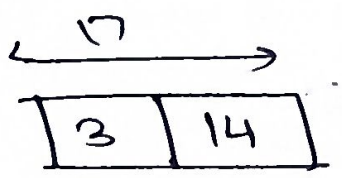
m = 17

no. of bits needed to represent logical address = 17

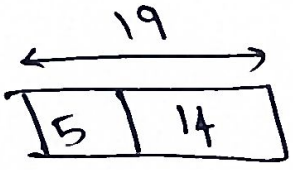
page size = 16 kB = $2^4 \times 2^{10}$

$= 2^{14}$ n = 14

LA



PA



physical address = $512\ 16kB = 2^9 \times 2^{10} = 2^{19}$

Total = 19 bit

no. of bits needed to represent physical address = 19

no. of pages = 2^3 = 8 pages

no. of frames = 2^5 = 32 frames

Q9 Consider the page table where

page 0 → frame 7

page 1 → frame 1

page 2 → frame 15

page 3 → frame 31

The remaining pages are not loaded in the memory. Convert the following memory locations from the logical address space into the physical address space

(i) 1024

(ii) 11000

(iii) 31588

(iv) 50000

(v) 123458

page size = 16kB

Formula

p = mem. addr / page size

d = mem. addr % page size

PA = frame no x frame size + d

Ans. (i) 1024

p = 1024 / 16 x 1024 = 1/16 = 0 ⇒ frame 7

d = 1024 % (16 x 1024) = 1024

PA = 7 x (16 x 1024) + 1024

= 115712

(ii) 11000

p = 11000 / (16 x 1024) = 0

d = 11000 % (16 x 1024) = 11000

PA = 7 x (16 x 1024) + 11000

= 125688

(iii) 123458

p = 123458 / (16 x 1024) = 7

↳ not in page table

⇒ page fault

(iv) p = 31588 / (16 x 1024) = 1 → frame 1

d = 15204

PA = 1 x (16 x 1024) + 15204

= 31588

(v) 50.000

p = 50000 / (16 x 1024) = 3 ⇒ frame 31

d = 848

31 x (16 x 1024) + 848

= 508752

Virtual Memory

* Virtual memory - separation of user logical memory from the physical memory

→ only a part of the program needs to be in memory for execution

→ logical address space can be much larger than physical address space.

Benefits: (i) Address spaces shared by multiple processes
(ii) more programs running concurrently
(iii) less I/O to load or swap processes

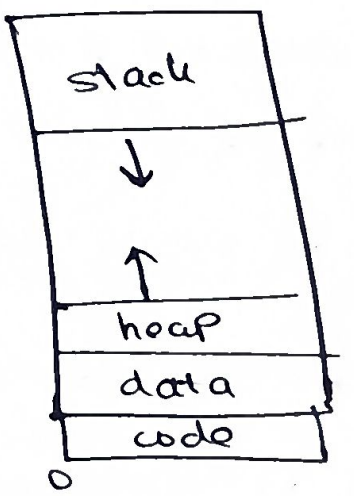
* virtual address space

→ logical view of how a process is stored in the memory

→ usually start at address 0, contiguous addresses until the end of space

→ physical memory organized in page frames

→ MMU maps logical to physical addr.



• Virtual memory can be implemented as:
(i) demand paging
(ii) demand segmentation

→ The heap in the virtual address space grows upwards in memory, as it is used for dynamic memory allocation.

→ The stack grows downwards in memory.

→ The large blank space or hole between the heap & the

stack is part of the virtual address space, but will require actual physical pages only if the heap or stack grows.

→ Virtual address spaces that include holes are called sparse address spaces.

→ Sparse address spaces are beneficial because the holes can be filled as the stack / heap segments grow or if we wish to dynamically link libraries during program execution

* Demand Paging

→ load only pages only when needed, rather than loading the entire program into the physical memory at program execution time.

→ pages that are never accessed are thus never loaded into the physical memory.

→ demand paging system similar to a paging system with swapping.

→ A lazy swapper is used ⇒ never swaps a page into memory unless that page will be needed.

→ A swapper that deals w/ pages = pager, in the context of demand paging.

Working

→ With swapping, the pager guesses which pages will be used before swapping out again

→ require hardware support to distinguish between pages in memory & pages on the disk - use the valid, invalid bit

→ Pages which are already present in the main memory are memory resident - there is no difference between this & non-demand paging

v => memory resident

* Page Faults

→ When the process to access a page that was not brought into memory.

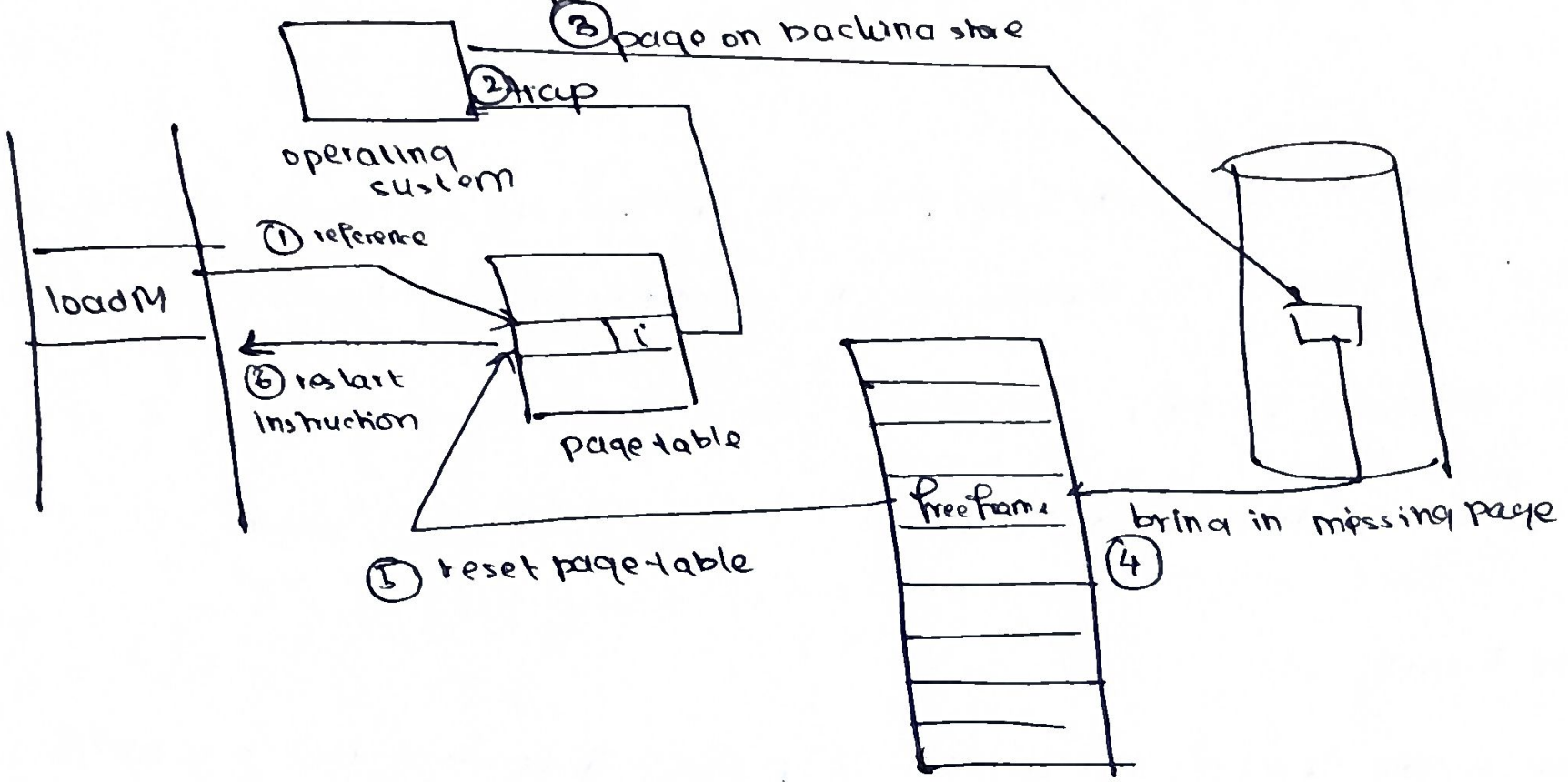
→ causes a trap to the operating system.

Handling Page Faults

1. check an internal table - determine whether the reference was a valid or invalid memory access.
2. If an invalid reference, terminate the process
3. If valid, but page not brought in, page it in.
4. Find a free frame
5. Schedule a disk operation to read the desired page into the newly allocated frame.

6. When the disk read is complete, modify to internal table to indicate that the page is now in memory.

7. Restart the instruction that was interrupted by trap. Process accesses page as though it had always been in memory



* Pure Demand Paging

→ In extreme cases, can start executing a process w/ no pages in the memory

→ When the OS sets the instruction pointer to the first instruction of the process, the process immediately faults for the page.

→ After this page is brought into the memory, the process continues to execute, faulting as necessary.

→ This is called pure-demand paging : never bring a page into memory until it is required.

* Locality of reference

→ some programs could access several new pages of memory w/ each instruction execution

→ would cause multiple page faults per instruction

→ However, this behaviour is highly unlikely, since programs tend to have locality of reference.

* Hardware support for demand paging

(i) page table - w/ valid / invalid bit

(ii) secondary memory - holds pages not in main memory

usually a high-speed disk, called the swap device

section of disk used for this purpose = swap space

* Performance of Demand Paging

Effective access time $= (1-p) \times ma \times$ page fault time

p = probability of a page fault

ma = memory-access time

write about steps taken by computer in case of page fault pg.37

* Demand Paging Optimizations

→ (i) Disk I/O to swap space is generally faster than using the file system.

(ii) Copy entire file image into the swap space at process startup and then perform demand paging from the swap space.

(iii) Some systems attempt to limit the amount of swap space used through demand paging of binary files. Demand pages for such files are brought directly from the file system. When page replacement is called for, these frames can simply be overwritten. Swap spaces must be used for pages not associated with a file - known as anonymous memory

* Swapping on Mobile OS

→ do not support swapping

→ demand page from the file system and reclaim read-only pages, such as code from applications

Page Replacement

→ prevent over-allocation of memory

→ swap out a process, freeing all of its frames & reducing the level of multiprogramming.

* Basic Page Replacement

1. Find the location of the desired page on the disk
2. Find a free frame:
   a. If there is a free frame, use it
   b. If there is no free frame, use a page-replacement algorithm, to select a victim frame.
   c. Write the victim frame to disk, change the page & frame tables accordingly
3. Read the desired page into the newly freed frame, change the page & frame table
4. Continue user process from where page fault occurred.

* Reducing overhead

→ If no frames are free, 2 page transfers are required

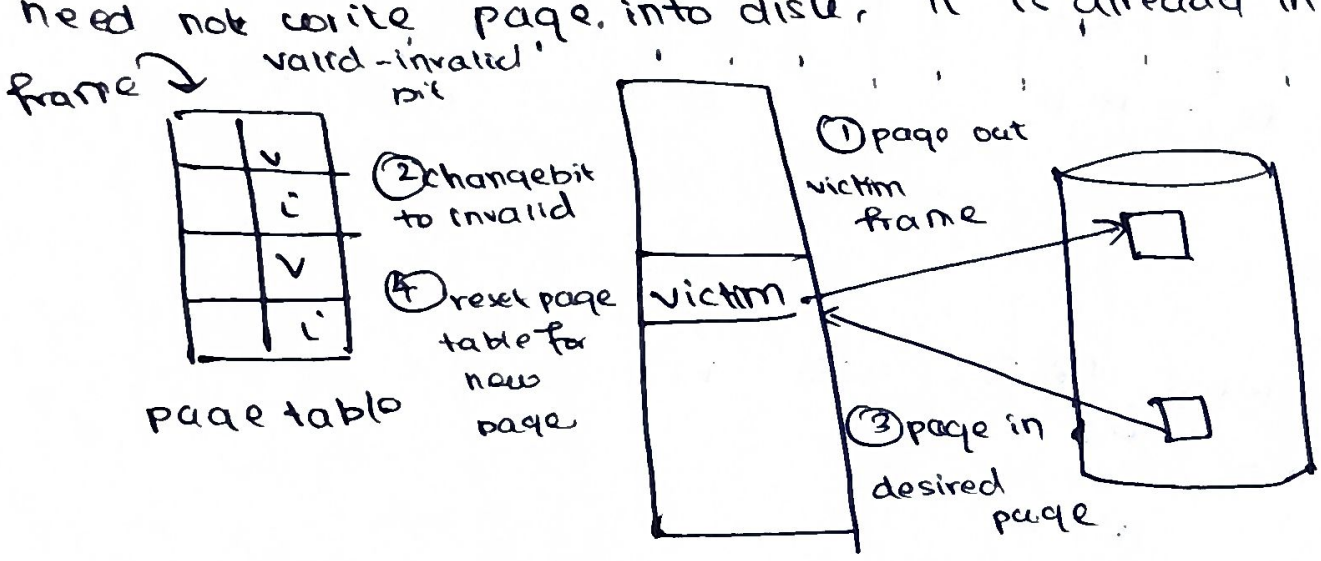
⇒ increases access time

→ solution: use a modify bit.

The modify bit for a page is set by the hardware whenever any byte in the page is written into, indicating that the page has been modified.

→ When a page is selected for replacement, examine the modify bit - if the bit is set, we know that the page has been modified since it was read into the memory.

→ If the memory bit is not set, the page has not been modified; need not write page into disk. It is already there



* Page and Frame Replacement Algorithms

(1) Fram - allocation algorithm

- determines

→ how many frames to give each process

→ which frames to replace

(2) Page - Replacement algorithm

→ want lowest page-fault rate on both first access & re-access

→ evaluate algorithm by running it on a particular string of

memory reference, (reference string), and computing the no. of page faults on that string.

* FIFO Page - Replacement

→ associates each page the time when that page was brought into memory

→ when a page must be replaced, the oldest page is chosen

Example: Reference string: 7 0 1 2 0 3 0 4 2 3 0 3 0 3 2 1 2 0 1 7 0 1

no. of frames = 3

	7	0	1	2	0	3	0	4	2	3	0	3	0	3	2	1	2	0	1	7	0	1
f_1	7	7	7	2		2	2	4	4	4	0					0	0			7	7	7
f_2		0	0	0	✓	3	3	3	2	2	2	✓	✓	✓	✓	1	1	✓	✓	1	0	0
f_3			1	1		1	0	0	0	3	3					3	2			2	2	1
	PF	PF	PF	PF		PF	PF	PF	PF	PF	PF					PF	PF			PF	PF	PF

Total = 15 page faults

Example 2: Reference string: 1 2 3 4 1 2 5 1 2 3 4 5

no. of frames = (a) 3

(b) 4

3 frames

	1	2	3	4	1	2	5	1	2	3	4	5
f_1	1	1	1	4	4	4	5			5	5	
f_2		2	2	2	1	1	1	✓	✓	3	3	✓
f_3			3	3	3	2	2			2	4	
	PF	PF	PF	PF	PF	PF	PF			PF	PF	

Total = 9 pg. faults

4 frames

	1	2	3	4	1	2	5	1	2	3	4	5
f_1	1	1	1	1			5	5	5	5	4	4
f_2		2	2	2	✓	✓	2	1	1	1	1	5
f_3			3	3			3	3	2	2	2	2
f_4				4			4	4	4	3	3	3
	PF	PF	PF	PF			PF	PF	PF	PF	PF	PF

Total = 10 page faults

* Belady's Anomaly

→ The no. of page faults may increase with an increase in the no. of pages.

* Optimal Algorithm

→ replace the page that will not be used for the longest period of time.

→ not practical - since one will not know diatts in future reference string.

→ used only for comparitive purposes.

Example: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

no. of frames = (a) 3
(b) 4

3 frames

	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
f_1	7	7	7	2		2		2			2				2			7		
f_2		0	0	0	✓	0	✓	4	✓	✓	0	✓	✓	✓	✓	0	✓	✓	0	✓
f_3		1	1	1		3		3			3				1			1		

Page faults = 9

4 frames

	7	0	1	2	0	3	0	4	2	3	0	3	0	3	2	1	2	0	1	7	0	1
f_1	7	7	7	7		3		3								1				1		
f_2		0	0	0	✓	0		0	✓	✓	✓	✓	✓	✓	✓	0	✓	✓	✓	0	✓	✓
f_3			1	1		1		4								4				7		
f_4				2		2		2								2				2		

page faults = 8

Example 2: Reference String: 1 2 3 4 1 2 5 1 2 3 4 5

no. of frames: (i) 3
(ii) 4

3 frames

	1	2	3	4	1	2	5	1	2	3	4	5
f_1	1	1	1	1	↷	↷	1	↷	↷	3	3	↷
f_2		2	2	2			2			2	4	
f_3			3	4			5			5	5	

page faults = 7

4 frames

	1	2	3	4	1	2	5	1	2	3	4	5
f_1	1	1	1	1	↷	↷	1	↷	↷	↷	4	↷
f_2		2	2	2			2				2	
f_3			3	3			3				3	
f_4				4			5				5	

page faults = 6

* Least Recently Used Algorithm

→ replace page which has not been used for the longest period of time

→ like optimal algo, but looking backward

Reference string: 7 0 1 2 0 3 0 4 2 3 0 3 0 3 2 1 2 0 1 7 0 1

Req. Faults?

Frames: (i) 3
(ii) 4

	7	0	1	2	0	3	0	4	2	3	0	3	0	3	2	1	2	0	1	7	0	1
f_1	7	7	7	2	↷	2	↷	4	4	4	0	↷	↷	↷	↷	1	↷	1	↷	1	↷	7
f_2		0	0	0		0		0	0	3	3					3		0		0		0
f_3			1	1		3		3	2	2	2					2		2		7		1

w/ 4 frames.

	7	7	7	2	3	0	4	2	3	0	3	0	3	0	3	2	1	2	0	1	7	0	1
f_1	7	7	7	7	3		3										3				3	0	
f_2		0	0	0	0		0										0				7	1	
f_3			1	1	1		4										1				1	1	
f_4				2	2		2										2				2	2	

page faults = 9 page faults

* Implementation of LRU

(1) Counters :

→ Every page entry has a counter; every time a page is referenced through this entry, copy the clock into the counter

→ When a page needs to be changed, look at the counters to find the smallest value.

(2) Stack Implementation

→ Keep a stack of page numbers

→ whenever a page is referenced, it is removed from the stack and put on the top.

→ The most recently used page is always at the top of the stack.

→ should be implemented using a doubly linked list with a head pointer and a tail pointer.

→ each update requires 6 pointer changes, but no need to search for replacement, as the tail pointer points to the bottom of the stack.

* Stack Algorithms

→ the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with n+1 frames

→ For LRU replacement, the set of pages in memory would be the n 'most recently used' pages.

→ If the no. of frames is increased, these n pages will still be the most recently referenced, so they would still be in memory

→ Stack Algorithms can never exhibit Belady's algorithm.

* LRU Approximation Algorithms

→ Very few computer systems provide hardware support for LRU page replacement.

→ LRU approximation algos are used.

* Reference Bit

→ A reference bit for a page is set by the hardware whenever that page is referenced.

→ Initially, all bits are cleared to 0 by the OS.

→ As a user process executes, the bit associated w/ each page referenced is set to 1 by the hardware.

→ can determine which pages have been used & which have not been used, by examining the reference bits

→ However, order of use is not known.

* Additional-Reference-Bits Algorithm

→ order info. by recording the reference bits at regular intervals

→ Those bits are kept in shift registers - contains history of page used.

→ 00000000 ⇒ page has not been used at all for eight time period

→ 1111 1111 ⇒ page has been used at least once in eight time period

A B

→ 1110 0100 vs 0111 0111 ⇒ A has been more recently used than B.

→ interpret 8-bit bytes as unsigned integers, page w/ lowest no. is the LRU page.

* Second-Chance Algorithm (clock algorithm)

→ inspect reference bit of selected bit

if 0 ⇒ replace page

if 1 ⇒ given page a second chance, move on to select the next FIFO page.

→ When the page gets a second chance, its reference bit is cleared, arrival time is set to the current time.

→ ∴ A page that is given a second chance, will not be replaced until all other pages have been replaced.

→ Implement using a circular queue

→ Note: If all bits are set ⇒ degenerates to FIFO replacement

* Enhanced Second - Chance Algorithm

→ modify second-chance algo by considering the reference bit and the modify bit as an ordered pair

4 possible cases:

1. (0,0) ⇒ neither recently used nor modified ⇒ best page to replace

2. (0,1) ⇒ not recently used but modified ⇒ not quite as good, because the page will need to be written out before replacement

3. (1,0) ⇒ recently used by clean ⇒ probably will be used again soon

4. (1,1) ⇒ recently used & modified ⇒ probably will be used again and the page will need to be written out to the disk before it can be replaced.

* Counting - Based Page Replacement

a. Least Frequently Used (LFU)

→ page with the smallest reference count is replaced

→ rationale is that an actively used page must have a large reference count

→ problem: page may be heavily used in the initial phase of a process and then never used again.

Example: Find the no. of page faults for the following reference string, when there are 3 frames

7 0 1 2 0 3 0 4 2 3 0 3 2

	7	0	1	2	0	3	0	4	2	3	0	3	2
F_1	7	7	7	2									
F_2		0	0	0	-								
F_3			1	1									
	F	F	F	F									

0 1 2 3 4 7

↓ ↓ ↓ ↓ ↓ ↓

∅ ∅ ∅ 0 0 ∅

1 1 1 1

Follow FIFO as 2nd rule

* Page Buffering Algorithms

→ keep a pool of free frames

→ when a page fault occurs, a victim frame is chosen.

→ The desired page is written into a free frame from the pool before the victim is written out.

→ This procedure allows the process to restart as soon as possible without waiting for the victim page to be written out.

(or) → maintain a list of modified pages

→ when paging device is idle - write pages there & set to non-dirty

(OR) → keep a pool of free frames, remember which page was in each frame.

→ If referenced again, no need to load contents again from disk

* Applications & Page Replacement

→ In some cases, applications accessing data through the operating system's virtual memory perform worse than if the OS provided no buffering at all.

→ eg. a database, which provides its own memory management & I/O buffering.

→ double buffering - OS is buffering I/O as well as applications

→ OS gives direct access to disk, as a large sequential array of logical blocks ⇒ raw disk

* Allocation of Frames

→ to determine the fixed amount of free memory among the various processes

Minimum no. of frames

→ as the no. of frames allocated to each process decreases, the page fault rate increases → process execution slows

→ determined by architecture

Maximum no. of frames

→ defined by amount of available physical memory

* Allocation Algorithms

① Fixed Allocation

A. equal allocation → divide frames equally among processes

B. proportional allocation ⇒ allocate according to size of process

s_i = size of process p_i

$S = \sum s_i$

m = total no. of frames

a_i = allocation for $p_i = \frac{s_i}{S} \times m$

eg. $m = 62$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 62 \approx 4$

$a_2 = \frac{127}{137} \times 62 \approx 57$

B. Priority Allocation

→ use a proportional allocation scheme using priorities rather than size

→ if process P_i generates a page fault, → select a frame for replacement from a process w/ lower priority number

C. Global & local Allocation

Global replacement - process selects a replacement frame from the set of all frames, one process can take a frame from another

→ execution time may vary greatly

→ commonly used because of greater throughput

local replacement - each process selects from only its own set of allocated frames

→ more consistent per-process performance

→ possible underutilized memory

D. NUMA - Non-Uniform Memory Access

→ A given CPU can access some sections of main memory faster than other sections

→ Those performance differences are caused by how CPUs & memory are interconnected in the system.

→ CPUs on a particular board can access the memory on that board w/ less delay than they can access memory on other boards.

→ called NUMA systems.

Managing NUMA Systems

→ Memory frames should be allocated "as close as possible" to the CPU on which the process is running

→ close ⇒ minimum latency

→ In Solaris, lgroups (latency groups) are created, where each lgroup gathers together close CPUs and memory.

* Thrashing

→ If a process doesn't have enough pages, the page fault rate is very high

→ If there are insufficient frames, there will be a page fault.

→ At this point, it must replace some page

→ Since all the pages are in active use, it must replace a frame that will be needed again right away

→ Consequently, it faults again and again, replacing pages it must bring back in immediately.

→ This high paging activity is called thrashing. A process is thrashing if it is spending more time paging than executing.

* Causes of Thrashing

→ If CPU utilization is too ⇒ increase degree of multiprogramming by introducing a new process

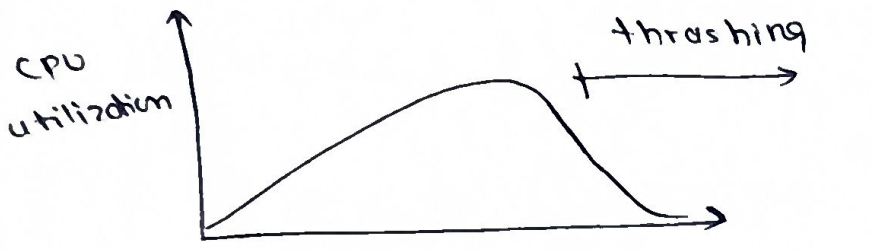
→ If a page fault occurs, the process waits for the paging device & CPU utilization decreases.

→ CPU scheduler sees decreasing CPU utilization & increases the degree of multiprogramming.

→ new process tries to take frames from running processes, causes more page faults.

→ system throughput plunges

→ To stop thrashing - decrease degree of multiprogramming



* limiting effects of thrashing

A. local replacement algorithm | priority replacement algorithm

→ If one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well.

B. Locality Model

→ As a process executes, it moves from locality to locality

→ A locality is a set of pages that are actively used together.

→ A program has different localities, which may overlap

→ Thrashing occurs if Σ size of locality > total memory size

→ limit effects by using local or priority page replacement