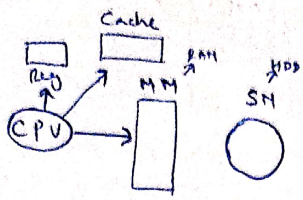
* 04- Memory Management

01- Need for multiprogramming & memory management



MM	Process Size
4MB	4MB

1 process = fraction of 'p' time doing I/O

CPU utilization = $(1-p)$

= 20% ($p = 80\%$)

MM	Process size
16MB	4MB

4 processes – 'p' time doing I/O each process simultaneously

CPU = $1-p^4 \cong 60\%$ ($p = 80\%$)

MM	P Size
32MB	4MB

8 processes – 'p' ($p = 80\%$)

CPU Util = $1-p^8 \cong 83\%$

MM	P
48MB	4MB

$P(80\%)$

12 process – CPU util = $1-p^{12} \cong 73\%$

CPU Util → $1-(p)^n$ (n processes in MM)

↓

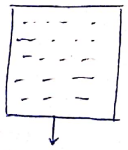
depends on degree of multiprogramming ($\propto$ proportional)

$\lim_{n\to\infty}$ CPU Util = 100%

[Memory management techniques helps to understand the CPU that there are infinite amount of main memory available instead of limited.]

* Memory manager provides security along with allocation & de-allocation techniques.
* Code → compiler → assembly code → assembler → machine language → Program → reside in secondary memory
* The executable code → main memory through linker, loader etc → process

02- Object Code, Relocation and Linker



* Compiler, Assembler, Loader & Linker are combined inside the compiler software itself
* Assembler is hardware dependent, each machine has its own assembler (unique).

Compiler
↓
assembly code
↓
assembler
↓
object code

} compile time

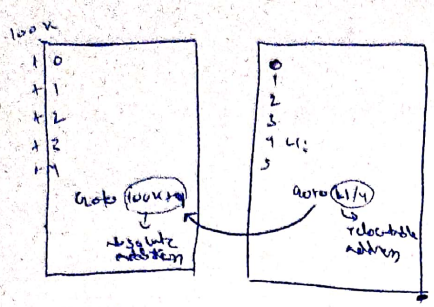
object Code

Header
Text Segment
Data Segment
Relocation info
Symbol table
Debugging info

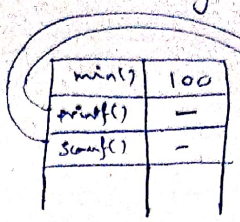
* Header is basically an index, which tells from where or which address other segments starts.
* Text Segment containing only instructions
* Data Segment containing data used in the program
* Relocation info containing where in the parts of the program relocatable address is used. At the time of loading the program into main memory, this info is used.
* Symbol table containing all the symbols used in the program. It contains the list of variables, list of functions defined & list of functions being called & its addresses.

Relocation:- When a program is written, generally its code starts with 0 or 1. So, there could be some jump statements that uses the locally program address as the label. But, in main memory, on the other location, there may be other process or kept for OS itself. So, an address where the program is to transferred, is obviously other than zero. So, the labelled address may must be changed. The starting address must be added.

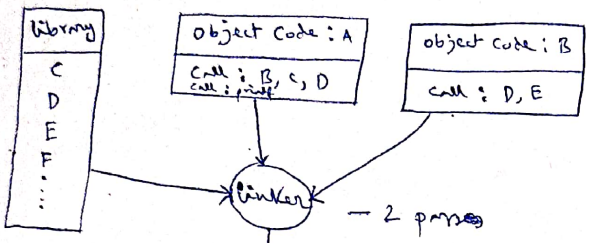
This is called relocation, & the address i.e on the program is called relocatible address. The final address is called absolute address.



* In compile time linking / early binding program is loaded into the main memory.



* Debugging info contains all the information used to trace variables when bugs are removed.

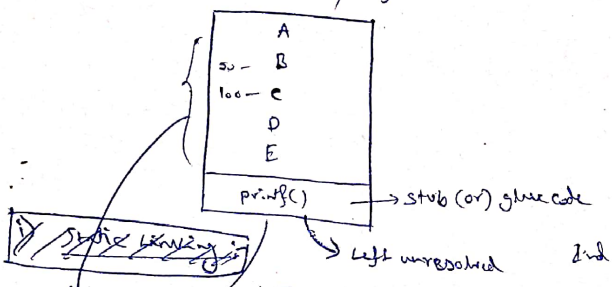


* Linker will find out what are the references that are present in an object code which are unresolved.

1st pass → what are all the segments that are present in the program & also figure out all the external references & where it is going to put each segment.

i) Segment table :- What are all the segments that need to be loaded.

ii) Symbol table :- What are the symbols that have to be resolved.



* The stub will execute on the runtime, the actual address of printf's definition will be placed here.
printf is now loaded into main memory

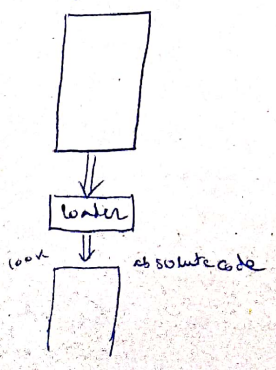
2nd pass → It will replace every unresolved references to their resolved addresses. Linker will also use same relocatable addresses so this thing is done by loader finally.

* Memory usage can be improved by using dynamic linking.
* If version is changed of a library, then dynamic linking saves the time by avoiding static linking & loading.
* Page fault is the only disadvantage of dynamic linking.

Linker → Relocation (relocatable address → relocatable address)
↳ Symbol Resolution (External references resolved)

* Static library is pre-loaded into the memory, so that absolute address is directly available to all linkers. This is static linking

03 – Loader



i) Program Loading (Hard disk → Main Memory)
ii) Relocation
iii) Symbol Resolution

Linker :- Symbol resolution + Relocation
Loader :- Program Loading + Relocation

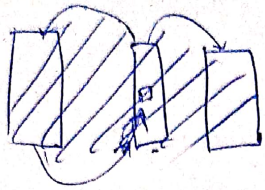
Combinedly called Linker/Loader.

Compile Time :- Symbolic names/constants → Relocatable addresses
Link Time :- Relocatable addresses → Relocatable addresses
Load Time :- Relocatable addresses → Absolute addresses
Run Time :- Dynamic Linking (dynamic linking libraries are added by OS)

04 - GATE 95 question on Linker

A Linker is given object modules for a set of programs that were compiled separately. What information need not be included in an object module?

a) object code, b) Relocation bits, c) names and locations of all external symbols defined in the object module, d) Absolute addresses of internal symbols.



Ans → D (because it is needed at the time of loading)

05 - GATE 98 question on Loader

In a resident OS computer, which of the following system software must reside in the main memory under all situations?

a) assembler, b) linker, c) loader, d) compiler

Ans → c (because of swapping/loading processes)

06 - 2001 Question on Relocation

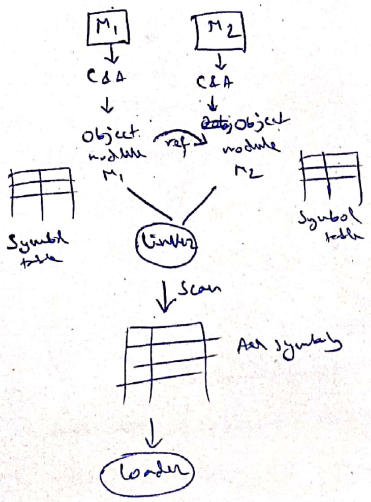
The process of assigning load addresses to the various parts of the program and adjusting the code and data in the program to reflect the assigned addresses is called - a) Assembly, b) Parsing, c) Relocation, d) Symbol Resolution

Ans → C

07 - GATE 2004 Question on Linker

Consider a program 'P' that consists of two source modules M_1 & M_2 contained in two different files. If M_1 contains a reference to a function defined in M_2, the reference will be resolved at - a) Edit time, b) Compile time, c) Link time, d) Load time

Ans → C



08 - GATE 2002 question on Dynamic Linked Libraries

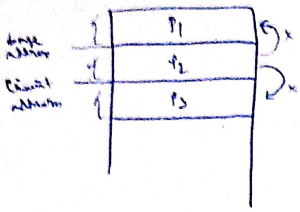
Dynamic linking can cause security concerns because -

a) Security is dynamic,

b) The path for searching dynamic libraries is not known till runtime,

c) Linking is insecure,

d) Cryptographic procedures are not available for dynamic linking.



Ans → B (dynamic libraries)

09 - GATE 2003 question on linking

Which of the following is not an advantage of using shared, dynamically linked libraries as opposed to using statically linked libraries?

a) Smaller size of executable file,

b) Lesser overall page fault rate in the system,

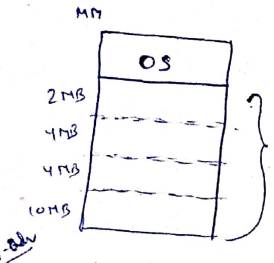
c) Faster program starting,

d) Existing programs need not be re-linked to take advantage of newer versions of libraries.

Ans → B (~~Static~~ no page fault in case of static linking ~~&~~, Page Fault might increase in dynamic linking)

10 - Fixed Partitioning

* Contiguous Memory Allocation (~~Most~~ oldest technique)

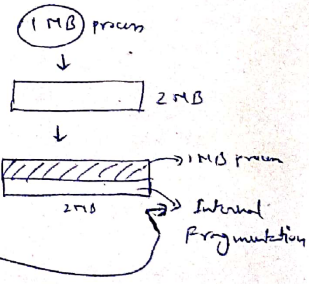


These p memory portions are divided into fixed size partition i.e Fixed Partitioning

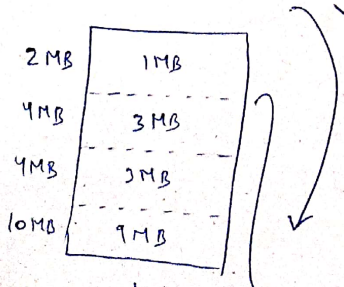
Disadv

- Internal Fragmentation.
- Limitations on process size.
- Degree of multiprogramming is limited or fixed.
- * External Fragmentation.

* Internal Fragmentation



* Whenever within a partition, a process occupy its place partly i.e the whole partition is not used. Then the remaining space cannot be used by other process, this is known as internal fragmentation.

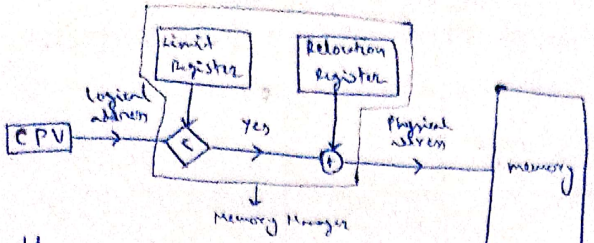


Total free space → 4MB

New process 4MB or less ~~that~~ cannot get space even if spanning them

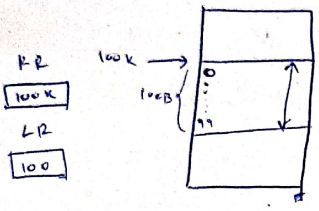
* It happens whenever a process requests a memory space in which the main memory & ~~still~~ cannot get it even if the space is available because the space that is available is not contiguous due to internal fragmentation.

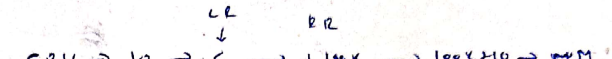
11 - Relocation and Protection in Contiguous Memory Allocation



Problem :→ After relocation when absolute address is generated, due to some of reason the ~~process~~ process may need to be swapped out ~~context switched~~. So after some time, when the process is again loaded into main memory ~~if not~~, loader adds the new memory location with the absolute code ~~it changes~~ not with the relocatable one, because it does not remember the address added during previous relocation.

Solution :→ Runtime binding. The concept of logical & physical address is used here. Logical address is relative to 0 address, ~~not~~ This is also a type of relocation.





So, ~~if~~ relocation is ~~easy~~ easy.

* Access of these registers can't be given to any process because of protection & misuse of outside interference.
To access these register one need special privilege from OS.

* The method is slow.

12 - Dynamic Partitioning

- No internal fragmentation, so no external fragmentation also (except a few cases).
- Partitions are created only when a process arrives, so exact amount of space can be given & no wastage.
- In a certain case, when some processes completes their tasks & free up their spaces, external fragmentation ~~can~~ can happen due to non-contiguousness of ~~the~~ available memory. This is the ~~main~~ reason of external fragmentation
- This problem can by solved by Compaction. Compaction is a process in which ~~all~~ all occupied spaces are pulled together & all ~~the~~ available space are pulled together. So that available space becomes contiguous. It is also called Defragmentation.
- But compaction takes a huge amount of time to copy ~~the~~ all the data of each processes. This time CPU sits idle & efficiency comes down.

Adv

1) Degree of multiprogramming is dynamic.
2) Size of the process is not limited by size of a partition.

Disadv

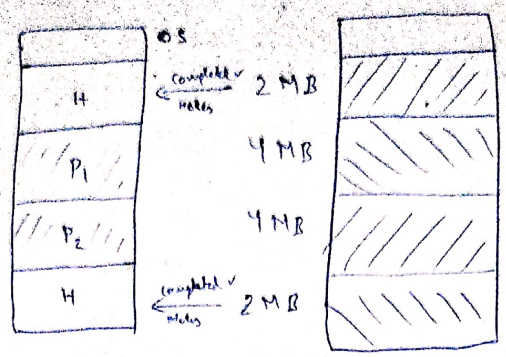
1) Allocation & deallocation of memory is complex.

13 - Bit Map for Dynamic Partitioning

- Data structures used for keep track of occupied & available spaces.

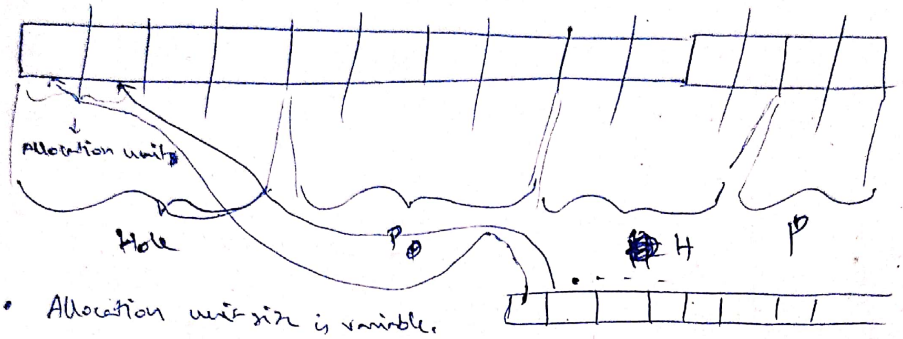
↓

1) Bit Map
2) Linked list



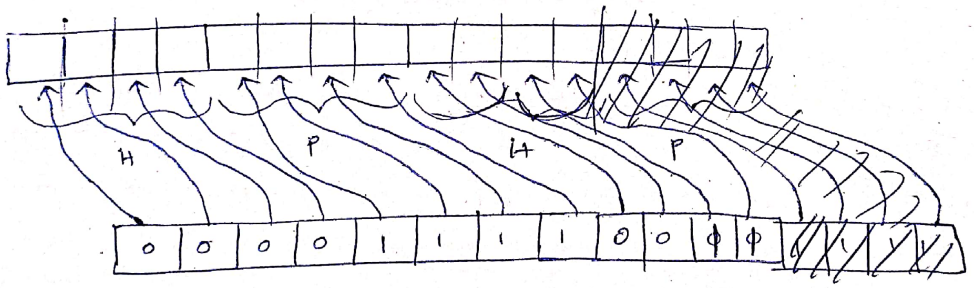
Bit Map

It is used to keep track of holes or free spaces in case dynamic partitioning.



- Allocation unit size is variable.
- The whole partition is divided into several allocation units.
- Then there are some contigious allocation bytes which are empty.
- A bit is assigned for each allocation unit.
  - ↳ 0 → if part of hole
  - ↳ 1 → if part of occupied memory

↓ Total No. of bits = Total no. of allocation units



Allocation unit → 4 Bytes = 32 bits

Bitmap → $\frac{1}{(32+1)} = \frac{1}{33}$ of memory will be occupied by bit map.

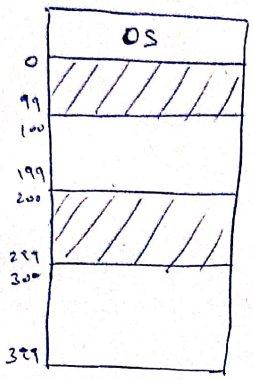
Disadv :-

- Increasing the size of allocation unit can lower the occupancy of bitmap, but there can be problems. Actual process may start or end from a mid point of an allocation unit, so that can't be said free or fully occupied. So, small allocation unit size gives better memory utilization.
- Finding out a hole for a particular process is a disadvantage because for that within the bitmap complex algorithm is used to find a perfect sequence of zero's for that process.

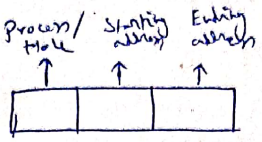
AIN

- When a process completes its task, only by doing all 1's to 0's a hole can be created & there is no shifting required for it, so it will be merged with all other holes automatically.

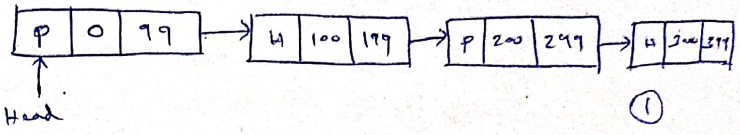
14- Linked List for Dynamic Partitioning



Memory



→ Linked List node structure



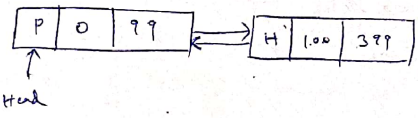
(1)

AIN

- Searching is faster compared to bitmap
- Memory required to maintaining ~~the~~ linked list is less compared to bit map.
- Adding a node when a new process arrives is easy.
- ~~Maintain~~ Maintaining a list in increasing order of starting address is beneficial because holes can be merged in very less time whenever a process completes (or pulled out) its job & making the occupied memory ~~a hole~~ into a hole.

- Maintaining a doubly linked list is more advantagious. So merging will be very easy.

(2)



15- First Fit, Next Fit, Best Fit, Worst Fit

i) **First Fit** :→ Whenever a new process arrives with a memory request, it scans the linked list & assigns a hole that is big enough to hold the process request, is called First Fit. Then the ~~node~~ hole is divided into two ~~nodes~~ parts, one for process allocation & other for available free space i.e hole.

- It is faster.
- Maintainance is easy.
- Most widely used algorithm (even in malloc)

ii) **Next Fit** :→ It works exactly like First fit, the only difference is first fit starts scanning the list from beginning & next fit starts scanning from the point of the list where it is left off.

- First fit is better than Next fit.

iii) Best Fit :→ In the linked list, scan all the ~~holes~~ holes & among them choose the smallest hole that can hold the newly ~~arrived~~ arrived process. This is known as Best Fit.

- It is slow because of searching.
- After allocating the process, the memory which is left is really small that cannot accomodate any process.
- First fit is still better compared to it.

iv) Worst Fit :→ It Scans the whole list & unlike best fit, it finds the biggest hole among all the holes & assign it to process. This is worst fit.

- It is slower because of ~~searching~~ searching.
- First fit is still better than Worst fit.

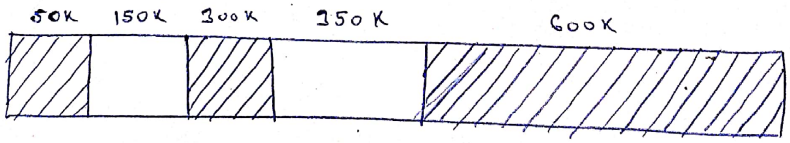
v) Quick Fit :→ It maintains different individual lists for frequently used process sizes. This is Quick Fit.

- Determining frequent process sizes is complex.
- Even if process sizes are determined, in case of ~~not~~ infrequent sizes again holes ~~must be~~ need to be maintained which is very costly & complex process.
- Allocation is quick & de-allocation will take a lot of time.

- Best fit works well if the holes are sorted ~~not this~~ in increasing order wrt size. (Same for worst fit → decreasing order). But still even after this again a problem comes into scenario i.e due to different addresses of each hole, the holes could not be merged. So again, efficiency lacks.
- So, First fit is best fitting algorithm.

16 – GATE 2004 Question on First fit and Best fit

Process requests → 300K, 25K, 125K, 50K



a) Either FF or BF
b) FF but not BF
c) BF but not FF
d) None

Ans → B (FF ✓, BF ✗)

↓

Last process could not be satisfied due to external fragmentation

17 – Variable Partitioning

Q1

A 1000 KB memory is ~~managed~~ managed using variable partitions but no compaction. It currently has two partitions of sizes 200 KB and 260 KB ~~respectively~~ respectively. The smallest allocation request, in Kbytes that could be denied is for –

a) 151, b) 181, c) 231, d) 541

$1000 - (150 + 200 + 150 + 260) = 240$ ✗

$1000 - (180 + 200 + 180 + 260) = 180$ ✓ → Ans → B

~~Total~~ $\frac{\text{Total} - (\text{Process sizes})}{\text{No of Jobs}}$ = Max equal size per hole

18 – GATE 98 Question on Fixed Partitioning

In a computer system where the 'best-fit' algorithm is used for allocating 'jobs' to 'memory partitions', the following situation was encountered –

Partitions ~~Sizes~~ in KB	4K	8K	20K	2K			
Jobs	2K	14K	3K	6K	10K	20K	2K
Time for execution	4	10	2	1	1	8	6

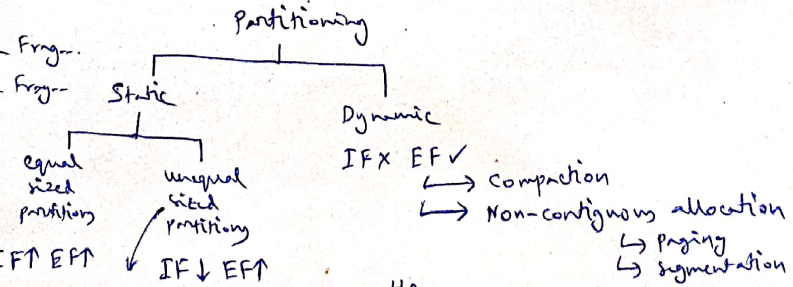
When will the 20K job complete?

4K	3K (0–2)		
8K	6K (0–1)	2K (1–7)	
20K	14K (0–10)	10K (10–11)	20K (11–19)
2K	2K (0–4)		

∴ 20K job → 19 time

19 – Summary On Partitioning

IF: Internal Frag..
EF: External Frag..



- Overlays :→ A program larger than ~~the~~ the highest partition can be run, because the whole program doesn't run together. So, some amount of program is moved to the partition & after ~~job~~ completion of ~~job~~, it is pulled out & new portion is moved. This concept is called overlays. Temporary solution for static & unequal sized partitions.
- Virtual memory concepts come from shortage of space in main memory due to large sized programs. Overlays is almost same, just it is done by user, if VM done by OS.

20- Overlays

Consider a 2 pass assembler

Pass 1: 70 KB
Pass 2: 80 KB
Symbol table: 30 KB Common routine: 20 KB
Total memory: 200 KB

- Overlay driver written by user to perform overlays.

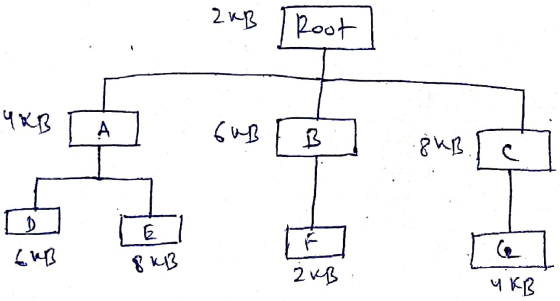
But at any time only one pass will be in use and both the passes always need ST and CR. If overlay driver is 10KB, then what is the minimum partition size required?

Pass 1: (70KB + 30 KB + 20KB) + 10 KB = 130 KB
Pass 2: (80KB + 30 KB + 20 KB) + 10KB = 140 KB

Minimum partition size = 140KB.

21- GATE 98 Question on Overlays

The overlay tree for a program is shown as below:

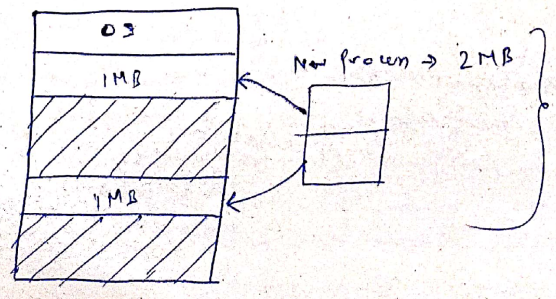


What will be the size of the partition (in physical memory) required to load (and run) this program?

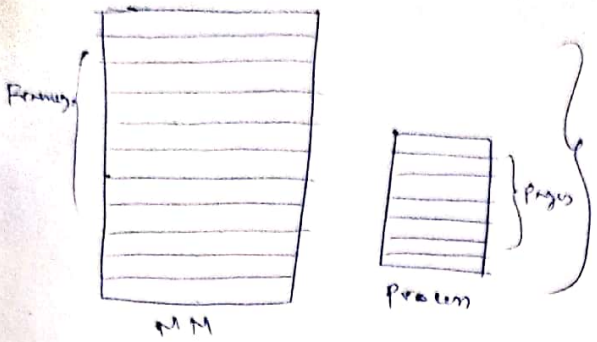
a) 12 KB, b) 14 KB, c) 10 KB, d) 8 KB

(R + A + D) = 12 KB
(R + A + E) = 14 KB
(R + B + F) = 10 KB
(R + C + G) = 14 KB Ans → B

22- Need For Paging



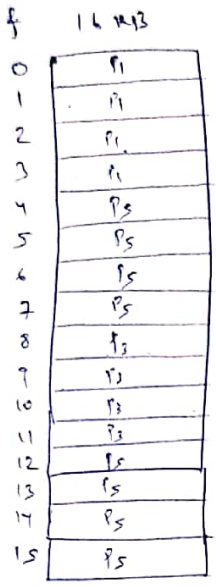
This is problematic i.e. # dividing a process with respect to hole size is very much complex & time consuming.



Divide the process without any reference to hole size & also divide the main memory in such a way that each part of main memory can hold one part of process.

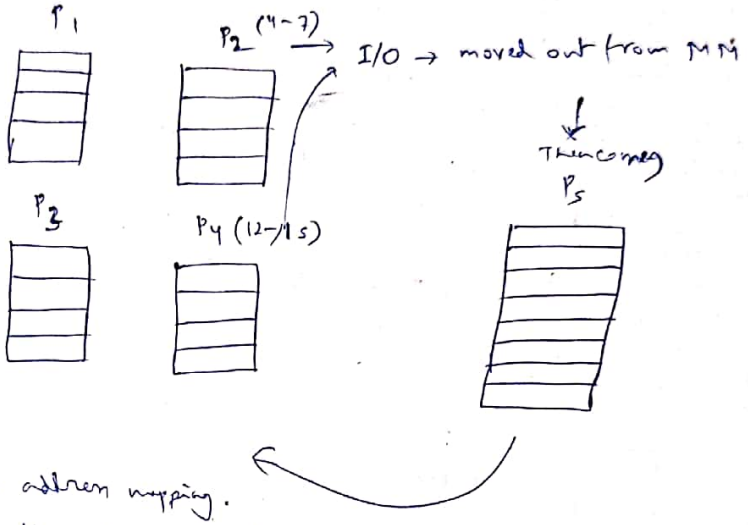
Parts of process → Page
Parts of MM → Frame

1 Page size = 1 frame size

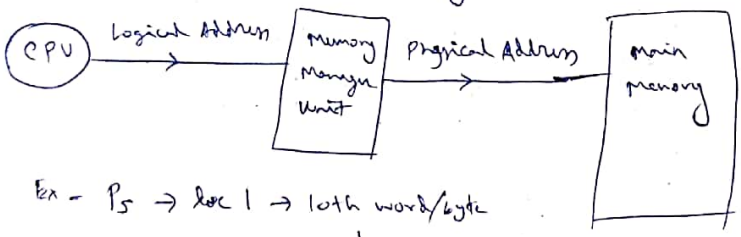


Frame size → 1 KB → Job, total 16 frames

4 processes → Page size → 1 KB



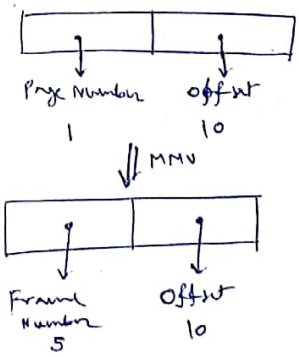
- The problem here is address mapping.
  Not done in load time done to relocatable code to absolute code conversion.
  This is done by run time binding.



Ex - P5 → loc 1 → 10th word/byte

MM ← loc 5

- Logical Address is split into two parts →

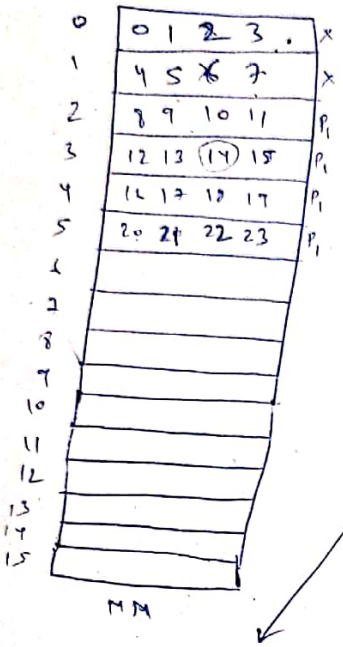


23 - Paging Explained With Example

Memory Size - 64 B

$2^6 = 64$ ⇒ so 6 bits are needed for address

X: occupied



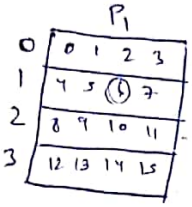
Frame size = 4 B

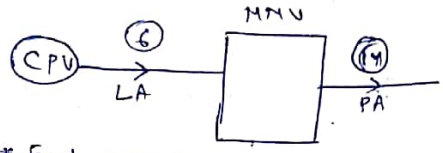
No. of frames = $\frac{64}{4}$ = 16 frames

Process size = 16 B

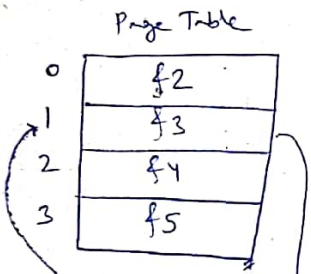
Page size = 4 B

No of pages in the process = $\frac{16}{4}$ = 4 pages





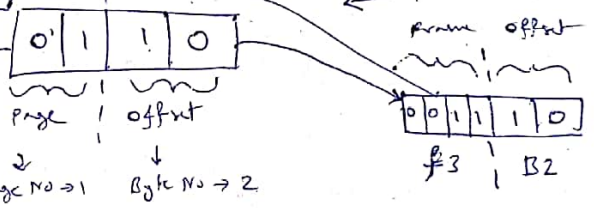
* Each process have a page table
* It contains information where the page is residing in MM. Basically PT contains Frame Numbers



- Logical Address size depends on process size
- Physical " " " " main memory "

Logical Address → 4 B for ~~this~~ P_1 → 0 1 1 0

~~Physical Address → 4 B for MM~~



- A process & its page table resides in MM, so MM is accessed two times.
- This is not done by loader as it converts the relocatable code to logical address, so we need run time binding.

24 - Basics of Binary Addresses

2^{10} - K

2^{20} - M

2^{30} - G

2^{40} - T

128 GB

$= 2^7 \times 2^{30}$ B $= 2^{37}$ B

$2^{28} = 2^8 \times 2^{20}$

$= 256$ MB

25 - Physical Address Space & Logical Address Space

- Smallest addressable unit in a computer system is called a Word.

- Physical Address Space = Size of the Main Memory

PAS = 128 KB → $2^7 \times 2^{10}$ B = 2^{17} B

Word size → 4 B = 2^2 B

PAS in words → $\frac{2^{17}}{2^2} = 2^{15}$ words

PA = 15 bits address

∴ PAS = M words

PA = $\log_2 M$ bits

- Logical Address Space = Size of ~~the~~ a process

LAS = 256 MB → $2^8 \times 2^{20}$ B = 2^{28} B

Word size → 4 B = 2^2 B

~~PAS~~ LAS in words → $\frac{2^{28}}{2^2} = 2^{26}$ B

LA → 26 bits

∴ LAS = L words

LA = $\log_2 L$ bits

- Virtual memory concept is useful if process size is very very large than main memory.
- Whenever the size of the process is larger than the main memory, we are going to put only a part of the process in the main memory & be able to run it, is called Virtual Memory.
- Now, VM concept is used even if the process size is lesser than MM to accomodate as much ~~as possible~~ process as possible. So, degree of multiprogramming is increased.
- LAS is ~~also~~ also called as Virtual Address Space (VAS) & LA is also called as Virtual Address (VA).
- Page Size = Frame Size = P Words.
- $(\log_2 P)$ bits are required for an address of a word within a page. These bits are called Page Offset.
- Page size = 4 KB
  Word size = 4 B
  - Total words / Page size in words = $\frac{4\text{KB}}{4\text{B}} = 1\text{KW} = 2^{10}\text{W}$
  - Each word need 10 bits for addressing.
  - Page Offset = 10 bits

26 - Page Table

PAS = Main Memory Size = M words → PA = $\lceil \log_2 M \rceil$ (ceil is used as convention)

LAS = Process Size = L words → LA = $\lceil \log_2 L \rceil$ = m bits

Page Size = P words = p bits

↳ Page Offset = $\lceil \log_2 P \rceil$ = p bits

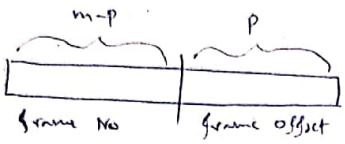
∴ LAS = 128 MB = 2^{27} B → LA = 27 bits
PAS = 1 MB = 2^{20} B → PA = 20 bits
PS = 4 KB = 2^{12} B → PO = 12 bits
} → need virtual memory

frame = 2^p W, PAS = 2^m W,

∴ No. of frames = $\frac{2^m}{2^p} = 2^{m-p}$

∴ PAS = $\boxed{2^{m-p} \times 2^p}$

PA →

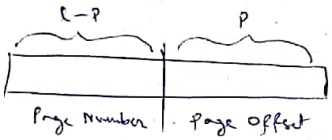


PS : 2^p words, LAS : 2^l Words

∴ No. of Pages = $\frac{2^l}{2^p} = 2^{l-p}$

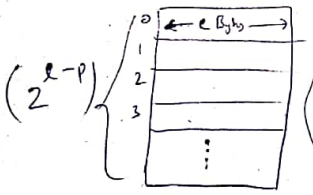
∴ LAS = $\boxed{2^{l-p} \times 2^p}$

LA →



Page Table :- For every page, present in the process there will be an entry in the page table.

No. of entries in the PT = no. of pages in the process

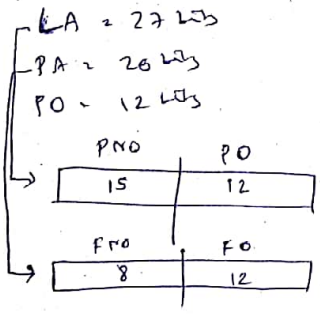


PT size = $(2^{l-p}) * e$ Bytes

- A page table entry contains the frame number in which this particular page (index) is going to be present.

PT size (if e is not given) = $(2^{l-p}) * (m-p)$ Bytes

(This is not the natural case, because a page table entry contains more information about that page along with frame number.
E.g - permissions (W/R/E), referenced or not, modified or not, history etc)



No of pages = 2^{15} = Page Table entries

Page table size = frame No. × 2^{15}

= 8 × 2^{15} bits

= 2^{15} Bytes = 32 KB

Page size = 4 KB

∴ So, multilevel paging is needed because the process is very big & PT is so big that it cannot fit in one frame.

∴ So PT needs to be divided into multiple pages.

27 - Numerical on Paging

[1W = 1B → Assumption]

Page Table Entry ↘ ↙ Page Table size

LAS	PAS	LA	PA	Page size	Page offset	Pages	Frames	PTE	PTS
128 KB	128 KB			4 KB					
256 KB	1 MB			4 KB				2B	
1 MB					10		256		
	512 MB			2^{12}		256			
				2^{12}		2^{10}	2^9		
				2^{14}		2^8			256 B

1) LAS = PAS = 128 KB = $2^7 \times 2^{10}$ B = 2^{17} B

LA = $\log_2 2^{17}$ = 17 bits

PA = $\log_2 2^{17}$ = 17 bits

Page size = 4 KB = $2^2 \times 2^{10}$ B = 2^{12} B

Page offset = 12 bits

LAS → | PNO (17-12=5) | PO (12) |

∴ No of pages = 2^5 = 32

↳ ∵ LAS = No of pages * Page size

PAS → | FNO (17-12=5) | FO (12) |

∴ No of frames = 2^5 = 32

PTE size = size of a frame = 5 bits

PTS = $2^5 * 5$ ~~Bytes~~ bits = $\frac{2^5}{2^3} * 5 = 2^2 * 5$ = 20 Bytes

2) LAS = 256 KB = $2^8 \times 2^{10}$ B = 2^{18} B → LA = $\log_2 2^{18}$ = 18 bits

PAS = 1 MB = 2^{20} B → PA = $\log_2 2^{20}$ = 20 bits

Page size = 4 KB = $2^2 \times 2^{10}$ B = 2^{12} B → PO = $\log_2 2^{12}$ = 12 bits

LAS → | PNO (18-12=6) | PO (12) | → No of pages = 2^6 = 64

PAS → | FNO (20-12=8) | FO (12) | → No of frames = 2^8 = 256

PTE size → 2B = 16 bits, PTS = 2^6 x 16 bits = 2^{10} bits = 1024 bits

= $\frac{2^6 \times 16}{8} = 2^7$ = 128 Bytes

3) LAS → 1 MB = 2^{20} B → LA = 20 bits

PO → 10 bits ∴ LAS → | PNO (20-10=10) | PO (10) |

Page size → 2^{10} = 1024 bits = 128 B

No of pages → 2^{10} = 1024

Frames = 256 → FNO = $\log_2 256$ = 8 bits → PAS → | FNO (8) | FO (10) |

↳ PA = 18 bits, PAS = $2^{18} = 2^{10} \times 2^8$ = 256 KB

PTE = FNo = 8 bits, PTS = $2^{10} \times 8$ bits = 2^{10} Bytes = 1024 B = 1 KB

4) PAS = 512 MB = $2^9 \times 2^{20} = 2^{29}$ Bytes

• PA = 29 bits

Page size = 2^{12} → PO = 12 bits

PAS → | PNO: 29-12 = 17 | PO: 12 | Frames = 2^{17}

Pages = 256 → 2^8 → PNO = 8

LAS → | PNO: 8 | PO: 12 | → 2^{20} = 1 MB

LA = 20 bits

PTE = Frame size = 17 bits • PTS = 256 × 17 bits

= $\frac{2^8}{2^3} \times 17$ Bytes

= 32 × 17 Bytes

5) Page size = 2^{12}

PO → 12 bits

Pages = 2^{10} → PNO → 10 bits → LAS = $2^{10+12} = 2^{22}$ B = 4 MB

Frames = 2^9 → PNO = 9 bits → PAS = $2^{9+12} = 2^{21}$ B = 2 MB

LA = 22 bits, PA = 21 bits, PTE = 9 bits, PTS = $2^{10} \times 9$ bits

6) Pages = 2^8 → PNO → 8 bits

PTS = 256 B PTS = NO of Pages × Frame size

⇒ 256 = 2^8 × Frame size

Hoof ∩ Frames = 2^8

⇒ Frame size = $\frac{256}{2^8}$ ⇒ Frame = 1 B

↳ Frame size = PNO = 1 B = 8 bits PTE = 8

↳ PAS → | PNO: 8 | FO: 14 | Page size = 2^{14}

LAS → | PNO: 8 | PO: 14 | PO = 14

PAS ⇒ 2^{22}, LAS ⇒ 2^{22} LA = PA = 22 bits

28 - Page Table Entry