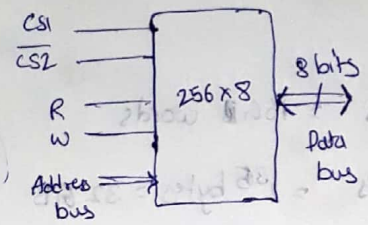
15/08/20

Memory Management:

(Primary memory or Physical memory or RAM)

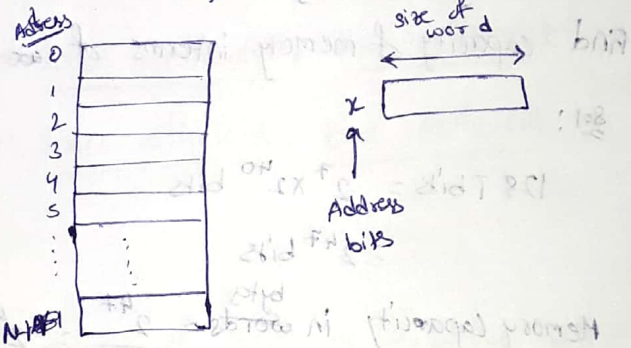
→ There are two views of memory

(i) Physical view:



(ii) Abstract view (programmer's view):

viewed as 1d array of words



Memory[0 ... N] (N words)

Eg: If size of word is 'm' bits and no of bits in address is 'n'

then

Memory Capacity (total words) $= 2^n = N$

→ If N = 8 words

$\Rightarrow n = 3$ bits

i.e.,

$N = 2^n$ words

$n = \log_2 N$ bits

$\Rightarrow$ N = 16 GB

$\Rightarrow$ i.e., $N = 16 \times 2^{30} = 2^{34}$

$n = 34$

→ If n=28 bit

a max of 2^{28} Bytes can be addressed

i.e, 256 MB

→ If n=34 bits and word length = 16 bits

find capacity of memory interms of words, bytes, bits.

Sol:

$2^n = 2^{34}$ words = 16G words

$2^{34} \times 2$ bytes $= 2^{35}$ bytes = 32 GB

$2^{34} \times 16$ bits $= 2^{38}$ bits = 256 G.bits

→ let capacity of memory be 128 Tbits and word size be 64 bits

find capacity of memory interms of bytes and words

Sol:

128 Tbits = $2^7 \times 2^{40}$ bits

$= 2^{47}$ bits

Memory capacity in bytes = $\frac{2^{47}}{8} = 2^{44}$ bytes

= 16TB

Memory capacity in words = $\frac{2^{47}}{64} = 2^{41}$ words

= 2 T. words.

Loading vs Linking

↳ Loading of executable program: from disk to memory.

Loading:

Loading is of two types:

i) Static loading:

* Entire program is loaded before execution

(ii) Dynamic Loading:

Modules can be loaded during runtime.

Eg: Consider

```
main()
{  if(...)
       f1();
   else
       f2();
}
```

In static loading we load main(), f1(), f2() into memory. However we use only one out of f1 & f2. This is space inefficient.

Thus we can use dynamic loading and load only main() initially and load either f1 or f2 based on the requirement. at the time of execution.

→ However execution time is more if we use dynamic loading.

→ Static loading: Space inefficient, time efficient

Dynamic loading: Space efficient, time inefficient

Linking:

Linking is the process of resolving external references

↳ function, variable (global) or any objects.

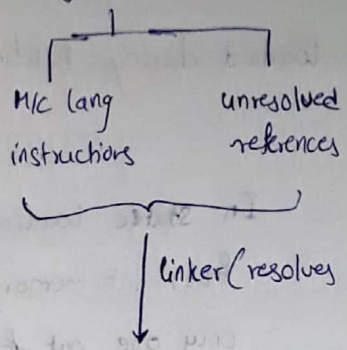
kk.c

```
extern int x; // external reference
main()
{
   :
   f() // since f() is defined after
   :     main() the address of f()
}        is not known earlier. This
f()      is an unresolved reference.
{
   :
   g() // BSA ——→ unresolved reference
   :
}
g()
{
   :
   h()//external reference
   :
}
```

ak.c

```
int x;
h()
{
   :
   scanf()
   :
}
```

Compiler generates object code



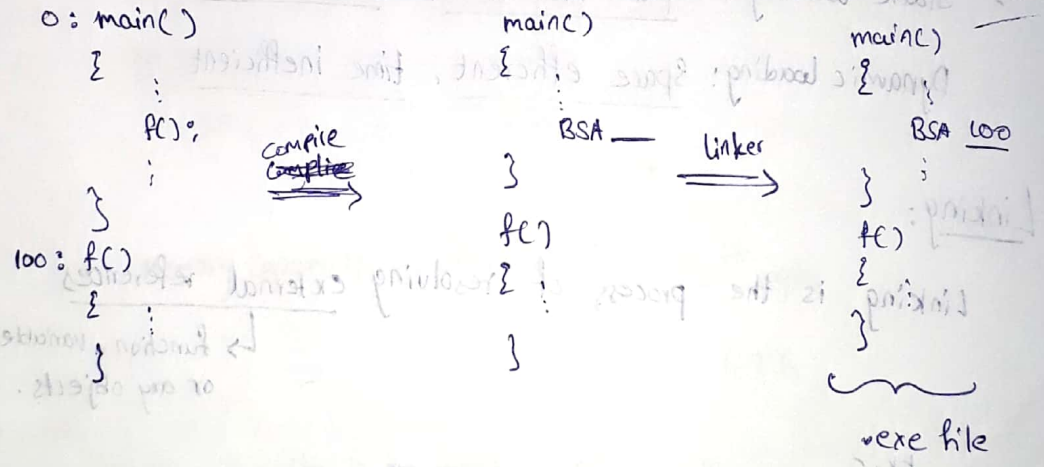
Linking is of two types

(i) Static Linking
(ii) Dynamic Linking

The linker resolves addresses through logical addresses

i) Static Linking:

* Linking is done before runtime.



Drawback:

* Sometimes unnecessary modules may be linked and this is inefficient. for which we sometime may postpone linking till runtime (i.e., late binding or dynamic linking)
* However execution of program is faster in the case of static linking.

ii) Dynamic Linking / Late Binding:

For every unresolved reference compiles associates a piece of code called Stub. This stub when executes at runtime calls linker.

→ The libraries which are linked at runtime are called Dynamic Link Libraries (DLL)

→ It is more space efficient.

Benefits of dynamic linking

* space efficiency
* Reusability of libraries
* flexibility of modification (debugging becomes easy)

Drawback

* time inefficient
* Security threat

→ Static linking is more secure.

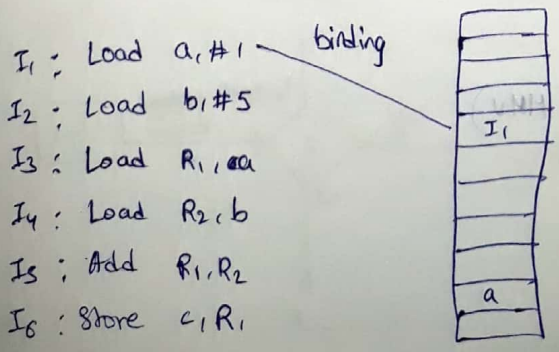
Modules needed by linker:

→ object code
→ external references
→ Relocation information

* Linker does not need physical address of memory.

Address Binding

Association of program instructions and data units to memory locations is called address binding.



The time at which binding takes place is called binding time.

Binding can take place at three times:

- (i) Compile time } Static
- (ii) Load time } Static
- (iii) Run time } Dynamic

Compile time Binding:

* Compiler associates physical addresses.
* The program is loaded into fixed memory locations.

Load time Binding:

* Here loader does the work of binding.
* Based on the value in base register, the instruction are loaded into memory.

Runtime Binding (Dynamic Relocation)

* Here loader does the work of binding at runtime.

→ If a process is swapped out

* it must be swapped into the same memory location if we use compiletime binding or load time binding.
* it can be swapped into different memory location if we use runtime binding.

Functions & Goals of Memory Manager:

Functions:

i) Allocation

ii) Protection

iii) Address Translation (done by MMU).

iv) Free Space Management

v) Deallocation

Goals:

→ Effective utilization of memory
i.e., Minimize Fragmentation

→ Manage the execution of larger program in a small memory area.

→ Memory management Techniques

Contiguous

i.e., program is centralized
i.e., program is present at one place

Eg: Partitions < fixed / dynamic
Overlays
Swapping

Non-contiguous

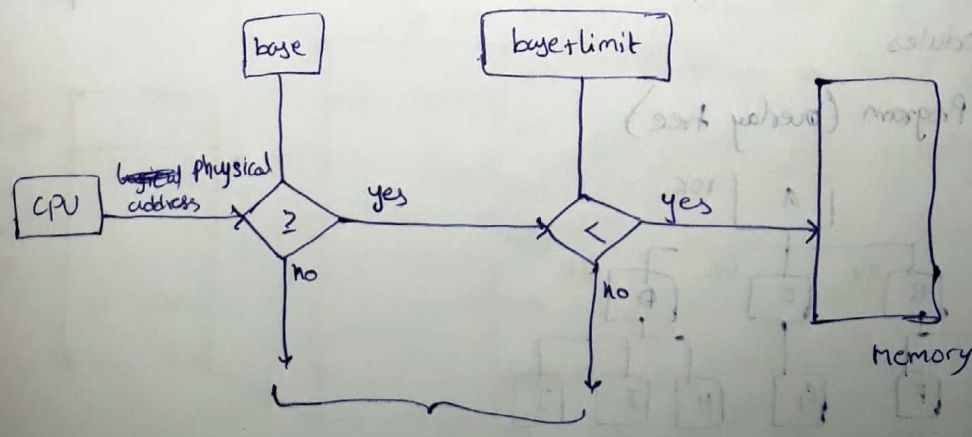
→ Here program may be divided into modules and different modules may be at different places

Eg: Paging
~~Segment~~
Segmentation
Segmented paging
Virtual Memory

Protection:

→ (Relocation register)

* base register contains starting address of the process.
* limit register contains size of process.



1. Contiguous Memory Management Techniques:

a) Overlays:

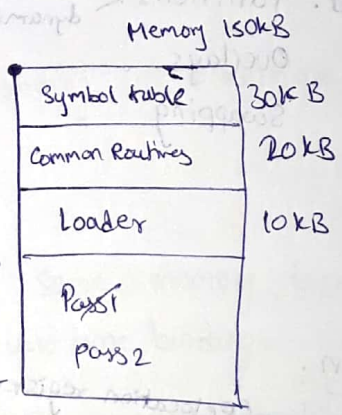
Assume we have a 2-pass Assembler.

Below are sizes of each module.

pass1 : 70 kB
pass 2 : 80 kB
Symbol table : 30 kB
Common Routines : 20 kB
Overlay loader (or) overlay driver : 10 kB

} total memory : 210 kB

Assume we have only 150kB of free memory



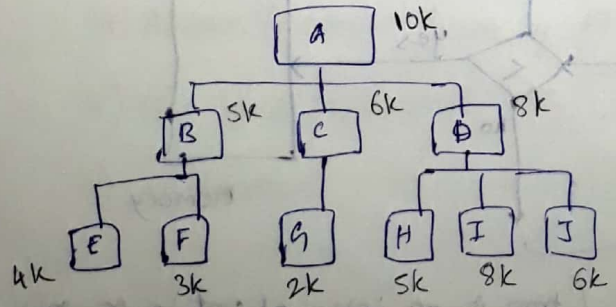
→ Pass1 and pass2 are independent modules. i.e., pass1 can run without any use of pass2 and vice versa

→ Now 1st overlay loader loads pass1. Later pass1 is overlayed (replaced) by pass2

Limitation:

→ Program must be possible to be divided into independent modules

Eg: Program (overlay tree)



For the above overlay tree, what is the minimum amount of memory required for executing? ~~the above ?~~

sol:

During pass1 we may need A, B, E (i.e., 19KB)

During pass2 we may need A, B, F (i.e., 18k)

In that way at some pass we may need A, D, I

i.e., 10+8+8 = 26k

∴ size of max pass = 26k

∴ Min memory required = 26k

* → Minimum memory required = Max{path lengths from root to the leaves}

b) Partitions:

Partitions is of 2 types:

i) Fixed Partitioning (MFT: Multiprogramming with Fixed no of Tasks)

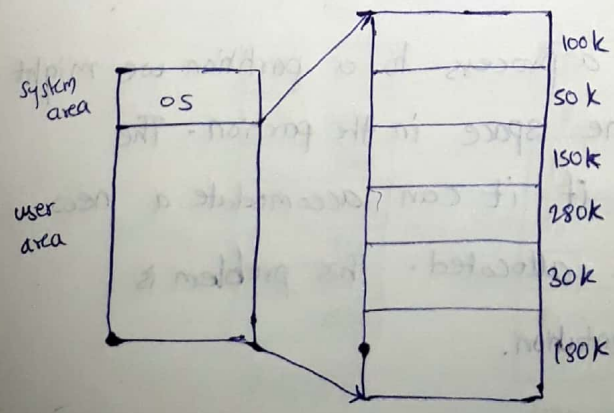
ii) Variable Partitioning (MVT: Multiprogramming with Variable no of Tasks)

i) Fixed Partitions: (or) Static Partitioning

User area: It is divided into fixed no of partitions.

Partitions may be of different sizes.

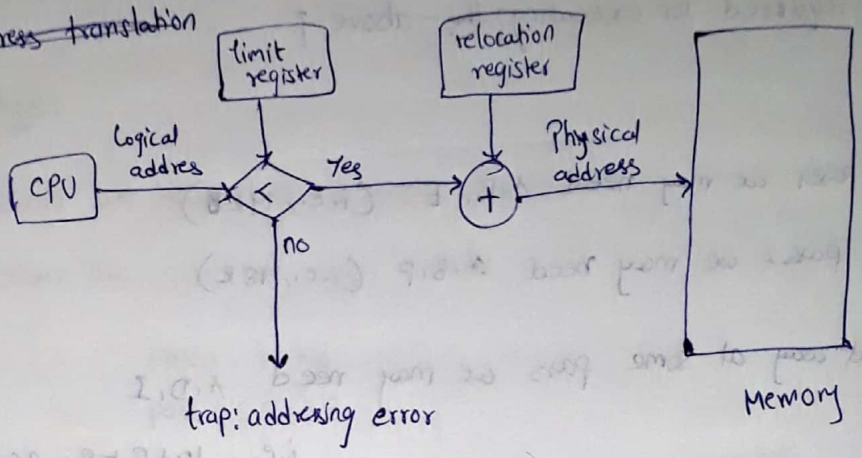
Each partition holds exactly one process.



Address translation & Protection:

Protection

address translation



Allocation Strategies:

First fit:

→ search from first partition and store it wherever enough sized partition is found

Best fit:

Smallest free and big enough partition is choosen

Worst fit:

Largest free and big enough partition is choosen

Next fit:

* Next fit works like fit except that search for free partition starts from last allocation.

Performance issues:

i) Internal fragmentation:

After allocating a process to a partition we might be left out with some space in the partion. The left out space even if it can accomodate a new process shouldn't be allocated. This problem is called internal fragmentation.

2) External Fragmentation:

No external fragmentation

3) Degree of Multiprogramming is restricted to no of partitions

4) ~~size of max~~ Maximum process size that is runnable is limited

5) Best fit is the most suitable allocation technique. cuz it has least amount of internal fragmentation.

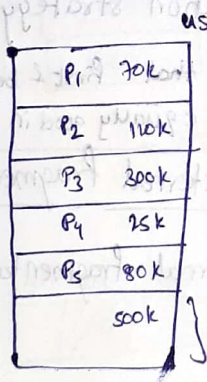
(ii) Variable Partitions (or) Dynamic Partitioning

Partitions are created dynamically (created) at runtime

At time t assume we have below requests

70k; 120k; 300k; 25k; 80k;
P_1 P_2 P_3 P_4 P_5

Here OS maintains a table to keep track of which parts of memory are available

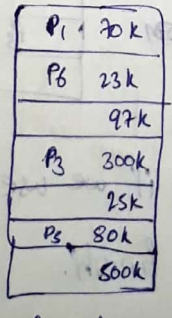


→ If P_5 finishes, its execution is

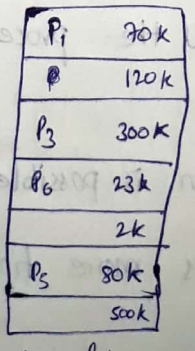
we have two holes of 80k and 500k adjacent to each other. Now these two will be joined. This joining of adjacent holes is called coalescing.

Assume after some time P_2 and P_4 has finished execution and we have a new process request of P_6 of 23k.

Now we may use first fit, best fit, worst fit, next fit



first fit



best fit

Performance issues

1. No internal fragmentation
2. We may have External fragmentation

Eg: for previous example,

assume we have a process req of 550k, we can't place in the memory. However total available space is more than 550k. Thus we say that we may have external fragmentation.

> we say we have EF iff
> - We can't allocate memory for process even if total available space is more than the size of process

3. Deg. of M.Pr is flexible
4. Maximum size of runnable process is also flexible.
5. Using best fit creates small free holes. These holes are wasted in the form of external fragmentation.

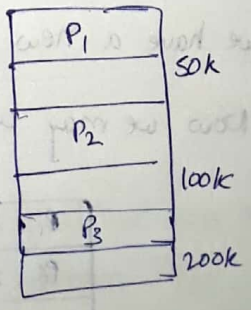
- Hence worst fit is best allocation strategy for dynamic partitioning. (In TB it is given that first & best are better than worst fit: first & best are equally good in storage utilization, however first fit is faster)

Overcoming the problem of External fragmentation:

we have 2 way for overcoming external fragmentation.

i) Compaction (or) Defragmentation:

At time t, assume memory is as shown below



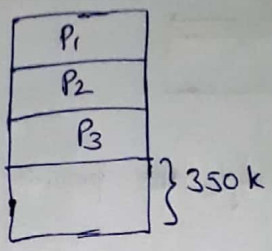
Now if we have a process (P_4) request with size 250k we cannot allocate it.

→ Compaction moves all the processes to one place.

→ However compaction is possible only if we use runtime binding.

→ Also compaction takes more time to finish.

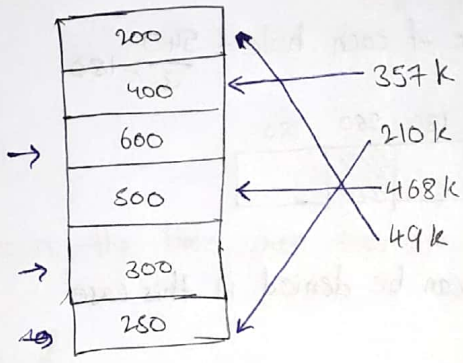
Memory after compaction is



(iii) Non-Contiguous Allocation (Paging)

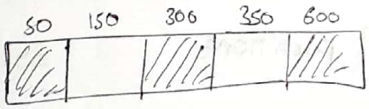
If we don't use contiguous allocation we can now ~~place~~ divide the program and place it in different holes.

H4/1

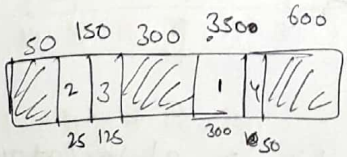


~~250 600~~ ∴ 600k & 300k

H4/2

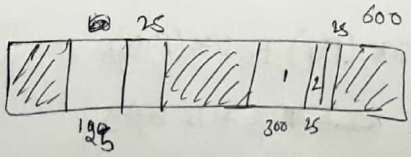


First fit



✓

Best fit



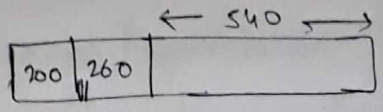
✗

only first fit

∴ opt (b)

H4/3

(i)



To get denied always we provide maximum hole possible

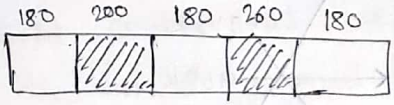
∴ 541k

(ii) To find smallest req to be denied we must create maximum possible holes of equal sizes

max possible holes = 3

total free space = 540

size of each hole = $\frac{540}{3} = 180$



∴ 181k can be denied in this case

H4M

Memory size = 2^{46} bytes

no of partitions = $\frac{2^{46}}{2^{24}} = 2^{22}$ partitions

$= 4M$ partitions

partition address (Ptr) = 22 bits

(i) size of pointer to the nearest byte

nearest byte to 22 bits is 3 bytes (24 bits)

(ii) size of each entry = size (Ptr) + size (PID)

= 3 + 4 bytes

= 7 bytes

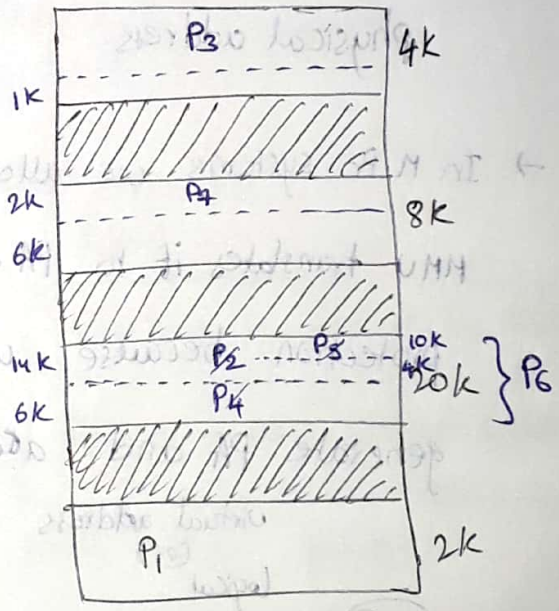
i.e., $7 \times 500 = 3500$ bytes

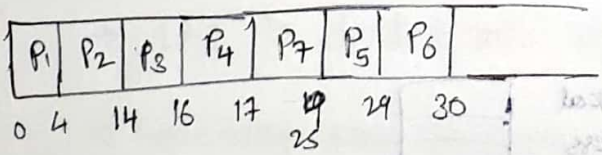
H5 15 Load time is the time when they are loaded into memory

The load time of

P_1, P_2, P_3, P_4, P_7 is 0

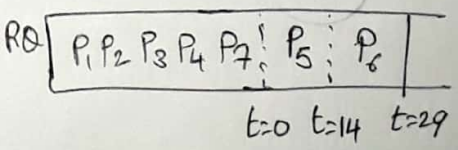
P_5 & P_6 must wait until 20k hole is freed





Load time of $P_5 = 14$

Load time of $P_6 = 29$



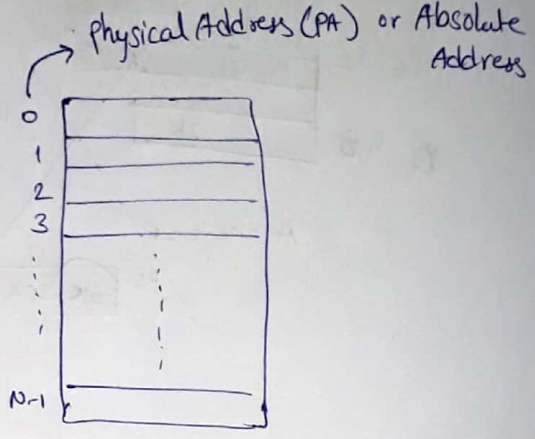
Completion times can be seen in the gantt chart.

2. Non-Contiguous Memory Allocation Techniques

Logical Address Space (LAS) vs Physical Address Space (PAS)

(or)

(Virtual Address Space (VAS))

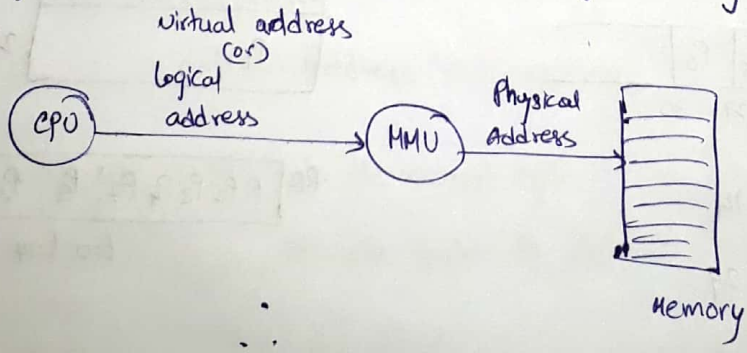


* → In all uniprogrammed OS, the address generated by the program is physical address

* → Also in compile time binding & load time binding, the address generated ~~is phy~~ by program is physical Address

* → In runtime binding, the address generated by the program is logical address which is not equal to the corresponding physical address.

* → In M.Pr systems we allow CPU to generate logical address and MMU translates it to PA. ~~This way~~ This approach provides protection because we are not letting executing program generate PA and access memory directly.



→ Based on the technique we use the hardware of MMU changes. ~~The hardware~~ It may be page table (or) segment table

→ If size of PA is 'n' bits, then PAS is 2^n words.
also size of MAR is n bits.

→ If size of VA is 'm' bits, then LAS (or) VAS is 2^m words
also size of address bus of CPU is m bits.

→ The address binding used in here is runtime binding.

→ LAS = 2^{LA} words

LA = $\log_2$ LAS bits

→ PAS = 2^{PA} words

PA = $\log_2$ PAS bits

→ theoritically

~~LAS ≥~~ LAS > PAS
LAS < PAS
LAS = PAS

→ But pratically

LAS > PAS

i) Paging: (simple paging)

Paging helps in sharing code

Assume

LAS = 8kB; PAS = 4kB Page size = 1kB (PS)

=> LA = 13 bits PA = 12 bits

Pagesize is defined by H/W

Paging increases context switch time

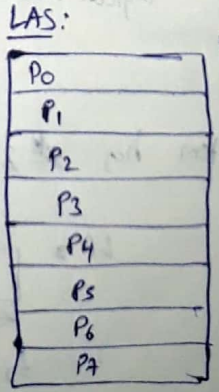
Step 1: Organization of LAS and PAS

→ LAS is divided into equal size units called pages.

→ Generally ~~page no of pages~~ page size is power of 2 (but not necessary though)

→ no of pages = $\frac{8kB}{1kB}$ = 8 pages

i.e., $\text{no of pages}(N) = \frac{LAS}{PS}$



→ Every page is given page number or page id.

→ Page No (P) = $\log_2 8 = \log_2 N$ i.e., $\log_2 N = \log_2 8 = 3$ bit

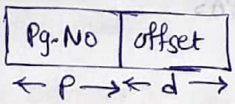
$N = 2^P$

→ Page offset (d) = $\log_2$ PS (bits) i.e., pg. off = $\log 1KB = \log 2^{10} = 10$

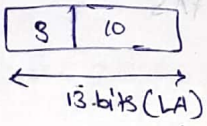
$PS = 2^d$

→ Here we generally select req page first, then using page offset then we go to required word.

→ Logical Address format:



i.e.,



Eg: LA = 24 bits; & PS = 4KB

then

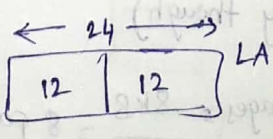
No of pages, $N = \frac{2^{24}}{2^{12}} = 2^{12}$ pages

$LAS = 2^{24} = 16$ MB

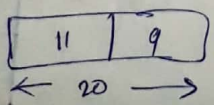
$P = \log_2 2^{12} = 12$ bits

$d = \log_2 PS = 12$ bits

Logical address format



Eg: A system has 2k pages with page offset of 9 bits then



$P = \log 2^{11} = 11$

$\Rightarrow LA = 20$

$LAS = 2^{20} = 1$ MB

$P = 11$

$d = 9$

Eg: System supports a LA of 32 bits if page size is 16 kB

Calculate no of pages in System

Sol:

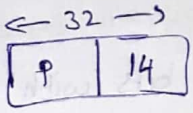
$LAS = 2^{32}$

$PS = 2^{14}$

no of pages $= \frac{2^{32}}{2^{14}} = 2^{18}$ pages

$\therefore N = 2^{18}$

Method 2:



$\Rightarrow p = 32 - 14 = 18$

$\Rightarrow$ no of pages, $N = 2^{18}$

STEP 2: Organization of PAS:

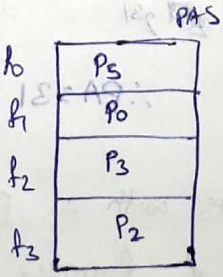
→ PAS is divided into equal size units known as frames (page frames)

→ Frame-size (FS) = Page size

→ Any page can be stored in any frame (Non contiguous allocation)

In the example,

PAS = 4 kB PS = 1 kB



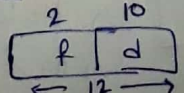
no of frames (M) = PAS / PS

OS maintains frame table to keep track of information like which frames are allocated & which are free

→ frame no (f) = $\log_2 M$ bits

→ frame offset = page offset = d

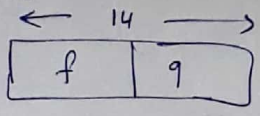
→ physical address format:



Eg: PAS = 16KB ; d = 9 bits

then

PAS Phy address format



⇒ f = 5

⇒ no of frames (M) = 2^5

⇒ page size = 2^9 = 512 Bytes

Eg: System has a LA of 35 bits with 4k pages. If it has 256 frames then calculate size of PA

Sol:

LAS = 2^{35} = 32GB

no of pages (N) = 4K = 2^{12}

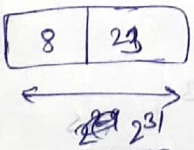
⇒ page size = $\frac{LAS}{N} = \frac{2^{35}}{2^{12}} = 2^{23}$ bytes

⇒ page offset (d) = 23

no of frames (M) = 256

∴ frame number (f) = $\log_2 256 = 8$

⇒



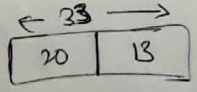
2^{31}

∴ PA = 31

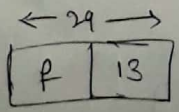
(Q28) System has LA of 33 bits with a page size of 8 kB. If PA is 29 bits. Calculate no of frames.

Sol:

LA



PA



⇒ f = 16

⇒ no of ~~pages~~ frames = 2^{16} = 64k frames.

Step3: Organization of MMU

MMU in paging is page table or page map table or address translation table.

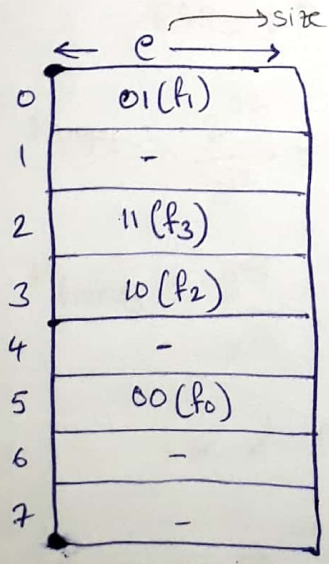
→ Each processes has its own page table

→ Page tables are generally stored in memory.

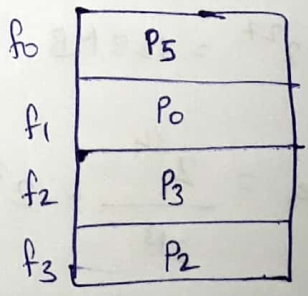
→ page tables is organized as a set of entries and each entry is known as page table entry.

→ No of entries in PT = no of pages in LAS

→ PT entries contains the frame no (f) in which the page is present.



Page table (MMU)

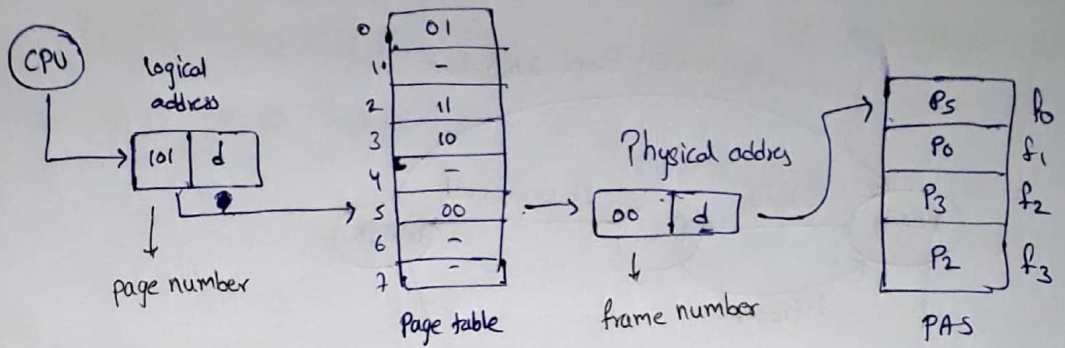


PAS

→ PT entries are generally in bytes.

→ Page table size = No of pages in LAS x size of each entry = N x e

Address translation (logical to physical)



∴ 101 | d corresponds to d^{th} word in page frame 0 of memory.

Q29 LA = 34 bits; PA = 27 bits;

PS = 8 kB;

LAS = ? N_{pages} = ? P.A.S = ?

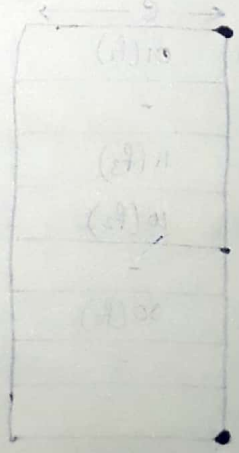
M_{frames} = ? p = ? f = ?

d = ?

Find format of logical address and physical address

Find size of page table (approx).

Sol:



LAS = 2^{34} = 16 GB

PA = 2^{27} = 128 MB

$N_{pages} = \frac{2^{34}}{2^{13}} = 2^{21}$ pages

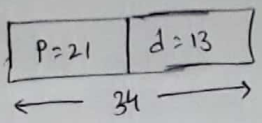
$M_{frames} = \frac{2^{27}}{2^{13}} = 2^{14}$ frames

$d = \log_2 PS = \log_2 2^{13} = 13$

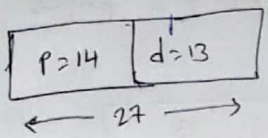
$p = \log_2 N = 21$

$f = \log_2 M = 14$

logical address format



physical address format



size of page table = no of pages (entries) × size of each entry

$= 2^{21} \times 16$ bits

$= 2^{22}$ bytes (size entry = 16)

$\cong$ 4MB

(Q30) Consider a ~~regis~~ system supporting a 32 bit VAS with a page size of 4KB. If PAS is 64MB calculate the approximate PT size in bytes:

Sol:

$PS = 2^{12}$

~~X~~ $VAS = 2^{32}$

$PAS = 2^{26}$

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

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

size of PT entry = 14 bits $\cong$ 2 bytes

size of PT = $2^{20} \times 2 =$ 2MB

H4/6

$PS = 2^{13}$ bytes

$PTS = 24 \times 2^{20}$ bytes

~~$= 3 \times 2^{23}$ bytes~~

$e = 24$ bits $= 3$ bytes

no of pages $= \frac{24 \times 2^{20}}{3} = 8 \times 2^{20} = 2^{23}$ pages

logical address:

23	13

$LA = 23 + 13 = 36$ bits

$LAS = 2^{36} = 64$ GB

H4/7

$VA = l$

$N = Z \Rightarrow p = \log_2 Z$

$M = H \Rightarrow f = \log_2 H$

LA:

p	d

(l)

$l = p + d \Rightarrow d = l - p$

$$d = l - \log_2 Z$$

PA:

f	d

$PAS = 2^{f+d} = 2^{f + l - \log_2 Z} = \frac{2^{f+l}}{Z} = \frac{2^{\log_2 H + l}}{Z} = \frac{H \cdot 2^l}{Z}$

$\therefore d = l - \log_2 Z$, $PAS = \frac{2^{f+l}}{Z} = \frac{H \cdot 2^l}{Z}$

H4/8

$LA = 32$ bits $\Rightarrow LAS = 2^{32}$

$PTES = 4$ bytes

$PTS = 1$ frame $=$ page size i.e., $PTS = PS$

$N \times e = PS$

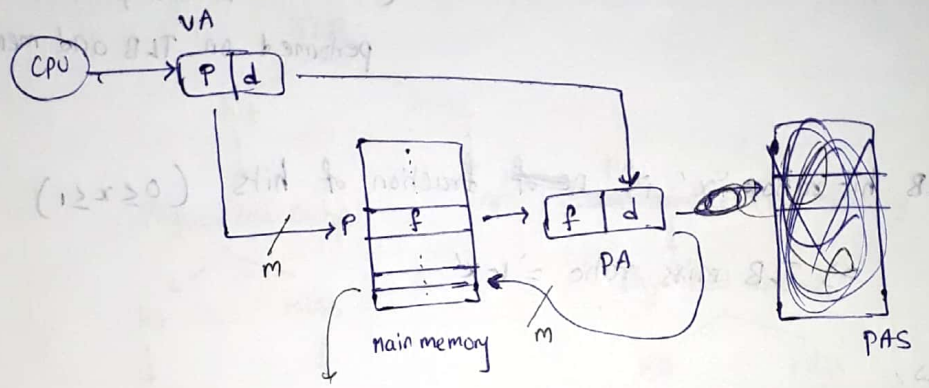
$$\frac{2^{32}}{PS} \times 4 = PS$$

$$\Rightarrow PS^2 = 2^{34}$$

$$\Rightarrow PS = 2^{17} \text{ bytes} = 128\text{kB}$$

Performance of Paging:

1) Temporal issue (Impact on time):



↦ Let main memory access time be 'm'.

So address translation time = m (overhead due to page table)

data access time = m

Due this throughput decreases

∴ **Effective memory access time = 2m**

Now we reduce this overhead 'm' using the concept of TLB.

TLB: Translation Look aside Buffer. (Logical address cache)

TLB is similar to cache.

→ TLB contains previously resolved VA and its corresponding PA

→ Let TLB access time be 'c' (c <<< m)

→ Now if corresponding VA is found in TLB, which takes less access time, we directly use the corresponding PA to access required data in the memory.

→ The event of finding required address in TLB is called TLB hit.

Here access time = c+m

→ The event of missing required address in TLB is called TLB miss

Here access time = c+2m (Also '2m' if a parallel search is performed on TLB and memory)

→ TLB hit ratio 'x' is fraction of hits $(0 \le x \le 1)$

=) TLB miss ratio = '1-x'

Now,

Effective Memory access time with TLB	$= x(c+m) + (1-x)(c+2m)$

→ Hit ratio = $\frac{\text{no of hits}}{\text{no of references}}$

Eg: m = 100ns

c = 20ns

x = 90%

Sol:

without TLB;

Effec. Mem. AT = 2(100) = 200

With TLB

Effe Mem. AT = 0.9(100+20) + 0.1(20+200)

= 108 + 22 = 130ns (less than without TLB)

If 'x' were 10% then

Effe. Mem. AT = 0.1(120) + 0.9(220) = 210ns (more than without TLB)

→ The TLB we used is Logical address cache.

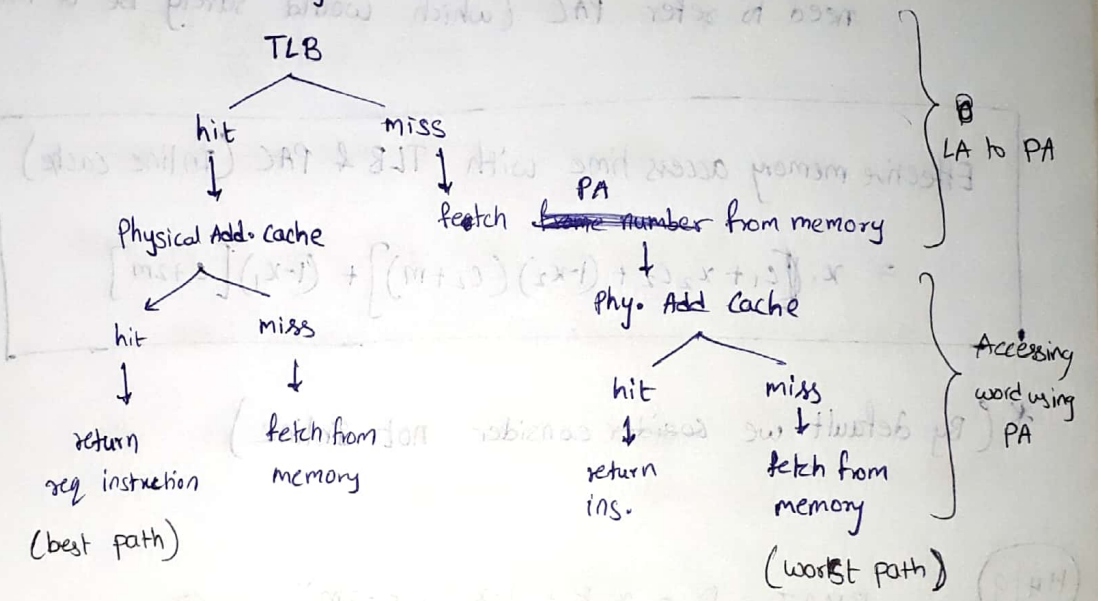
~~If we use physical address~~

After obtaining required PA we need to access memory again.

To reduce this we can use physical address cache. (PAC)

Physical address cache contains ~~both~~ PA and corresponding data.

→ with this now accessing instruction is



→ Let hit ratio of TLB = x_1

hit ratio of PAC = x_2

access time of TLB = c_1

acess time of PAC = c_2

Memory access time = m

Effective memory access time with both TLB & PAC

$$= x_1 \left[c_1 + x_2 c_2 + (1-x_2)(c_2+m) \right] + (1-x_1) \left[\underbrace{c_1+m}_{\text{for PA}} + x_2 c_2 + (1-x_2)(c_2+m) \right]$$

(This is valid if cache is not In-line)

* → not In-line cache means both TLB & PAC are independent

* → In-line cache means TLB & PAC are dependent

~~For example~~

i.e., whenever we remove an entry from TLB (VA & PA) the corresponding entry in PAC (PA & instruction or data) is also removed.

In this case if TLB miss occurs then there is no need to refer PAC (which would surely be a miss)

Effective memory access time with TLB & PAC (Inline cache)

$$= x_1[c_1 + x_2 c_2 + (1-x_2)(c_2+m)] + (1-x_1)[c+2m]$$

* (By default we ~~consider~~ consider not Inline)

H4/9

$$EMAT_1 = D = x_1 k + (1-x_1)(k+2m) \quad -①$$

$$EMAT_2 = Z = x_2 k + (1-x_2)(k+2m) \quad -②$$

$$\Rightarrow ① - ② \Rightarrow D - Z = (x_1 - x_2)k + (1 - x_1 - 1 + x_2)(k+2m)$$

$$D - Z = (x_1 - x_2)k - (x_1 - x_2)(k+2m)$$

$$D - Z = (x_1 - x_2)[k - k - 2m]$$

H4/9

without TLB

$$EMAT = D = 2m \Rightarrow m = \frac{D}{2}$$

with TLB

$$EMAT = Z$$

$$x\left(k + \frac{D}{2}\right) + (1-x)(k+D) = Z$$

$$xk + \frac{Dx}{2} + k + D - xk - xD = Z$$

$$\frac{Dx}{2} = k+D-z$$

$$x = \frac{2(k+D-z)}{D}$$

(14/4/10)

LA = 32 $\Rightarrow$ LAS = 2^{32}

PS = 2^{13}

word size = 4 bytes

e = page table entry size = 4 bytes (1 word)

$$N = \frac{LAS}{PS} = 2^{19} \text{ pages}$$

Size of page table = N × e = 2^{19} words

time req to load page table = $2^{19} \times 100$ nsec

total process runtime = 100 msec = 100×10^6 nsec

$$\text{req. fraction} = \frac{2^{19} \times 100}{100 \times 10^6} = \frac{2^{19}}{10^6} \cong 0.5242$$

i.e., 52.42%

2) Spatial Issue (space optimization)

Assume

LA = 32 bits ; PS = 4kB

$\Rightarrow$ $N_{pages} = 2^{20}$

Page table size, $\boxed{PTS \propto N_{pages}}$

PTS = 1M entries

If e(PTES) is 4 bytes

then PTS = 4MB

i.e., Every process will have a page table of 4MB

* It is not desirable to have larger page tables. So we need to reduce page table size of a process.

Reducing PT size of a process

i) Increase page size:

$PTS = N * e$

$$PTS \propto N \propto \frac{1}{PS} \Rightarrow PTS \propto \frac{1}{PS}$$

However increasing PS has its own draw back.

Eg: Assume we have a program of size 1026 bytes

if PS = 1024 bytes

here we require 2 page in which we use only 2 bytes of 2nd page

i.e., Internal fragmentation = 1022 bytes

if PS = 2 bytes

then we require 513 pages.

i.e., Internal fragmentation

* ⇒ Increasing page size increases internal fragmentation

So, now, we need to come up with an optimal page size

Optimal page size:

Let VAS = S bytes

PS = p bytes

PTE = 'e' bytes

$\Rightarrow PTS = \frac{VAS}{PS} \times PTE$

$\Rightarrow$ $$PTS = \frac{S}{p} \times e$$

Internal fragmentation can happen only in the last page

In the worst case IF could be almost equal to page size

$\therefore$ Avg. IF = P/2

→ page table size & Avg IF are clearly overheads

$\therefore$ optimal page size is such that total overhead is minimum

i.e., $\frac{se}{p} + \frac{p}{2}$ is minimum

Let $f = \frac{se}{p} + \frac{p}{2}$

$$\frac{df}{dp} = \frac{-se}{p^2} + \frac{1}{2} = 0$$

$$\Rightarrow \frac{se}{p^2} = \frac{1}{2} \Rightarrow p = \sqrt{2se}$$

$\therefore$ optimal page size, $P = \sqrt{2se}$

w.r.to overhead of PTs & IF

W24/11

$VAS = 2^{16}$

$PS = 2^{12}$

$N = 16$ pages

$N_{text} = 8$ pages

$N_{data} = 5$ pages

$N_{stack} = 4$ pages

} 17 pages

a) $\therefore$ does not fit

a) Assume PS = 4 bytes

$N_{pages} = \frac{2^{16}}{2^2} = 2^{14}$ pages $= 16384$

text : $\frac{2^{15}}{2^2} = 2^{13}$ pages $= 8912$

data = $\frac{16386}{4} = 4096.5 \Rightarrow 4097$ pages

} 16257

Stack = $\frac{15870}{4} = 3967.5 \Rightarrow 3968$ pages

Thus for PS = 4 bytes, the program fits

b) Assume ~~max~~ max possible page size = 2^x bytes

$\Rightarrow N_{pages} = \frac{2^{16}}{2^x} = 2^{16-x}$ pages

text : $\frac{2^{15}}{2^x} = 2^{15-x}$ pages

data : $\left\lceil \frac{16386}{2^x} \right\rceil$ pages

stack : $\left\lceil \frac{15870}{2^x} \right\rceil$ pages

$\Rightarrow 2^{15-x} + \left\lceil \frac{16386}{2^x} \right\rceil + \left\lceil \frac{15870}{2^x} \right\rceil \leq 2^{16-x}$

$2^{15-x} + \left(2^{14-x} + 1\right) + \left\lceil \frac{15870}{2^x} \right\rceil \leq 2^{16-x} \quad (\forall x \geq 2)$

$\Rightarrow \left\lceil \frac{15870}{2^x} \right\rceil \leq \frac{2^{16} - 2^{15} - 2^{14}}{2^x} - 1$

$\Rightarrow \left\lceil \frac{15870}{2^x} \right\rceil \leq \frac{2^{14}(4-2-1)}{2^x} - 1$

$$\left\lceil \frac{15870}{2^x} \right\rceil \leq \frac{2^{14} - 2^x}{2^x}$$

$$\left\lceil \frac{15870}{2^x} \right\rceil \leq 2^{14-x} - 1$$

put $x = 5$

$$496 \leq 512 - 1 \checkmark$$

put $x = 7$

$$124 \leq 128 - 1 \checkmark$$

put $x = 8$

$$62 \leq 64 - 1 \checkmark$$

put $x = 9$

$$31 \leq 32 - 1 \checkmark$$

put $x = 10$

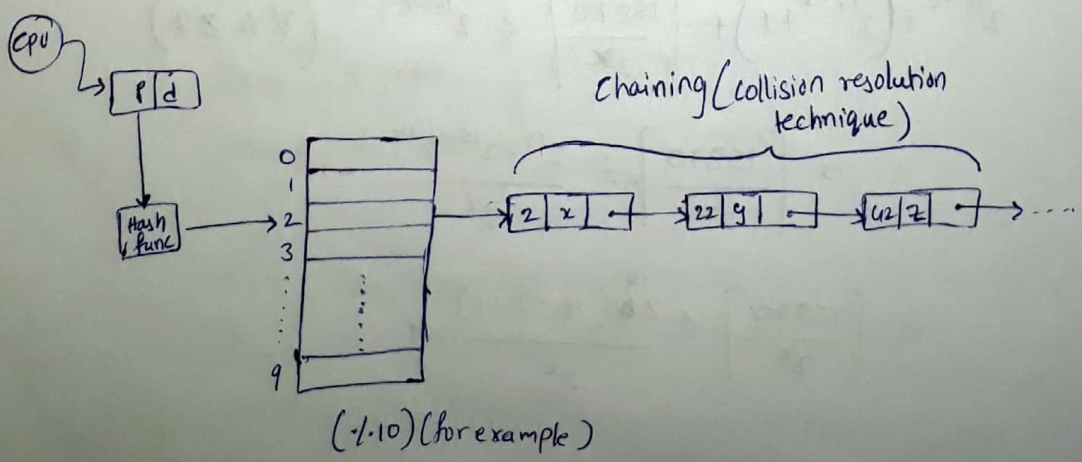
$$16 \leq 16 - 1 \times$$

∴ Max page size such that the program could fit is

$$2^x = 2^9 = 512 \text{ bytes.}$$

(ii) **Hashed Paging / paging with Hashing**

* This techniques helps reduce size of page table.



(-1.10) (for example)

→ LAS says max possible process size. However we may have smaller processes. In this case size of hash page table is very less than normal page table.

For example,

LAS = 2^{32} = 4GB (max possible process size)
PS = 4kB

In normal page table size of page table for any process is same.

Eg Assume we have process of 128kB

$$\Rightarrow \text{no of pages} = \frac{128}{4} = 32$$

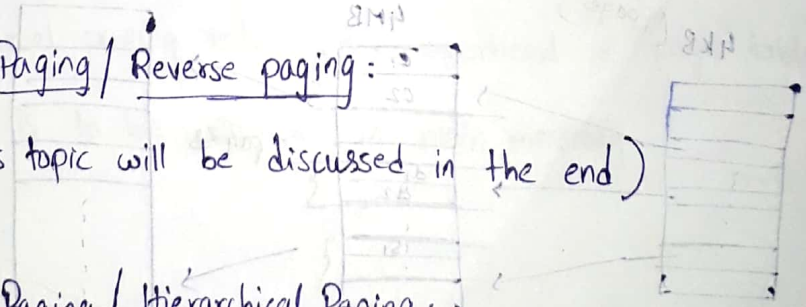
∴ Hash page table will have only 32 entries

However normal page table has 2^{20} entries.

→ However hash page table may require extra time for searching through the list. So hash page table is space efficient but not time efficient.

(iii) Inverted Paging / Reverse paging:

(This topic will be discussed in the end)



2. Multi-Level Paging / Hierarchical Paging:

* The basic objective of multi-level paging is to reduce page table size overhead.

* Multilevel paging is paging on page table.

* Paging generally involves 3 steps:

(i) divide the address space in pages

(ii) Store the pages into frames of memory

(iii) Access the pages of the address space through page table.

For example

Consider

LA = 32 bits , PS = 4kB

Now $N_{pages} = 2^{20} = 1M$ pages

> without multilevel paging, the page table must be stored contiguously if it spans across several pages

Assume size page table entry = 4 bytes

∴ PTS = 4MB

Now the page table itself is more than size of one page

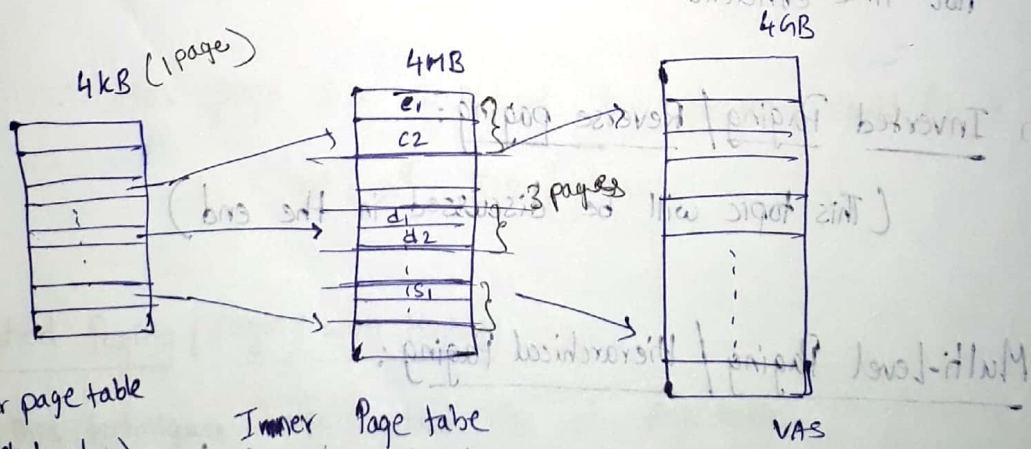
PT must be stored in $\frac{4MB}{4kB} = 1k$ pages

Now we page the page table using another page table (Outer page table)

Size of outer page table = 1k × 4 bytes (→ no of pages in inner page table)

= 4kB

i.e., 1 Page exactly



Outer page table (1st level page table) (Page Directory)

Inner Page tabe (2nd level page table)

→ Now in the above sceneria, assume we have a process of size 20 kB.

i.e., $\frac{20kB}{4kB} = 5$ pages

~~a normal page~~

Assume these 5 pages are

(2 code + 2 data + 1 stack)

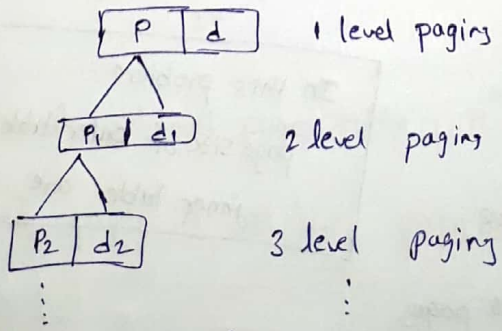
→ Assume these 3 parts are present in 3 different location such that code is in 2 contigious pages, data is in 2 contigious pages and stack in one page such that we need 3 pages of page table. (As shown in the figure)

→ Now in the case of multilevel paging we need outer pagetable, and the 3 pages of inner page table for the execution of the processes. (i.e., 4KB).

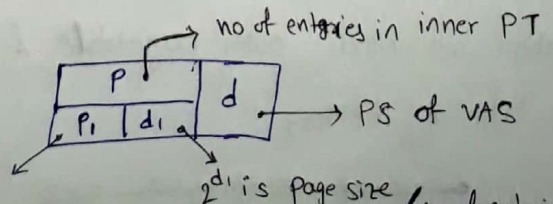
→ If we don't use multilevel paging then we must have all 1M entries of page table (i.e., 4MB) in main memory.

→ Thus multilevel paging reduces the overhead of page table size that is to be stored in main memory.

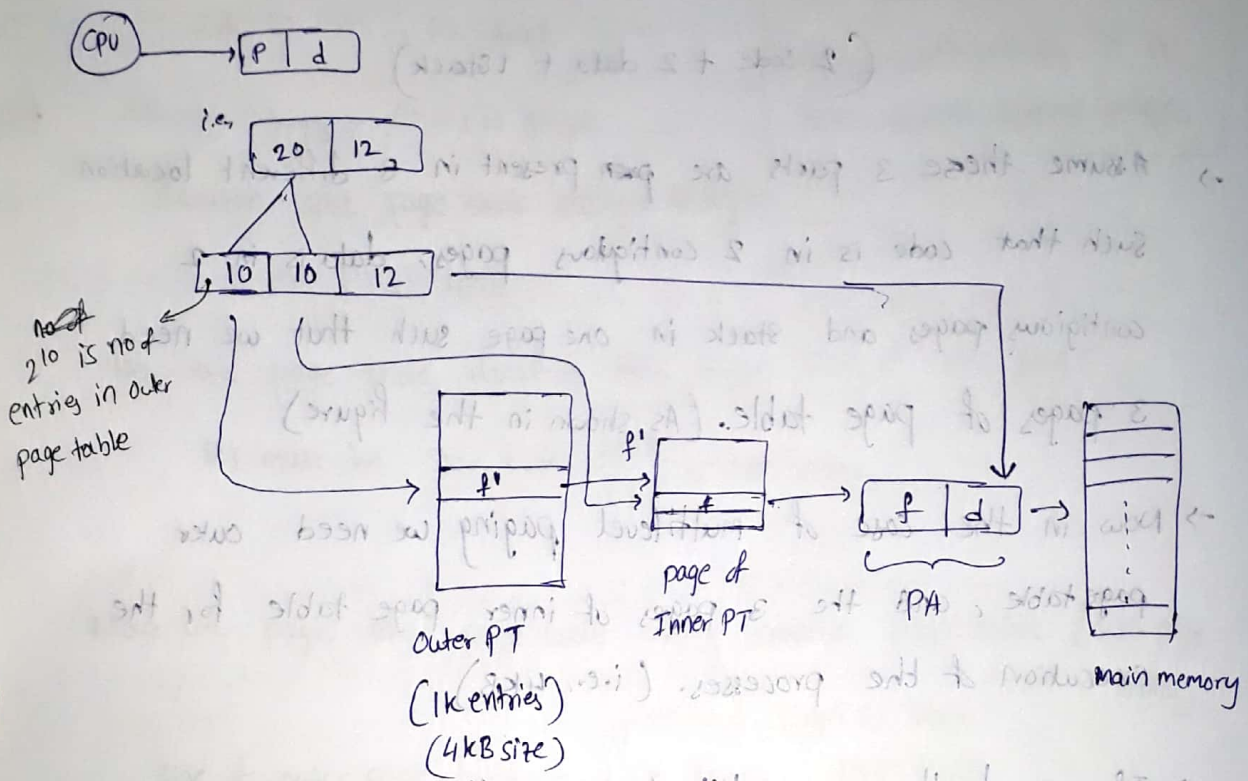
→ Addressing:



∴ Final logical address format for 2 level paging is



VA = 32 bits PS = 4kB

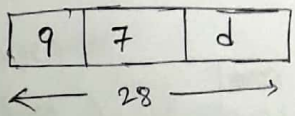


→ Here f' is frame of main memory in which required page of inner page table is located.

→ f is frame of main memory in which required page of a process is stored

→ For there offset 'd' is used to obtain required word.

H4 /12



$\Rightarrow d = 28 - 16 = 12$

(i) $\Rightarrow$ pg size $= 2^{12} = 4$ kB

$N_{pages} = \frac{2^{28}}{2^{12}} = 2^{16}$ pages

In this problem page size of outer table & inner table are different

(ii) for translation we need outer pagetable & one page of inner page table in memory

i.e. ~~2 page = 2 * 4kB = 8kB~~

$2^9 * 4B + 2^7 \times 4B = 2^{11} + 2^9$

$= 2^9(5)$

$= 512(5)$

$= 2560$ bytes

$= 2.5$ KB

18/08/20

* If we use 2-level paging, we require 3 memory accesses in total.

~~Memory~~

∴ Effective memory access time = 3m without TLB

* Silly for n-level paging without TLB

EMAT = (n+1)m

(H4/13) VA = 46 bits ⇒ VAS = 2^{46} = 64 TB

PTES = 4 bytes = 2^2

let PS = 2^x

$N_{pages} = \frac{2^{46}}{2^x} = 2^{46-x}$

Size of 3rd level page table = $2^{46-x} \times 2^2 = 2^{48-x}$ bytes

no of pages in 3rd level page table = $\frac{2^{48-x}}{2^x} = 2^{48-2x}$

size of 2nd level page table = $2^{48-2x} * 2^2 = 2^{50-2x}$

no of pages in 2nd lvl PT = $\frac{2^{50-2x}}{2^x} = 2^{50-3x}$

Size of 1st lvl page table = $2^{50-3x} \times 2^2 = 2^{52-3x}$

Given that

Size of 1st lvl page table = size of frame

i.e., $2^{52-3x} = 2^x$

$\Rightarrow 52-3x = x$

$\Rightarrow x = 13$

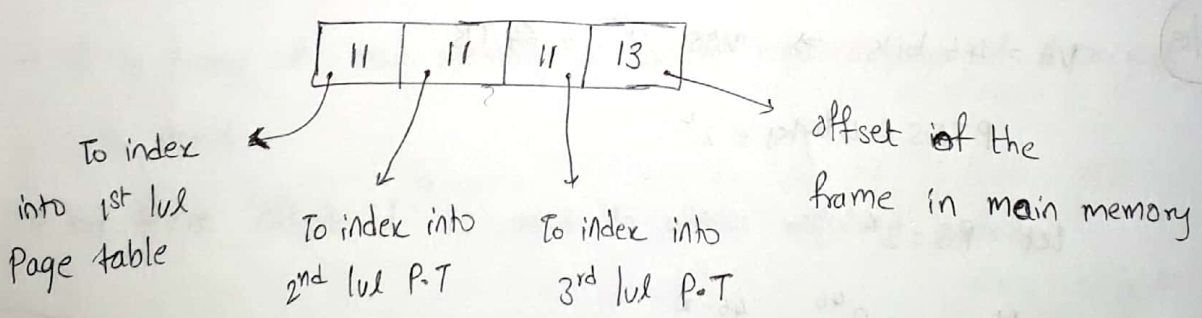
$\therefore$ page size = $2^{13} = 8kB$

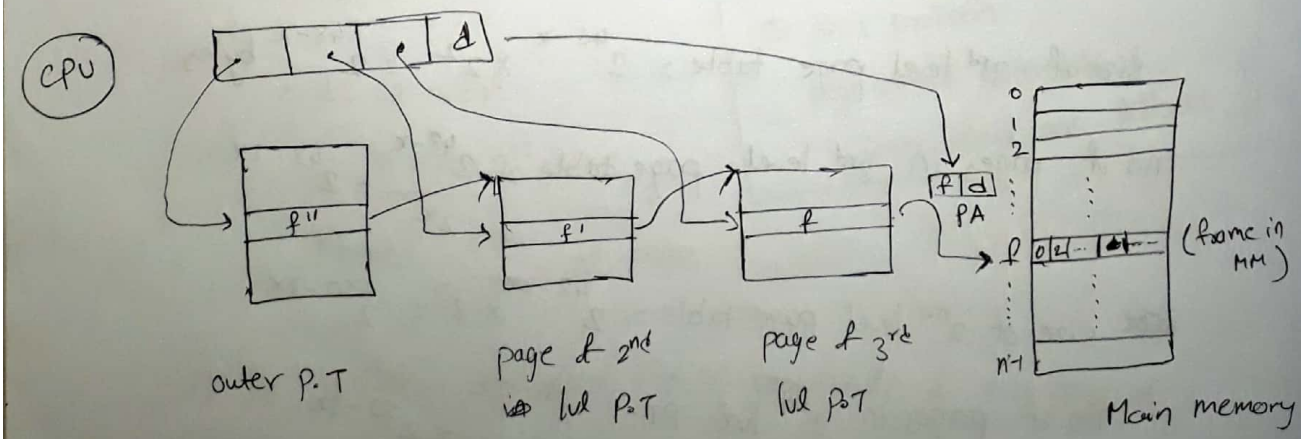
Page table entrie size is same at all levels of pagey tables.

$\therefore$ no of PT entries in one page of page table

$= \frac{2^{13}}{4} = 2^{11}$

$\therefore$ logical address format





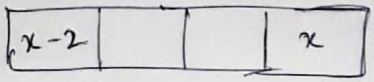
Method 2:

Let 2^x be req. PS

Size of outer (1st lvl) P.T = 2^x

no of entries = $\frac{2^x}{PTES} = \frac{2^x}{4} = 2^{x-2}$

LA:

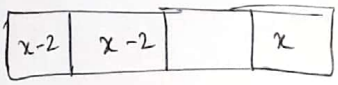


Silly after indexing to outer P.T, we go to a page of 2nd lvl

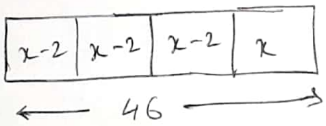
P.T

no of entries in a page 2nd lvl PT = 2^{x-2}

∴ LA



Silly we get



∴ $3(x-2)+x = 46$

$\Rightarrow 4x = 52 \Rightarrow x = 13$

∴ Page size = $2^x = 2^{13} = 8$ kB

From the above question we can come up with a general formula for P.T with n level paging

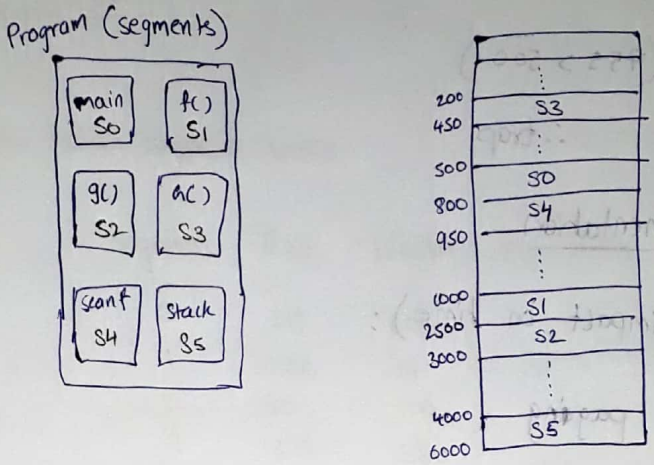
Assum VAS = 2^s bytes ; PS = 2^x bytes ; PTE = 2^c bytes

Size of OPT n-level paging = $\left\lceil 2^{s-nx+nc} \right\rceil$ bytes

(with uniform pages at all levels)

II. Segmentation

Paging does not preserve users view of program



→ At user view program is divided into segments (each segment may be of diff sizes).

→ Each segment has segment id - (assigned by loader)

→ Different segments a stored non-contiguously. However a segment itself is stored totally at one location (contiguously)

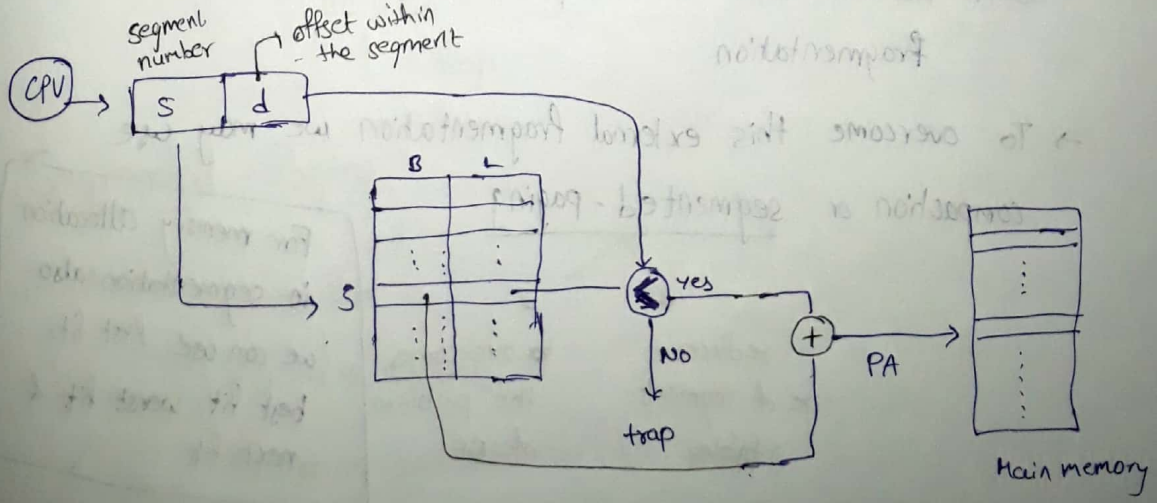
→ Here MMU is segment table.

	B	L
0	500	300
1	1000	1500
2	2500	500
3	200	250
4	800	150
5	4000	2000

B - Base address of segment

L - length of segment

→ Every process has its own segment table



Eg: For above example

LA<3,50> ⟶ PA<200+50> = 250

Eg: LA<2,755>

└(755 > 500)

∴ trap

Performance of Segmentation

(i) Temporal issue (impact on time):

* Exactly same as paging

i.e. EMAT = 2m without TLB

Slly we can add TLB & PAC (same equation as in the case of paging)

(ii) Spatial issue

* Large segment table is undesirable

* One of ways to reduce size of segment table is to use paging on segment table.

Note:

* In case of segmentation, we don't divide PAS.

∴ PAS is organized with variable partitioning (MVT)

~~So~~ we know that variable partitioning suffers from external fragmentation

→ To overcome this external fragmentation we may use compaction or segmented-paging

- reduces size of segment table
- to overcomes the problem of EF

For memory allocation in segmentation also we can use first fit, best fit, worst fit & next fit

Note:

	IF	EF
Paging	✓	X
Segmentation	X	✓

IF - Internal Fragmentation
EF - External Fragmentation

Q31) Consider below segment table

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

what the physical addresses for following logical addresses

a) 0,430

219 + (430 - 0) = 219 + 430 = 649

b) 1,10

2300 + 10 = 2310

c) 2,500

500 > 100

∴ trap

d) 3,400

1327 + 400 = 1727

e) 4,112

112 > 96

∴ trap

Segmented - Paging:

Assume LA = 34 bits

<s,d> = <18, 16> bits

⇒ Maximum segment size = 2^{16} = 64kB

total no of segments possible = 2^{18} = 256k segments.

→ To avoid EF we apply paging on segment.

(i) Divide address space (segment) into pages

(ii) Store these pages in frames.

(iii) Access the frames through page table of a segment

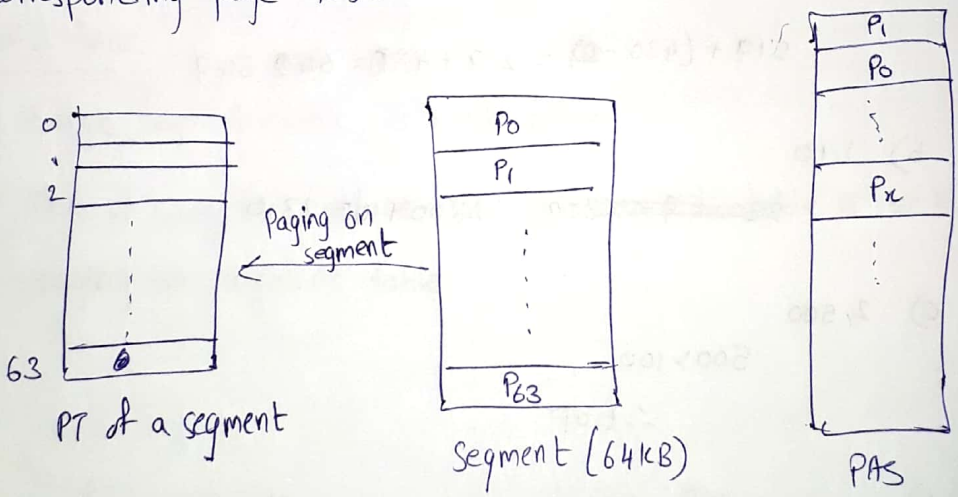
Let page size = 1 KB

no of pages in a segment = $\frac{64\ KB}{1\ KB}$ = 64 pages

→ Now PAS contains frames but not segments.

→ Now the page table of segment contains an entry for each page of the segment

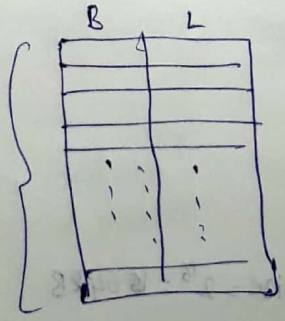
→ Now every segment must be accessed through its corresponding page table.



no of Page tables = no of segments.

Thus we have 256k page tables.

Structure of segment table:

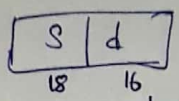


256k entries
(i.e., no of segments
= no of page tables)

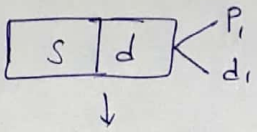
B - base address of corresponding page table of the segment

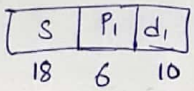
L - length of segment

logical address format



we have applied paging on segment i.e., on 'd'

∴ 

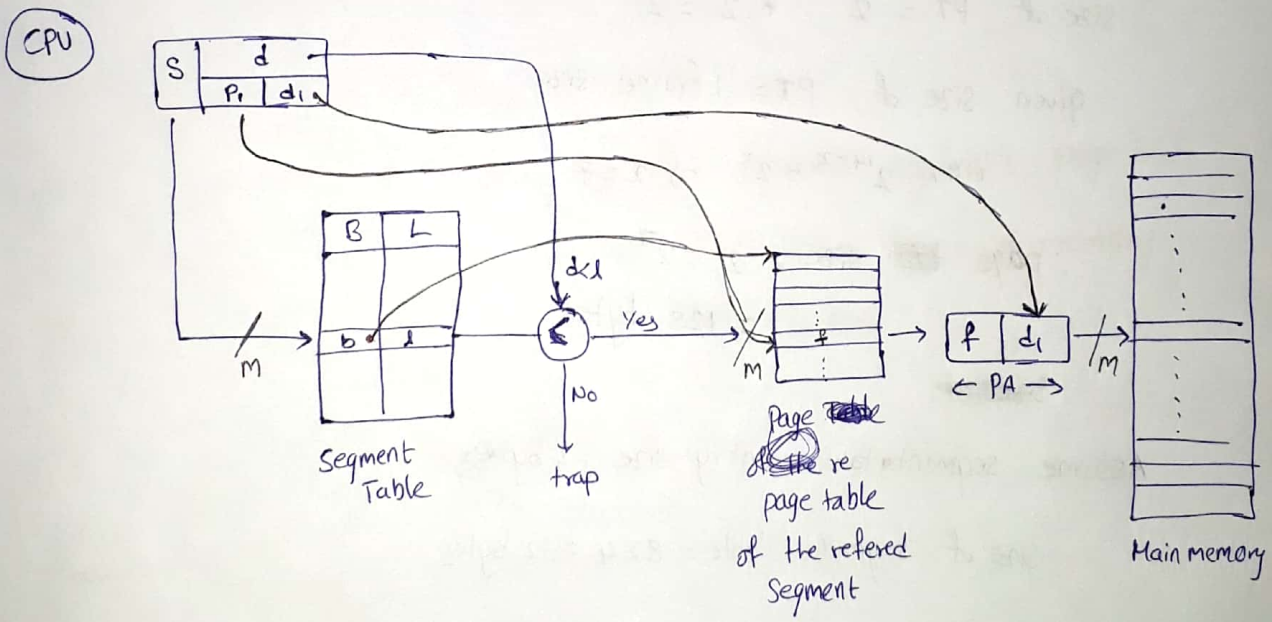


2^{P_1} - no of entries in each page table

2^{d_1} - size of page

d_1 say offset within the page

Address Translation



* Here we need to access segment table, then corresponding page table then the required data.

i.e., 3 memory accesses

∴ EMAT = 3M

Like in the case of paging we can use TLB to reduce EMAT.

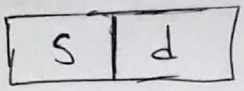
Here also the TLB contains VA & corresponding PA.

(14/14) VAS = PAS = 2^{16} bytes = 64 kB

no of segments in VAS = 8

size of each segment = $\frac{2^{16}}{2^3} = 2^{13}$ = 8 kB

VA:



Pg table entry size = 2 bytes

Let page size = 2^x

no of pages in a segment = $\frac{2^{13}}{2^x} = 2^{13-x}$

i.e., no of entries in a PT = 2^{13-x}

Size of PT = $2^{13-x} * 2 = 2^{14-x}$

given size of PT = 1 frame size

i.e., $2^{14-x} = 2^x \Rightarrow x = 7$

∴ page size = $2^x = 2^7$

= 128 bytes

Assume segment table entry size = 4 bytes

size of segment table = 8 × 4 = 32 bytes

Virtual Memory (VM):

VM gives an illusion to the programmer that a huge amount of memory is available for executing larger programs in small memory area.

→ It is a technique that supports the goal of managing execution of larger programs in small memory area.

→ CPU assumes that it can executes a program as big as VAS.

→ However PAS may not be as big as VAS.
So we need to store the program on disk.
From disk we load required pages into memory.

→ Sometimes required page may not be found in main memory. In that case required page is loaded into memory from the disk. This is known demand paging.

Demand paging:

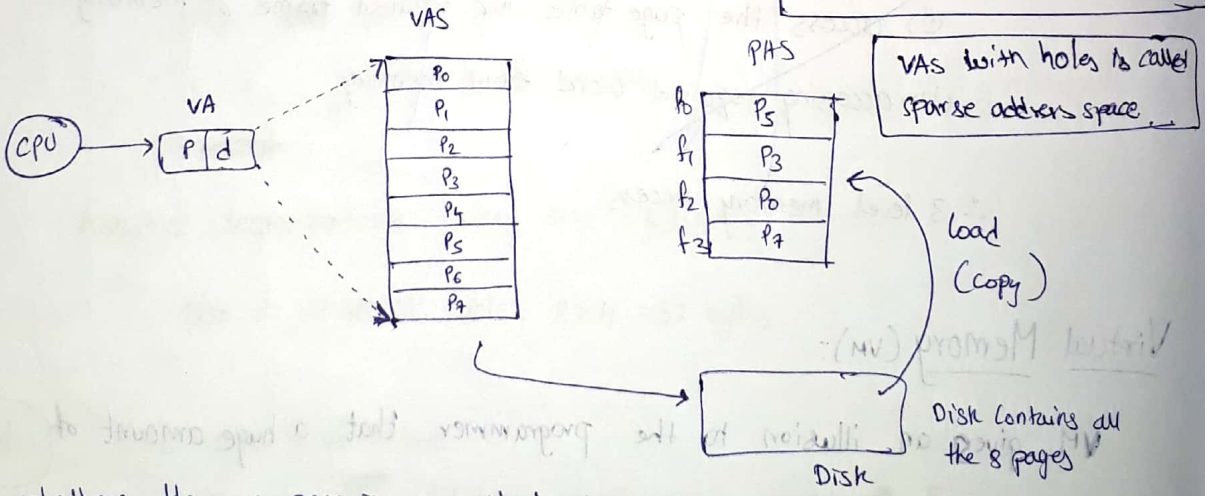
Loading page from disk to memory at runtime based on demand is known as demand paging.
i.e., Virtual memory is implemented through demand paging.

VM implementation through Demand Paging:

Let VAS = 8kB ; PAS = 4kB ; PS = 1kB

⇒ N = 8 M = 4 d = 10

Sharing:
→ System libraries are shared by mapping system libraries into VAs
→ Silly processes can share mem for process communication.
Shared memory region is considered to be part of both the processes' VAS



VAs with holes is called sparse address space.

→ Whether the req page is present in the memory or not is known by valid/invalid (V/I) bit which is in the page table.

→ If required page is found in the memory then it is called page hit.

→ If required page is not found in the memory then it is called page miss or page fault.

0	10	1
1	–	0
2	–	0
3	01	1
4	00	0
5	00	1
6	–	0
7	11	1

Page table

Page fault service:

1. Process that caused page fault gets blocked
2. Virtual memory manager now starts running (in kernal mode). Virtual memory manager tells disk manager about which page is missing.
3. Disk Managers communicates with device driver.
4. Device driver reads the page (through disk mechanism) and transfers required page to OS using DMA.
5. OS will try to load the page in one of the frames of user process.
6. If there is no empty frame to load new page then virtual memory manager has to initiate page replacement through which one of the pages in memory is replaced with required page. Here we have 2 cases:
   (i) page that is selected for replacement is not modified then we can directly overwrite it with new page. (1 disk operation)
   (ii) If it is modified (dirty page) then the modified page has to be saved in the disk. Later the new paged is loaded (i.e., 2 disk operations)
7. After the new page is loaded we need to update page table (Eg: changing V/I bit etc.)
8. Unblock the user process

   Now when the process is scheduled again, it generates the same VA that it had generated previously and continues its execution.

This is VM implementation through demand paging.

→ page fault service time is time taken to service a page fault.

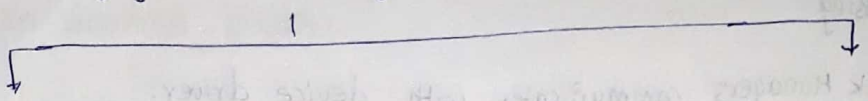
~~Hence~~ ~~we need to reduce~~

This service time reduces throughput.

So we need to reduce page fault rate.

If there is a TLB hit, then there is no page fault

→ Demand paging is of 2 types:



~~Pure Demand~~

Pure Demand paging

→ Execution starts with no ~~pages~~ pages in the main memory

Prefetched Demand paging

→ Execution starts with loading a few pages

(By default we consider pure demand paging while solving questions)

(Q32) The size of virtual memory supported is eventually limited by size _____

Ans:

size of disk (secondary storage)

(Q33) ~~What~~

Note:

For real time implementation of virtual memory

PAS ≤ VAS ≤ Disk AS

Performance of Virtual Memory

(i) Temporal Issues: (time)

Assume

main memory access time = 'm'

page fault service time = 's' & (s >>> m)

page fault rate = 'p' ($0 \le p < 1$)

page hit rate = 1-p

EMAT with Demand paging	= (1-p) m + p*s

(∵ s >>> m we don't take s+m)

In general pg. fault service time may range in milliseconds.

H4/16

PFST = 10ms

Mem. AT = 1 μs

page hit rate, $P = \frac{99.99}{100} = 0.9999$

$$EMAT = \frac{99.99}{100}(1) + \frac{0.01}{100}(10 \times 10^3) \ \mu s$$

$$= \frac{99.99}{100} + \frac{99.99}{100}$$

$$= \frac{99.99}{100} + 1$$

$$= 1.9999 \ \mu s \cong 2 \mu s$$

H4/17

page fault rate = $\frac{1}{k}$

page hit rate = $\frac{k-1}{k}$

Effective instruction time = $\frac{k-1}{k} i + \frac{1}{k}(i+j) = \frac{(k-1)i}{k} + \frac{i}{k} + \frac{j}{k}$

$$= i + \frac{j}{k}$$

H4/18

PFST = 8 ms (empty frame or unmodified frame)

= 20 ms (modified frame)

Mem. AT = 100 ns

70% of the page to be replace is modified the time.

required EMAT ≤ 2000 ns

Let p be page fault rate

EMAT = (1-P)(M) + P(0.7(20) + 0.3(8))

= (1-P)(100) + P(14 + 2.4)

= 100 - 100P + 16.4P

⇒ 100 - 83.6P ≤ 2000 ns

$EMAT = (1-P)(100) + P(0.7(20\times10^6) + 0.3(8\times10^6))$

$= 100 - 100P + P(14\times10^6 + 2.4\times10^6)$

$= 100 - 100P + 16.4P\times10^6$

$\Rightarrow (164\times10^5 - 100)P \le 2000 - 100$

$\Rightarrow (164000 - 1)(100P) \le 1900$

$P \le \frac{19}{163999}$

max acceptable page fault rate = $\frac{19}{163999}$

H4/19

Mem. AT = M (page hit)

= D (page fault)

for some process

EMAT = X units.

page fault rate, P=?

$$X = (1-P)M + P(D)$$

$$X = M - MP + PD$$

$$\Rightarrow P(D-M) = X - M$$

$$P = \frac{X-M}{D-M}$$

Note:

Considering TLB & page fault rate the effective memory access time is

$$EMAT = x(c+m) + (1-x)\left[(1-P) * m + P * s\right]$$

$m + c$ ↳ (negligible)

x - hit ratio of TLB

P - page fault rate

(ii) page replacement policies:

Page reference string or reference string:

* set of successively unique pages refered in the given list of virtual addresses.

Eg: VAs: <702; 755; 864; 084; 122; 360; 768; 934; 015; 125; 654>

Let page size = 100

Also all the above addresses are in ~~dec~~ decimal

page P_0 contains addresses 0 to 99

P_1 " " 100 to 199

⋮ ⋮

page number = $\left\lfloor \frac{VA}{PS} \right\rfloor$

page offset = VA % PS

∴ Reference string for above VA's is

⟨7,8,0,1,5,7,9,0,1,6⟩

(successive page numbers are unique)

Every reference string has 2 parameters:

(i) length (i.e, 10 in above example)

(ii) no of unique pages (n) (i.e, 7 in above example)

Frame Allocation Policies

Assume we have

'n' no of process $(P_1 \dots P_n)$ -

demand of each process P_i be S_i no of frames

total demand for frames = $D = \sum_{i=1}^{n} S_i$

total frame available be: M $(D > M)$

frames allocated to process P_i be a_i

Eg: Let

n=5, M=50

	S_i	a_i ①	a_i ②
P_1	5	10	3
P_2	20	10	10
P_3	44	10	22
P_4	18	10	9
P_5	12	10	6
D:	100		

① Equal allocation policy

Every process gets equal no of frame

$\therefore \frac{50}{5} = 10$ frames

② Proportional allocation

no of frames allocated is proportional to size

i.e., $\frac{S_i}{D} \times M$ no of frames for process P_i

③ x% rule

Every process is given x% of the frames it has demanded.

This x is choosen based on main memory size and other factors.

Page Replacement Techniques (pure demand paging by default)

Consider for a process

reference string : 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

length of reference string, $l = 20$

no of unique pages, $n = 6$

1) FIFO:

Assume no of frames allocate for the process = 3

based on the time we loaded we replace

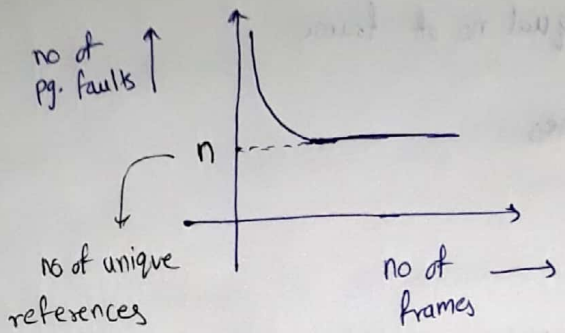
7	7	7	2	2	4	7
	0	0	0	3	3	3
		1	1	1	0	0

} frames

from above figure we have 15 page faults

$\therefore$ page fault rate $= \frac{15}{20} = 0.75$

If we increase no of frame allocated, page faults decreses



If no of frames allocated = 4

7	7	2									
0	4	7									
1	0										
2	1										

∴ 10 page faults

page fault rate = $\frac{10}{20} = 0.5$

Ref-string II : ⟨1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5⟩

$l = 12$

$n = 5$ (uniq. ref)

FIFO

no of frame allocate = 3

1	4	5									
2	1	3									
3	2	4									

∴ 9 page faults

Pg fault rate = $\frac{9}{12} = 0.75$

if no of frames allocated = 4

1	5	4									
2	1	5									
3	2										
4	3										

∴ page faults = 10

page fault rate = $\frac{10}{12} = 0.83$

In this example we got more page faults with more no of frames which is an anamoly.

This is known as Belady's anamoly

↳ with increase in no of frames to the process, the page fault rate increases

→ Beladys Anamoly is observed only in FIFO & FIFO based replacement.

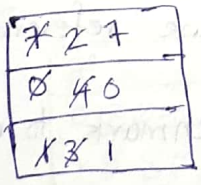
19/08/20

2) Optimal replacement:

Optimal replacement says that replace that page which will not be used for longest duration of time in future references.

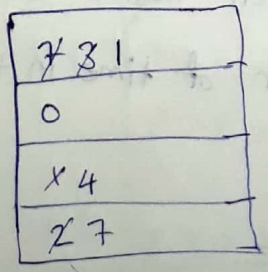
ref string: < 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 >

3 frames:



∴ 9 page faults

4 frames:



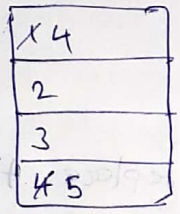
∴ 8 page faults

Ref string : <1,2,3,4,1,2,5,1,2,3,4,5>

3 frames :

∴ page faults = 7

4 frames :



∴ page faults = 6

→ Optimal replacement ~~gener~~ has the least page fault rate.

Drawback :

→ Optimal page replacement is not practically implementable since we can never know the future references.

→ However this can be used as benchmark to measure the performance of other algorithms.

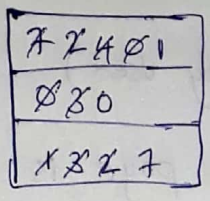
3) Least Recently Used (LRU) :

→ LRU replaces that page which hasn't been refered for the longest duration of time in the past.

∴ selection criteria : Time of reference

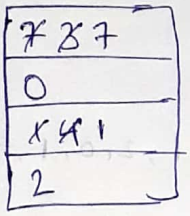
Ref string: <7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1>

3 frames:



∴ no of page faults = 12

4 frames:



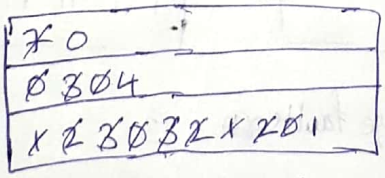
∴ no of page faults = 8

4) Most Recently Used: (MRU)

* Most recently used page is replaced

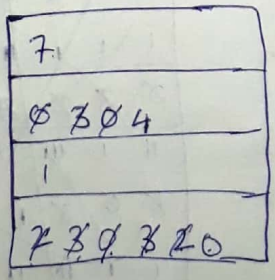
Ref string: <7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1>

3 frames:



∴ no of page faults = 16

4 frames:



∴ no of page faults = 12

5) Counting Algorithms

- LFU (Least Frequently Used)
- MFU (Most Frequently Used)

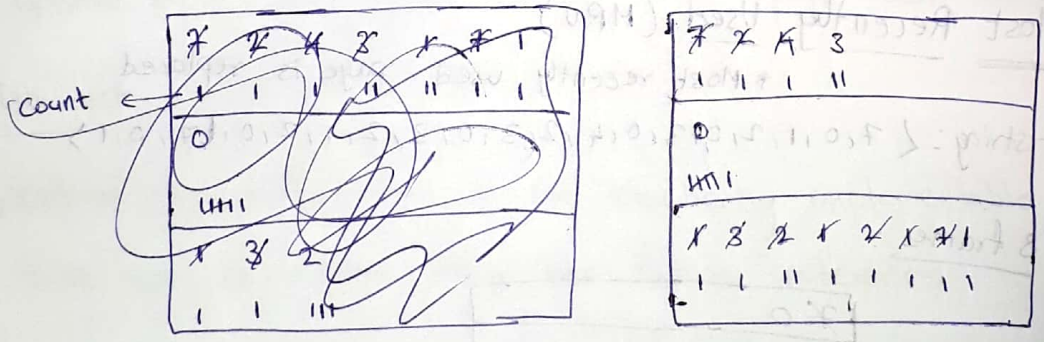
* Frequency means no of times the page is ~~refren~~ refered since it has been brought into memory. (i.e., if we load a page that has been loaded previously we start the count from 0)
* FIFO may be used ~~to~~ to resolve a conflict

~~3 frame~~ :

Ref string: < 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 >

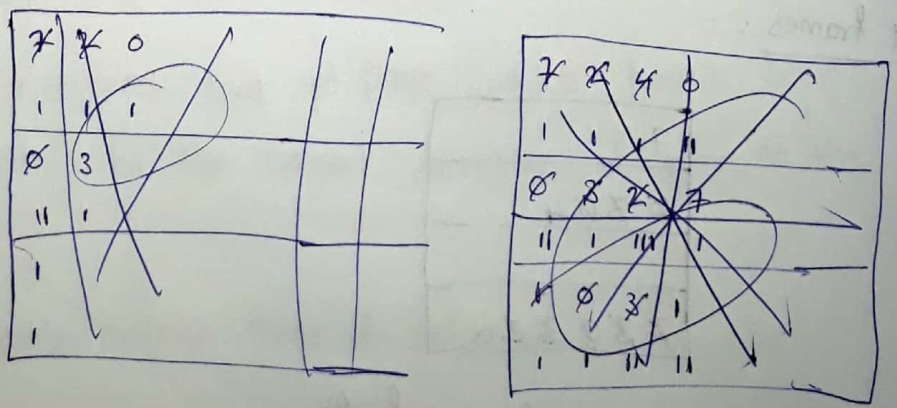
3 frames

LFU :



no of page faults = 13

MFU :

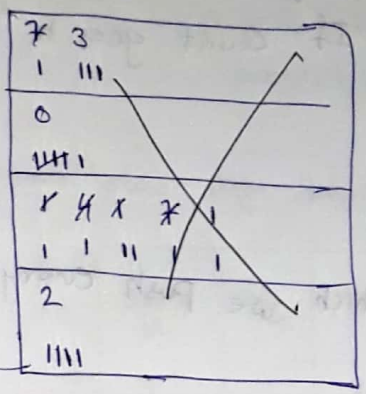


∴ no of page faults = 15

4 frames :

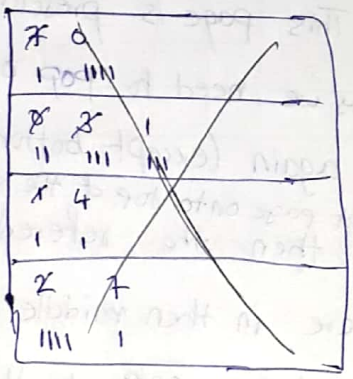
LFU :

Queue: 7 0 1 2 3 4 1



no of page faults = 9

MFU :



no of page faults = 9

12 is Ans

Note :

→ Practically LRU has been found to give less page faults (i.e., best performance & close to optimal)

Implementation of LRU :

* Here we need to monitor time of reference (TOR)

There are two methods to monitor time of reference

(i) Counter method :

we maintain a counter register which is incremented for every memory reference. whenever we refer a page we assign the value of counter as time of reference for that page. Initial value of counter is 0.

Limitation:

* An n-bit counter can count only 2^n values after which count starts from 0. If count goes to '0' then the algorithm fails.

(ii) Stack method

→ Here we use a stack into which we push every new page we replace.

→ At the time of replacement, we need to replace the least recently used page. This page is present at the bottom of the stack. Thus we need to pop out all the elements and push them again (except bottom element) and finally push the latest page onto top of the stack.

→ If a page hit occurs, then the refered place might be present somewhere in then middle of the stack. This element has to be on to the top of the stack for which we need to perform push & pop operations again.

Advantage:

* Unlike counter method, it works good in any case.

Disadvantage:

* push & pop operation requires more time.

LRU Approximations:

* LRU approximations are the class of replacement techniques that pretend to work like LRU. i.e., they approximate to the behaviour of LRU.

we have 4 types of LRU approximations

i) Reference bit (R):

R < 0 : page not refered so far during present epoch
R < 1 : page is refered ~~so far~~ atleast once so far during present epoch.

Consider below page table

Pg No	frame No.	V/I	TOL	R
0	a	1	2	1
1	b	1	4	0
2	–	0	–	–
3	c	1	0	1
4	d	1	3	1
5	e	1	1	0

while implementing this algorithm we may not have TOL column

(TOL - Time of loading ~~reference~~)

(Here in the Pg table Pg P_1 & P_5 are not refered in the present epoch)

* Time is divided into equal interval and each interval is called an epoch.

* when an epoch is finished all the 'R' values are cleared.

Algorithm:

→ * If page hit occurs, then set 'R' of the corresponding page table entry.

→ If page fault occurs,

Start searching ~~for R=0~~ from the first entry for R=0 and replace the first encountered entry's corresponding page.

→ If 'R' value is one for all the pages then the algorithm fails.

* To overcome ~~on~~ this limitation we use next LRU approximation which is additional reference bits.

~~(ii) Additional~~

(ii) Additional Reference bits:

Here we have more than one reference bits, say 'n' indicating past 'n' references of past 'n' epochs

For example take n=8

	R_1 R_2 ..... R_7 R_8
P_i :	10011111
P_j :	01101111
R_k :	10111111

* Least significant bit corresponds to current epoch
* Most significant bit corresponds to most previous epoch

For above example if a page fault occurs P_j is replaced.

(∵ the 4 least significants of the 3 processes are '1'

However R_4 of P_j is 0 and R_4 of P_i & P_k are 1

∴ Replace P_j)

* After every epoch we need to left shift the reference bits by one bit.

* However this method also has a chance to fail when all the reference bits of all the processes are 1's. Anyway the probability for this is very less.

(iii) Second Chance (clock Algorithm) (Read from TB)

Selection criteria: (Time of loding & Reference bit)

→ The page with R value as 0 and least TOL (early loaded page among R=0 pages) is replaced.

The pages whose TOL is less than replaced page, their R value is set to '0'.

Also the arrival time of those pages is set to current time

→ when all R values are 1's, the earliest loaded page is replaced.

i.e., The algorithm, when all R values are 1, degenerates to FIFO

* Hence this algorithm (FIFO based) also suffers from Belady's anomaly.

(ii) Enhanced Second chance / Not recently used (NRU): (Read from TB)

Selection criteria: (Reference bit (R) & modified bit (M))

Modified bit (or) Dirty bit < 0: page is not modified since its loading
1: page is modified since the time of its loading

R	M	
0	0	(I)
0	1	(II)
1	0	(III)
1	1	(IV)

← This means the page is not refered in this epoch but is refered, and modified in some previous epoch

* If a page

then page with RM=00 is replaced

if no such page RM=01 is replaced

if no such page RM=10 is replaced

if no such page RM=11 is replaced

* The above search starts from the first entry.

Page Buffering Algorithms:

(i) Maintaining free frame pool:

i) choose victim; ii) read page into one of free frame from pool; iii) victim is written out later. (This will help restart process quicker). victim is added to free frame pool.

we may even maintain list of modified pages and write them on disk and reset their modified bit. This ↑ probability of find clean frame

we can even keep track of page present in frames of free frame pool. So that if same page is req it doesn't need to be read in. This all will help in ↓ penalty for replacing victim page.

H4/20

length of ref string: L

no of unique pages: k

allocated frames: Z

* In the worst case every reference can be a page fault

i.e., L

* Since it is pure demand page we must have atleast k page faults

~~In the best case we can have all page hits after loading the servicing the last page fault~~

Eg: Let z=3, k=5

Ref string: 1, 2, 3, 4, 5, 4, 5

∴ k

In the best case the above could happen.

H4/21

Ref ~~string~~: $P_1, P_2, P_3, P_4 \cdots P_{n-1}, P_n, P_n, P_{n-1}, \cdots P_3, P_2, P_1$

↳ (This is not reference string since we have Pn twice successively)

After refering till Pn we will have

$P_n, P_{n-1}, P_{n-2} \cdots P_{n-(k-1)}$ in the frames

∴ The next k references $(P_n, P_{n-1}, P_{n-2} \cdots P_{n-(k-1)})$ will be hits

total references = 2n

no of hits = k

∴ no of page faults = 2n − k

H4/22 Ref string: ~~1, 2, 4, 5, 1, 2, 2~~, 2,

1, 2, 4, 5, 1, 2, 3.

∴ length of refstring = 7

no of frames allocated = 1

∴ 7 page faults

H4/23

VAS = PAS = 2^{16} bytes = 64 kB

PS = 512 bytes = 2^9 bytes

PTES = 4 bytes = 32 bits

no of frames = $\frac{2^{16}}{2^9} = 2^7$

no of bits to store frame number = 7
V/I = 1
R = 1
Modified = 1
Page Protection = 3
} 13 bits

$\therefore$ 32 - 13 = 19 bits

19 bits can be used to store other information

no of pages = $\frac{2^{16}}{2^9} = 2^7$

size of page table = $2^7 \times 4 = 2^9 = 512$ bytes

Thrashing:

* Excessive / high paging activity
  ↓
  act of servicing page fault

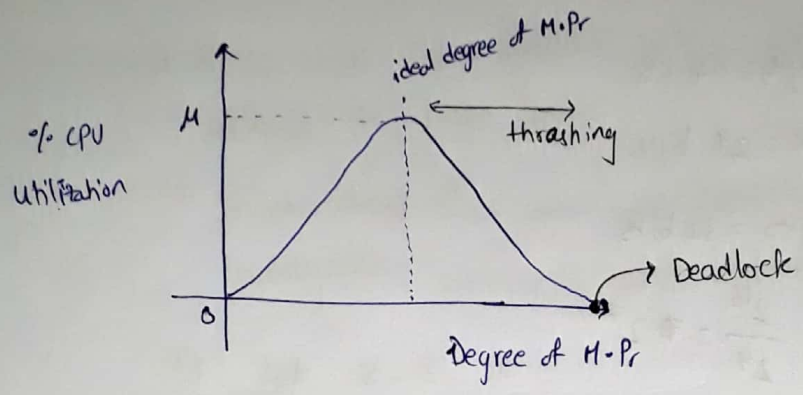
i.e., High page fault rate

* Thrashing is undesirable
* Thrashing cause more processes to block

Cause of Thrashing

→ Lack of frames (i.e., Lack of memory)
→ High degree of multiprogramming
} Primary reasons for thrashing

(we also have some secondary reasons which we will explore later)



Thrashing Control strategies:

Prevention

* Prevention is done by controlling degree of multiprogramming. (Controlled by long term scheduler)

Detect & Recover

Detection : * High degree M.Pr & low CPU utilization
=> thrashing
* High paging (high disk utilization)

The above 3 conditions confirms the presence of thrashing

Recovery: Decrease the degree of M.Pr
(This done by suspending the processes)

Secondary reasons for thrashing:

→ Page replacement policy

→ page size:
page size must be large to reduce page faults.
It is because with large PS we have less pages.
(Large PS => less pages => less page faults)

→ Programming techniques & Data structures used by programmers. This fact is explained through below case studies.

Case study I:

```
integer A[1..128][1..128];
```

1) for i ← 1 to 128
    for j ← 1 to 128
        A[j][i] = 1;

2) for i ← 1 to 128
    for j ← 1 to 128
        A[i][j] = 1;

Assume we store the array in row major order

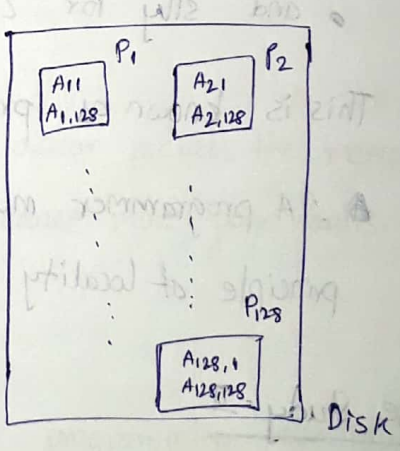
also let page size = 128 words

and each element of array is one word

⇒ each row is stored in one page exactly

Assume we are using pure demand paging and replacement technique FIFO.

→ Assume the OS has allocated 127 frames to the process.



page faults with 1st program:

→ A[1][1] A[1][2] .... A[1][128] A[2][1] ... A[128][1]

Here every reference causes a page fault

i.e., 128 page faults.

Silly for 2nd column we get 128 page faults

In this way

total no of page faults = 128 × 128

Page faults for 2nd program :

→ A[1][1], A[1][2] ... A[1][128]

A[1][1]: page fault; A[1][2] ... A[1][128]: page hit

∴ 1st row causes one page fault

Silly we get 1 page fault for 2nd row

∴ total no of page faults = 128

→ ∴ 2nd program is 128 times faster than that of 1st program

Note:

* If we use row major order to store 2d array then it is better to access in that order
  - and silly for column major order.

  This is known as principle of locality

* ∴ A programmer must use data structures that ensure principle of locality.

Case Study - II

* If we consider array & linked list,

  array allocation is contiguous

  linked list allocation is non-contiguous.

  ∴ linked list may cause more page faults.

* The elements of array obeys principle of locality.

Case Study III

Linear Search vs Binary Search

→ In demand paging, b binary search causes more page faults because we don't access elements contiguously.

whereas in linear search we search in contiguous way causing less page faults and obeying principle of locality.

Case Study - IV

Stack:

Stack operation which are push & pop are performed at the same end.

∴ less page faults

Hashing:

Hashing distributes key across the memory and is more likely to cause more page faults.

Note:

→ In general a Data Structure or a programming technique in demand paging is said to be good iff it obeys the concept of locality of reference.

Working Set Strategy

* The objective of working set strategy is to control thrashing and utilize memory effectively.
* Working set strategy is based on principle of locality of reference.

Consider below program

```
main()//10kB        f()//5kB        g()//30kB       h()//3kB
{                   {               {               {
    :                   :               :               :
    f();                g();            h();            printf();//25kB
    :                   :               :               :
}                   }               }               }
```

Total program size = 73 kB

Assume page size = 1kB

Assume the process has been allocated 35 frames

* This is static allocation.

→ As the process executes it changes from one locality to other locality (i.e., from one function to other function). When process is in main 10 frames would be enough. When it is executing in f(), 5 frames would be enough.

→ Thus allocating frames based on the locality in which the process is executing would reduce thrashing and utilizes memory more effectively.

→ To meet this requirement we use dynamic frame allocation technique.

* The working set strategy's working is based on the locality of reference.

It estimates the locality in which the process is running and no of frames are allocated accordingly.

* Estimating the size of locality:

Consider the reference string for process P_i

$\Delta = 10$

P_i : ⟨ 7, 0, 1, 2, 3, 0, 2, 16, 18, 22, 18, 16, 35, 38, 39, 34, 35, 37, 38, 36, 34, 36, 37 ⟩

Assume these references are generated in time t. After time t, we need to know the locality in which the process is executing and size of locality.

* observing the patterns we can know localities & size of localities.

* to estimate the size of locality we define a parameter called working set window (wsw) at time t for a process

Working set window$_i^t$ (wsw) = { set of unique pages refered in the past Δ references }

where Δ is an integer.

For example take Δ = 10

⇒ wsw_i^t = { 34, 35, 36, 37, 38, 39 }

⇒ $wsw\text{-}size_i^t = 6$

↓

size of locality is 6 pages

∴ 6 is the demand

Framework:

→ Let 'n' be no of processes

→ S_i : Demand for frames by process P_i (Estimated as shown in the previously example)
(i.e., wsw size$_i$)

→ total demand, $D = \sum_{i=1}^{n} S_i = \sum_{i=1}^{n} wsws_i$

→ Available frames = M

Case (i):

$D \cong M$

i.e., Balanced & no thrashing

Case (ii)

$D < M$

i.e., No thrashing & we can further increase deg of M.pr

Case (iii)

$D > M$

i.e., Thrashing

∴ Working sets strategy helps control thrashing and utilize memory better.

Note:

* The success of Working Set Strategy depends on the value of Δ.

* If Δ is too small ⇒ more page faults.

* If Δ is too large

⇒ we also hold pages of previous locality and this is ineffective use of memory

H4/24

page ref : c c d b c e c e a d

$\Delta = 4$

$WST_0 = \{a, d, e\}$

0

i.e., $\{e, d, a\}$ (-2, -1, 0)

∴ ref string: e d a c c d b c e c e a d

-2 -1 0 1 2 3 4 5 6 7 8 9 10

• • • • • ⇒ 5 page faults

t_0 : <e d a> : 3

t_1 : <e d a c> : 4 → At time t_1 'c' is refered since wsw at t_0 doesn't have 'c' it causes page fault

t_2 : <d a c> : 3

t_3 : <a c d> : 3

t_4 : <c d b> : 3 → Silly at time t_4 'b' is refered since 'b' is not present in wsw at t_3 a page fault is generated

t_5 : <d b c> : 3

t_6 : <d b c e> : 4

t_7 : <b e c> : 3

t_8 : <c e> : 2

t_9 : <c e a> : 3

t_{10} : <c e a d> : 4

Avg no of frames needed = $\frac{32}{10}$ = 3.2 frames