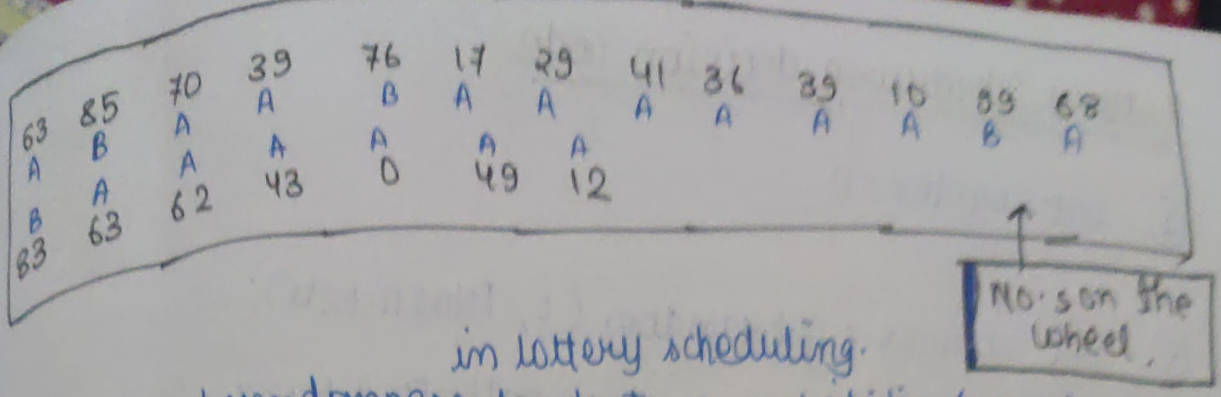
Proportional Share (PS)

It is based around a simple concept instead of optimizing for TAT or RT a scheduler might instead tried to guarantee that each job or process obtain a certain percentage of CPU time. A good example of PS scheduling is known as lottery scheduling.

Basic Concept

- **Tickets**: It is used to represent the share of a resource that a process should recieved. The percent of tickets that a process has represents its share of the system resource in problem.
  Ex: Consider two processes A and B and further that A has 75 tickets while B has only 25. Thus what we would like is for A to recieve 75% of the CPU and B the running 25%.
  Holding a lottery is straight forward the schedule must know how many total tickets there are (in eg: 100. Therefore the scheduler than picks the winning ticket which is a number from 0-99 assuming A holds tickets 0-74 and B holds 75-99, The winning ticket simply determines whether A or B runs.

~~The~~.



in lottery scheduling.

The use of randomness leads to a probilistic correctness in meeting the desired proportion. But no guarantee in this example we only gets to run ~~for 0~~ 4 out 20 times slides instead of the desired 25% of allocation

Ticket mechanism:

lottery scheduling also provides a no. of mechanisms to manipulate tickets in different & some time useful ways.

Concept of ticket currency: Currency allows a user with a set of tickets to allocate among their own jobs in whatever currency they would like. The system than automatically converts said currency into the correct global value.

Ticket transfer

With transfer a process can temporarily hand off its tickets to another process this availability is useful in a client-server setting where a process (client process) sent's a message to a server asking it to do some work on the clients behalf. To speedup the work the client can pass the tickets to the server and thus tried to maximise the performance of the server while the server is handling the clients request; when finished, the server than transfers the tickets back to the client and all is as before.

Lottery Scheduling decision code

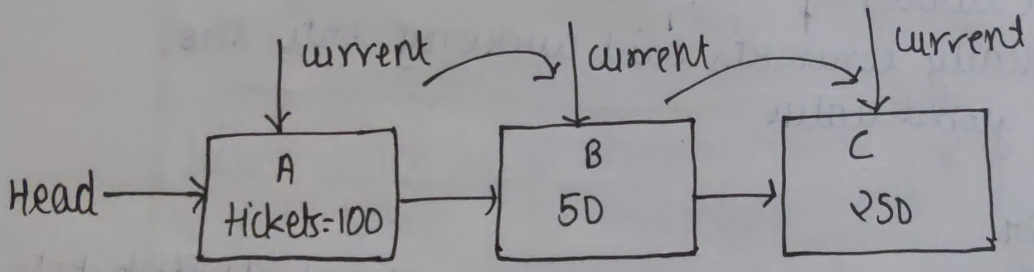
① int counter = 0

② int winner = getRandom (0, TotalTickets);

// winner use same call to a random number generation to get value, between 0 and the total tickets.

③ node * current = head;

④ while (current)
{
counter = counter + ~~counter~~ current → tickets;
if (counter > winner)
break;
current = current → next;
}



counter = 0
winner = 300

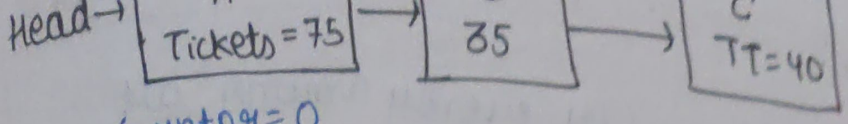
counter = 0 + 100
= 100

counter = 100 + 50

counter = 150 + 250
= 400

greater than winner

∴ process C start executing.



winner = 30

counter = 0

↓

counter = 0+75 = 75

counter > winner

Process A gets executing.

#Note: If counter & winner is same than condition become

```
if (counter >= winner)
    break;
```

Stride Scheduling. (Deterministic FSS)

A 100	B 80	C 40	who runs
0	0	0	A
100	0	0	B
100	80	0	C
100	80	40	C
100	80	80	

Completely fair Scheduling

The main focus of CFS is highly efficient and scalable manner. To achieve this goal CFS aims to spend very little time making scheduling decisions. Through both its inherent design and its clever use of datastructure is well suited to the task.

Basic Operation:

CFS, simply divides a CPU evenly among all computing process it does so through a simple counting based method known as virtual runtime. As each process runs it accumulates virtual runtime. In the most basic case each process vruntime increases at the same rate in proportion with real time. When a scheduling decision occurs CFS will pick the process with the lowest vruntime to run next.

VVIM

The main issue is how does the scheduler know when to stop the currently running process and run the next one?

(1) If CFS switches too often, fairness is increased as CFS will ensure that each process recieves its share of CPU even over miniscule time window but at the cost of performance.

(2) If CFS switches less often, performance is increased but at the cost of near term fairness

Control Parameters

4 Sched-Latency:

CFS uses this value to determine how long one process should run before considering a switch. Typically Sched-latency value is 48ms
CFS divides this value by the number (N) of processes running on the CPU to determine the time slice for a process and thus ensure that over this period of time CFS will be completely fair.
for example: If there are N=4 processes running then per process slice = $\frac{48}{4} = 12$ ms.
CFS uses this value to determine how long one process should run before considering a switch
CFS then schedules the first process and runs it until it has used 12ms of runtime and then checks to C if there is a process with lower vruntime to run instead. In this case there is, and CSF is switched to one of the other three jobs and so forth.
Eg: Consider 4 jobs A, B, C, D each run for two times slices in this fashion two of them (C & D) then complete, leaving first to remaining which each run for 24ms in round robin fashion.

(Q) what if there are too many processes running?

→ Here CFS acts another parameter min_granularity which is usually set to a value like 6ms. CFS will never set the time slice of a process to less than this value, ensuring that not too much time is spent in scheduling overahead.

For Example:

If there are (too many) 10 process are running over original calc. would divide Sched latency by 10 to determine the time slice (4.8ms).
However because of min-granularity CFS will set the time slice of each process to 6ms instead.

Note: CFS utilizes a periodic timer interrupt which means it can only make decisions at fixed time intervals this interrupts goes off frequently (1ms), giving CFS a chance to wakeup and determine if the current job has reached the end of its run. If a job/process has a time slice that is not ~~for a~~ perfect multiple of the time interrupt interval i.e CFS tracks vruntime precisely which means that over the long interval it will eventually approximate ideal sharing of the CPU.

Weighing or Niceness

CFS also enable controls over process priority from enabling users or administrators to give some processes a higher share of the CPU. It does this through a classic unix mechanism known as the Nicelevel of a process.

The Nice prameter can be set anywhere from -20 to +19 for a process with a default of zero.

+ve Nice values imply lower priority and -ve nice values imply higher priority. CFS Maps the nice value of each process to a weight as shown here.

```
static const int prio_to_weight[40] = {
 /* -20 */ 88761, 71755, 56483, 46273, 36291
 /* -15 */ 29154, 23254, 18705, 14949, 11916,
 /* -10 */ 9548, 7620, 6100, 4904, 3906
 /* -5 */ 3121, 2501, 1991, 1586, 1277
 /* 0 */ 1024, 820, 655, 526, 423
 /* 5 */ 355, 272, 215, 172, 137
 /* 10 */ 110, 87, 70, 56, 45
 /* 15 */ 36, 29, 23, 18, 15,
};
```

These weights allow us to compute the effective time slice of each process. But accounting for their priority differences the formula use to do so is as follows

assuming N processes.

$$\text{Time\_slice}_k = \frac{\text{weight}_k}{\sum_{i=0}^{n-1} \text{weight}_i} \text{ sched\_latency.}$$

Example

Consider there are 2 process A and B. A, because its more precious job is given a higher priority by assigning it a nice value of -5. ~~we~~ B, has default priority. (0).

weight of A = 3121 because it has priority -5
weight of B = 1024 " " --- 0.

$$\text{Time\_slice of A} = \frac{3121}{\text{weight}_A + \text{weight}_B} \times 48 = \frac{3121}{3121+1024} = \frac{3121}{4145} \times 48 = 36$$

$$\text{Time\_slice of B} = \frac{1024}{4145} \times 48 = 11.85 \text{ ms} \approx 12 \text{ ms}$$

In addition to the generalize the time slice calculation, the way CFS calc. vruntime must also be adopted.
for new formula vruntime - The actual runtime that process i has accured (runtimei) and scales it inversely by the weight of the process by dividing the default weight of 1024 by its weight of i.
As A's vruntime will accumulates at one third rate of B's

$$\text{Vruntime}_i = \text{Vruntime}_i + \frac{\text{weight}_0}{\text{weight}_i} \times \text{runtime}_i$$

Note: One unique aspect of the construction of the table of weights above is that the table preserves CPU proportionality ratio's when the diff. in nicevalue is constant

the main focus of CFS for a scheduler there are many facets of efficiency but one of them is as simples as this. when the scheduler has to find the next job to run. It should do so as quickly as possible. Simple data structure like list do not scale modern systems sometimes are comprised of many processes and thus searching through a long list every 50 milliseconds is wasteful.

CFS addresses this by keeping process in a red black tree. A red black tree is one of many types of balanced trees. In contrast to a simple binary tree (which can degenrate to list like performance under worst case insertion patterns) Balance trees do a little extra work to maintain low depth. and thus ensure that operations are logarithmic in time.

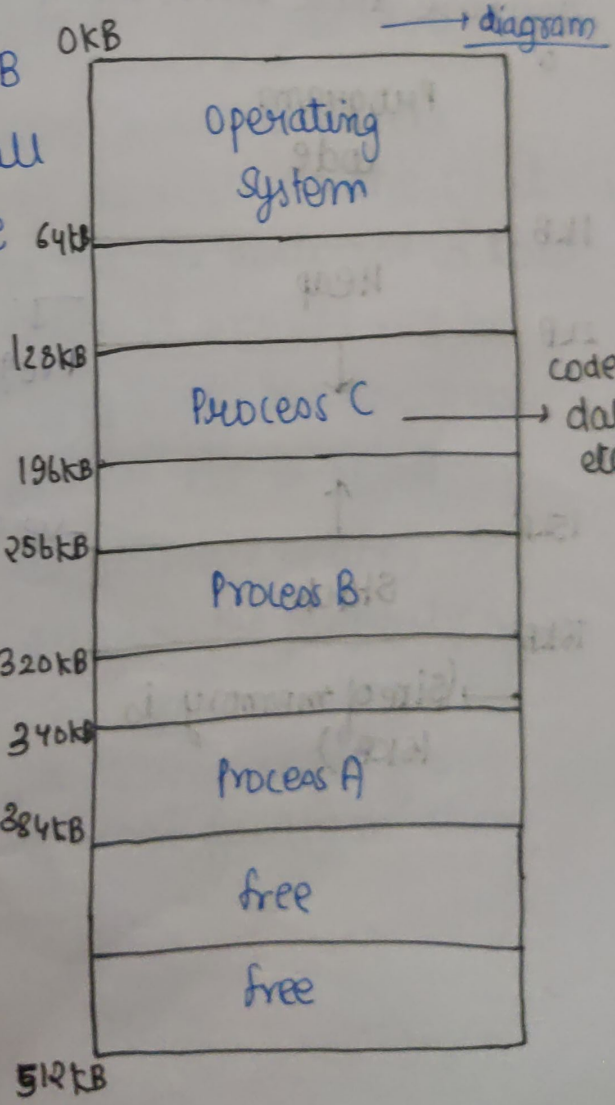
CFS doesn't keep all process in this structure rather only running process are kept there in. If a process goes to sleep (waiting for an I/O to complete or for a network packet to arrive) It is removed from the tree and keep track of elsewhere.

Memory Management

Multiprogramming and Time sharing.

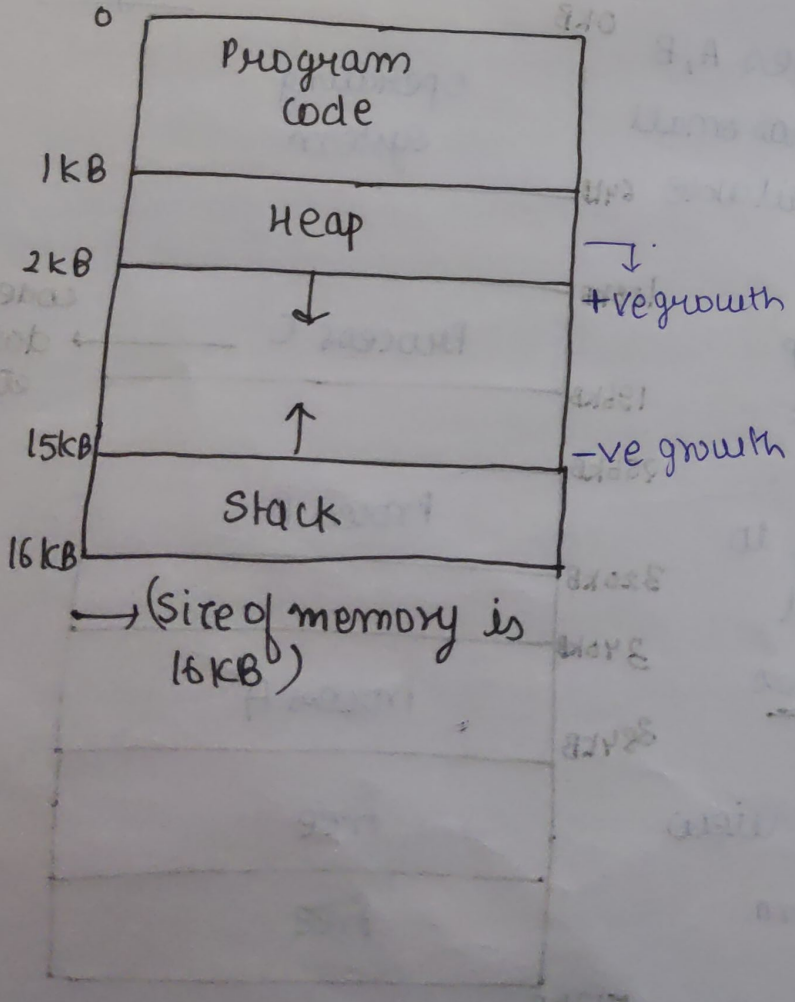
MP in which multiple processes were ready to run at a given time and the OS would switch b/w them one way to implement Time sharing would be to run one process for a short time slice giving it full access to all memory then stop it. Save all of its state to some kind of disk (physical memory) load some other processes states run it for a while and thus implement some kind of root sharing of the machine. limitations of this approach is too slow when memory request increases.

In the diagram, 3 processes A, B and C. And each of them has small parts of 512KB memory available for them.



However, we have to keep all processes demand and requires the operating system to create an easy to use abstraction of physical memory known as address space. Address space is the running programs view of memory in the system.

The address space of a process contains all of the memory state of the running program. For example the code of the program have to live in memory somewhere and thus they are in the address space. The program while it is running uses a stack to keep track of where it is in the function call chain as well as to allocate local variables and pass parameters and return values to and from routines. Finally the heap is used for dynamically allocated user managed memory, such as that you might recieve from a call to malloc() in C or new in cppor (OOP) Java. So let us just assume those three components - code, stack and heap.



The main goal of OS for ease of use:

Transparency: OS virtual memory को ऐसे implement करता है जिससे running program को वो invisible हो।

Protection:

Goal of OS:

30-march-22

1 Transparency: The OS should implement the virtual memory that is invisible to running program. Thus, the program should not be aware of the fact that memory is virtualized. Rather the program behaves as if it has its own private physical memory.

Efficiency: The OS should strive to make the virtualization as efficient as possible, both in terms of time (i.e not making programs run much more slowly) and space.
In implementation time efficient virtualization, the OS will have to rely on hardware support, including hardware features such as TLB's

Protection: The OS should make sure to protect processes from one another as well as the OS itself from processes. When one process performs a load, a store, or an instruction fetch it should not be able to access or affect in any way the memory contents of any other process or the OS itself. (that is, anything outside its address space). Protection thus enables us to deliver the property of isolation among processes. Each process should be running in its own isolated cocoon, safe from the ~~#~~ ravages of (issues) of other faulty or even malicious processes.

Hardware based Translation (Address Translation)

The hardware Transforms each memory access where the desired ~~location~~ information is actually loaded. Thus on each and every memory reference, an address translation is performed by the Hardware to redirect application memory references to their actual locations in memory. Of course the hardware alone can't virtualize memory as it just provides the low level mechanism for doing so efficiently. The OS must get involved takes place, it must thus manage memory, keeping track of which locations are free and which are in use.

Assumptions:

Ⓐ Specifically we will assume for now that the users address space must be placed contigiously in physical memory.

Ⓐ The size of the address space is not too big specifically that it is less than the size of physical memory.

Ⓐ Each address space is exactly the same size.

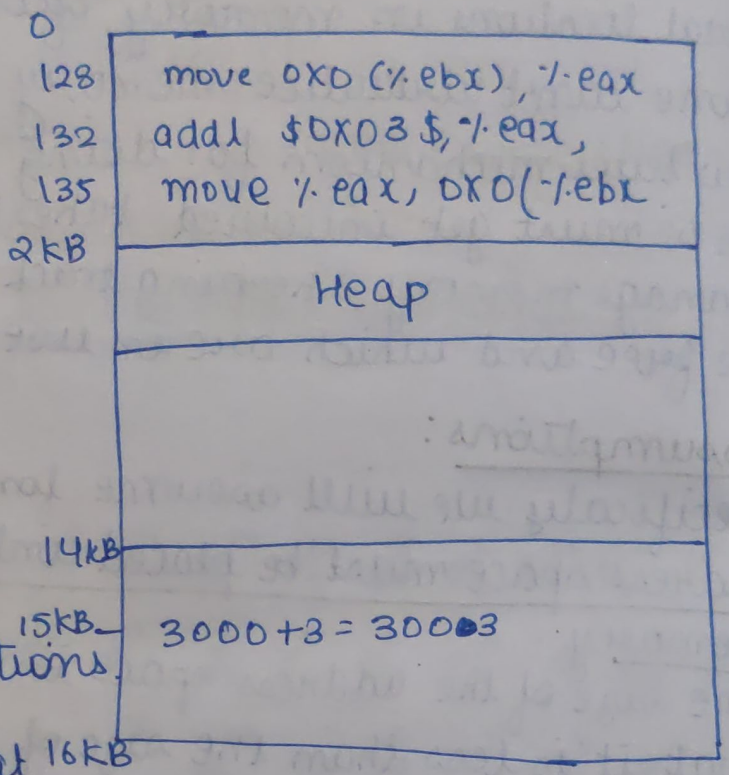
```
128: movl  0x0 (%ebx) %eax;
132: addl  $ 0x03$,%eax;
135: movl  %eax, 0x0 (%ebx)
```

loads a value from memory it increments it by 3 and then stores the value back into memory.

```
void func()
{
    int x=3000;
    x=x+3;
    printf("%d",x)
}
```

and turns into assembly. which look something like this (3 lines written previously)

when these instructions run from the perspective of the process. the following memory access takes place.



Step1: fetch instruction at address 128

Step2: Execute the ins. and load the value from ebx to eax.

Step3: fetch instructions at address 132

Step 4: Execute this instructions.

Step5: fetch the instruction at address 135

Step6: Execute this instruction. and stored the value to address ~~by~~ 15kB

ebx kepp value to the local variable

---

* Interposition [3]
  ↳ transparency.

why Dynamic Relocation?
To virtualize memory os wants to place the process somewhere else in physical memory. not at add.0.

Thus we have problem: How can be relocate this process in memory in a way that is transparent to the process?

Dynamic Relocation

Base and Bound (limit) registers

First time sharing machines is a simple idea referred to as base and bounds also referred as dynamic relocation
Specifically we will need to two hardware registers within each CPU one is called the base register and the other is bound register.
This base and bound pair is going to allow us to place the address space anywhere we did like in physical memory and do so while ensuring that the process can only access its own address space. However when a program starts running the OS decides wherein physical memory, it should be loaded and sets the base register to that value.
Now, when any memory reference is generated by the process it is translated by the processor in the following manner.

$$\underset{\text{address}}{\text{physical memory}} = \underset{\text{address}}{\text{virtual}} + \text{Base}.$$

Suppose a process of address space of size 4KB has been loaded at physical address at 16KB. so what are the result of the address translation for 1KB, for 3000B, for 4400B.

for 1KB → 16KB + 1KB = 17KB

for 3000B → 16KB + 3KB = 19KB

for 4400 → Bound out

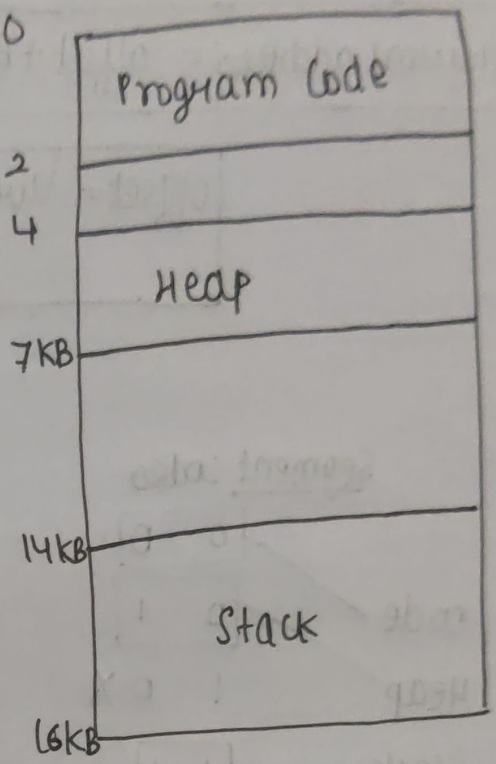
~~Hard~~ Segmentation

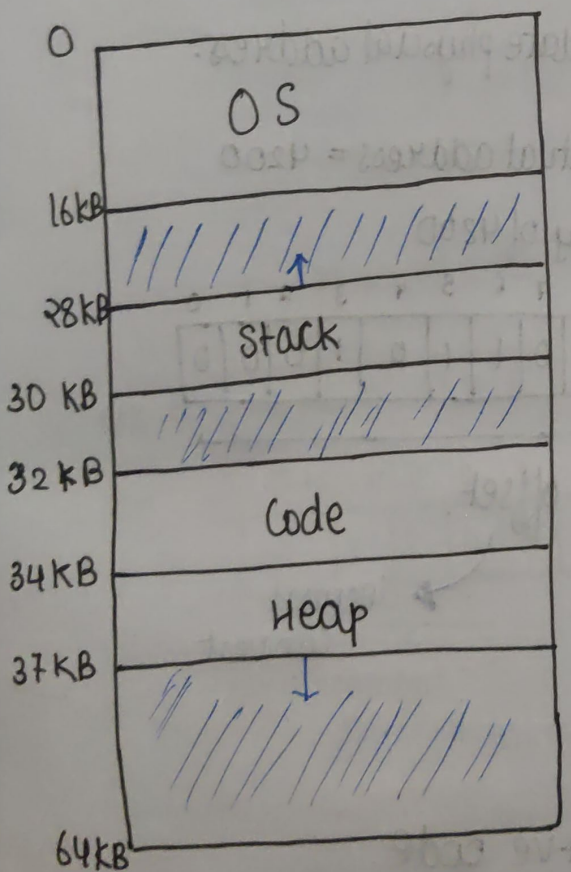
So we have been putting the entire address space of each process in memory with the base and bound registers the os can easily relocate processes to different parts of physical memory.
We have noticed there is a big chunk of free space right in the middle between the stack and the heap.
Although the space between the stack and heap is not being used by the process, it is still taking up physical memory when we relocate the entire address space somewhere in physical memory Thus the simple approach of using a base and bounds register pair to virtualize memory is wasteful.
It also makes it quite hard to run a program when the entire address space doesn't fit into memory.

To solve this problem new idea adopted which is known as Segmentation or generalized base/bound method. The idea is simple Instead of having just one base & bound pair in our memory management unit (mm
A segment is just a contigious portion of the address space of a particular length and in our address space we have three logically different segments (code, stack and heap).

what segmentation allows the OS to do is to place each one of those segments in different parts of physical memory and thus avoid filling physical memory with unused virtual address space.

Segment	Base	Size
Code	32KB	2KB
Heap	34KB	3KB
Stack	28KB	2KB



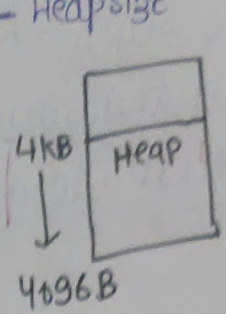


* Defragmentation

Imp page:

Virtual address = 4200

Heap = 34KB



physical address = offset + Base

offset = Virtual address - start heap address

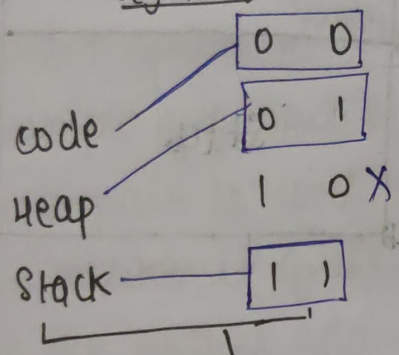
offset = 4200 − 4096

= 104

Physical address = 104 + 34

= 138

Segment codes



Calculate physical address.

virtual address = 4200

Binary of 4200

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	0	1	0	0	0

Segment	Base	Size
Code	32KB	2KB
Heap	34KB	3KB
Stack	28KB	2KB

Segment codes

↑

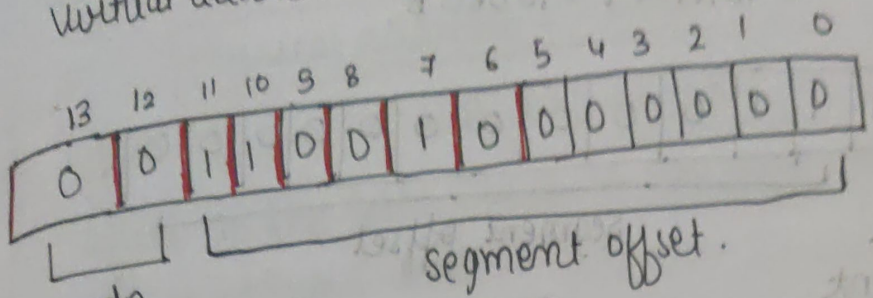
This will tell the base

Segm. offset.

→ decimal convert

for heap and code it is +ve code.

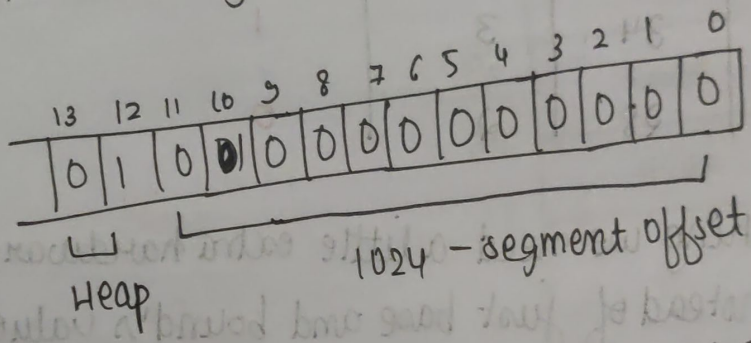
Virtual address = 3200



physical address = 3200 + 32kB

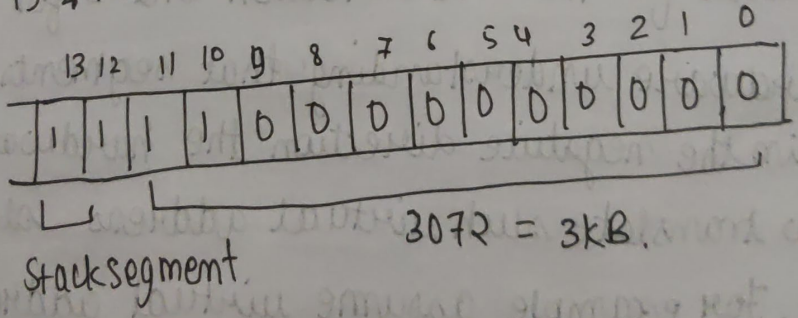
= 3232kB

V.A = 5kB. = 5120Bytes

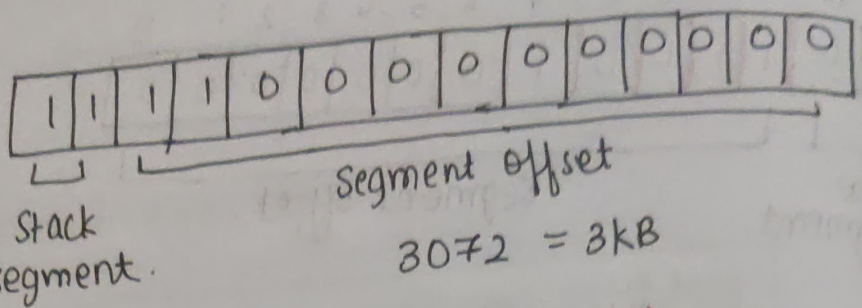


physical add = 1024 + 34 = 35kB

VA = 15 kB = 15,360



virtual address = 15KB
= 15X1024 = 15360 B.



Stack have -ve growth

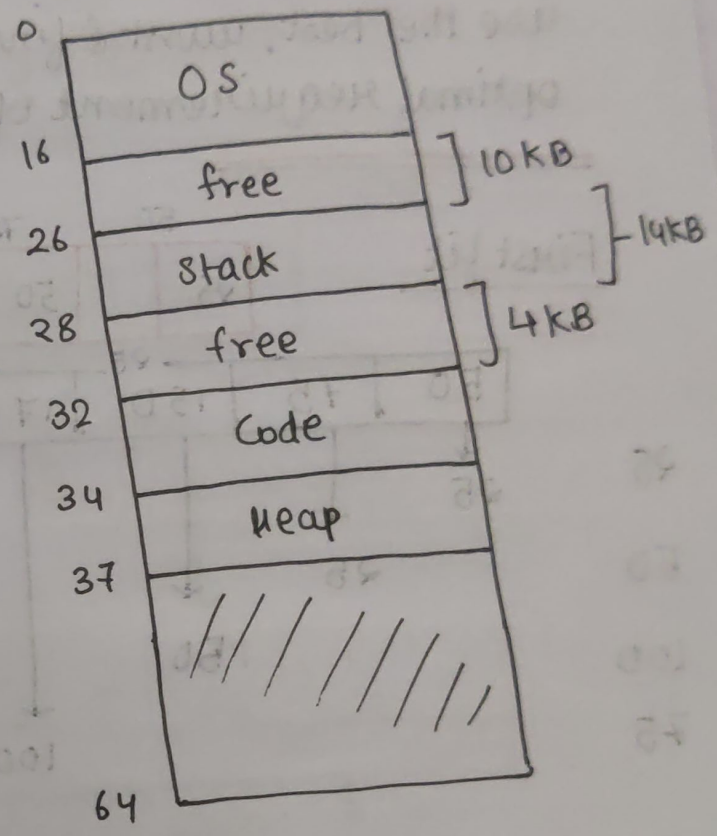
Segment	Base	Size (max. 4KB)	grow
code	32	2	1
Heap	34	3	1
stack	28	2	0

The first thing we need a little extra hardware support. Instead of just base and bound's value the hardware also needs to know which way the segment grows (a bit that is set to 1 when the segment grows for the +ve direction and 0 for -ve) With the hardware understanding that segments can grow in the negative direction the hardware must now translate such virtual address slightly differently. For example assume virtual address to be 15KB. which should map to physical address 27KB so in this segment bit is 1,1 and offset is 3kB. A segment can be maximum size is 4KB

And thus the correct -ve offset is 3KB - 4KB = -1KB.

physical address = 28KB - 1KB = 27KB.

which equals to -1KB we simply add the -ve offset to the base 28KB to arrive at the correct physical address 27KB



Suppose 14KB, 10KB, 250, 500, 600, 360.

200, 50, 300, 100, 40. → chunks segments.

First fit: for 200 → 250 left out 50

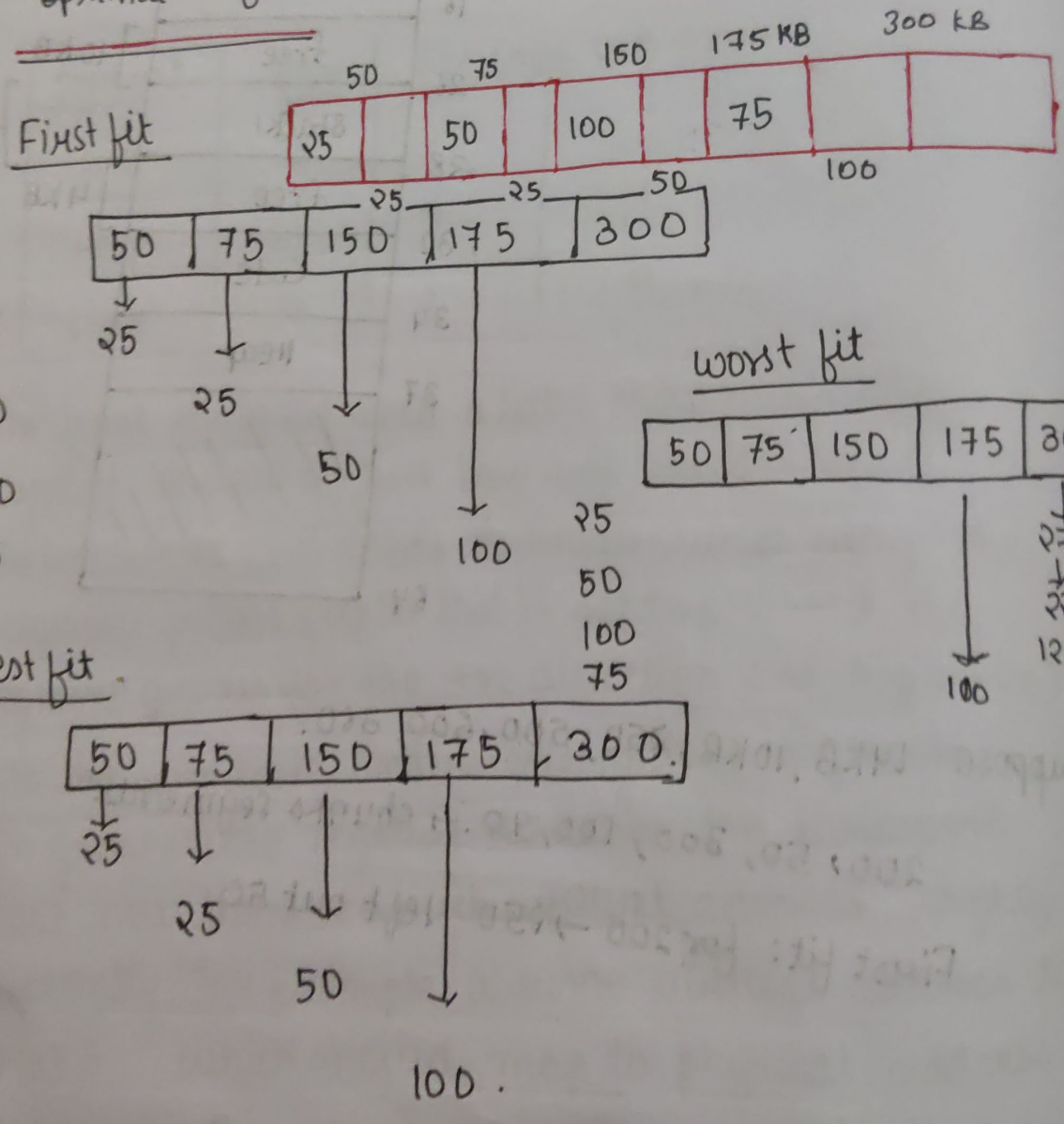
Q) Process request are given as

25kB, 50kB, 100kB, 75kB

and free list of space is

50kB, 75kB, 150kB, 175kB, 300kB.

use the best, worst & first fit algorithm to find the optimal requirement of the process



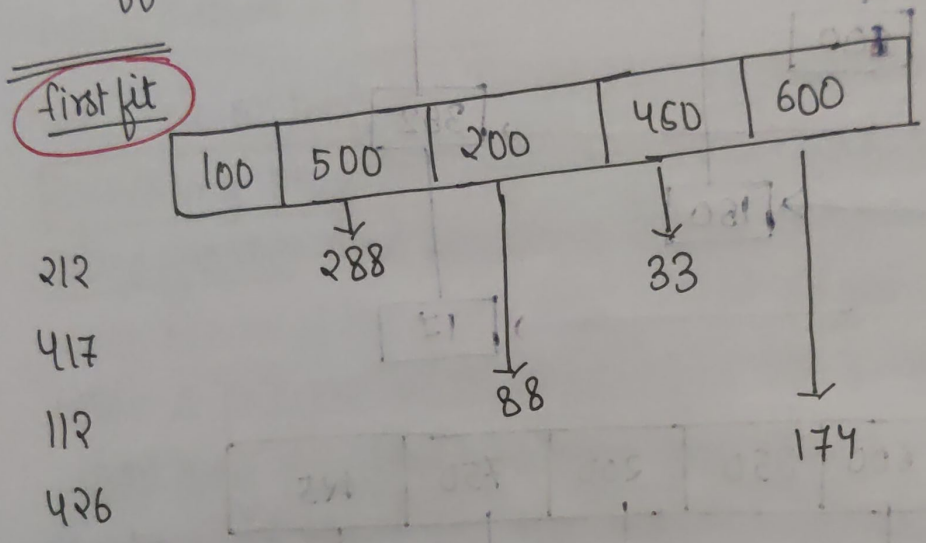
Consider 5 memory partitions of size

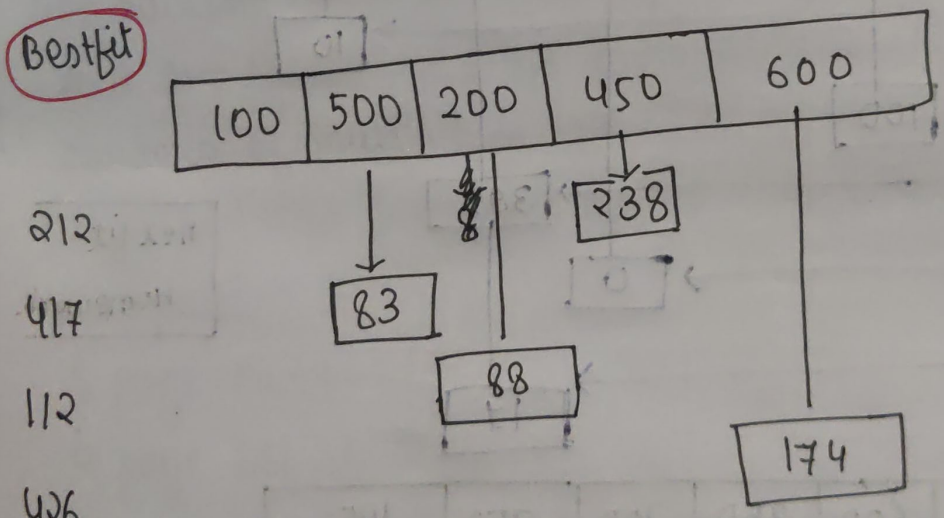
100 KB, 500KB, 200KB, 450KB, 600KB in same order

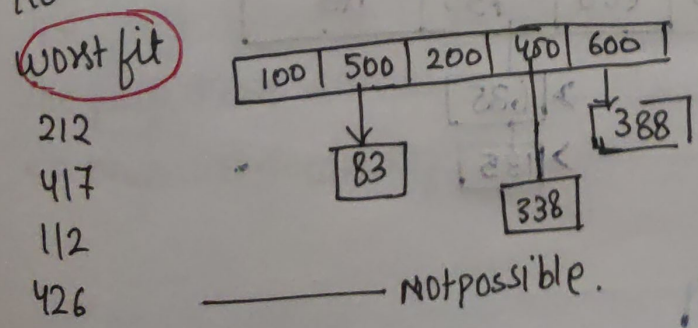
& seq. of req for blocks of size

212, 417, 112 KB, 426 KB in same order come,

then which of the following algo's makes the efficient use of memory.



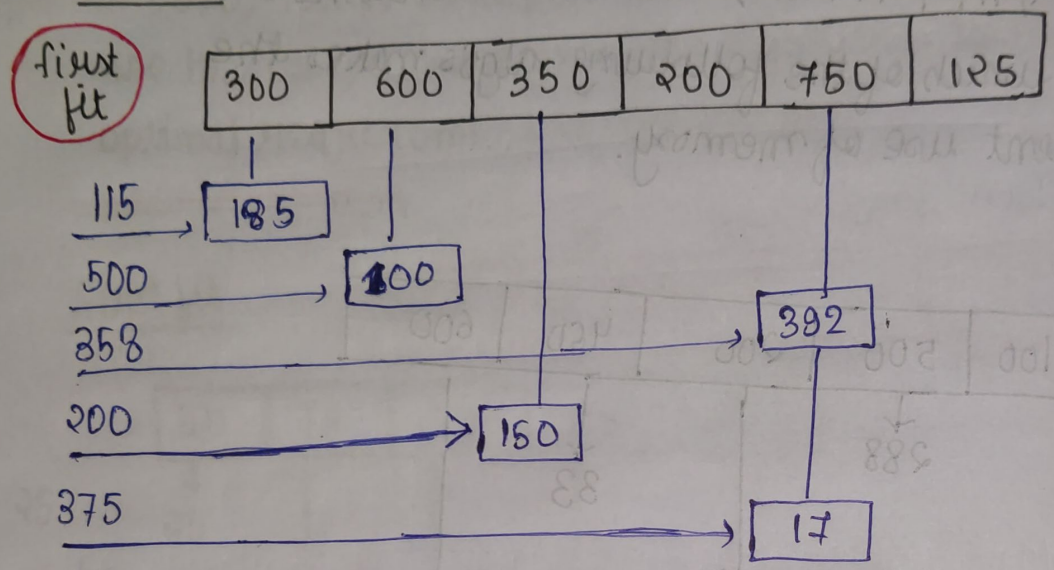


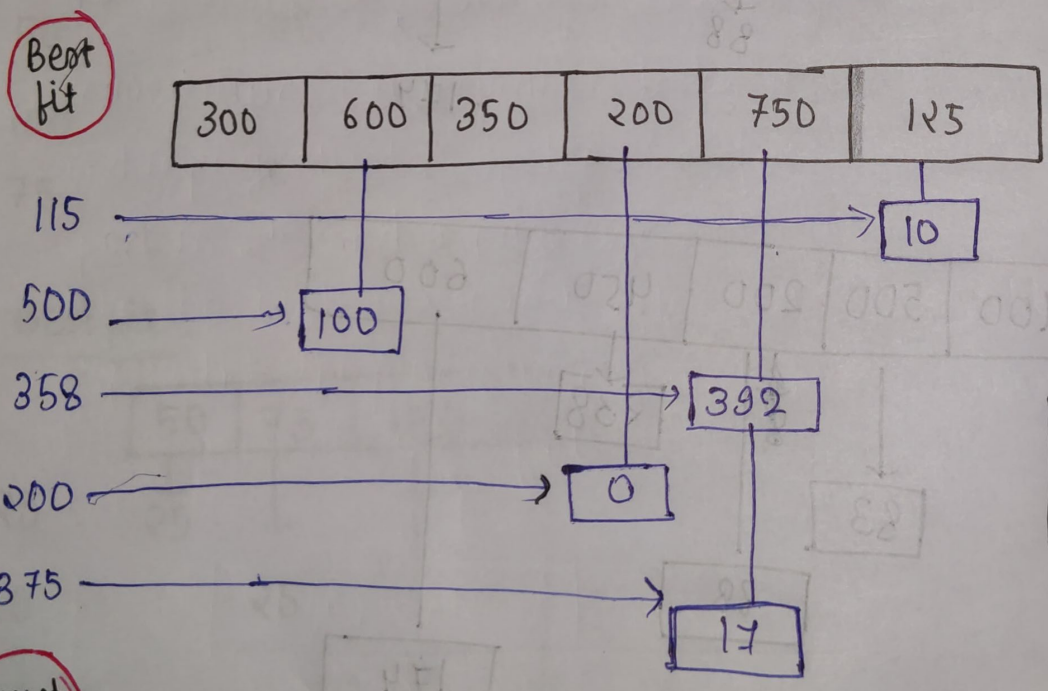


Q) Given six memory partitions

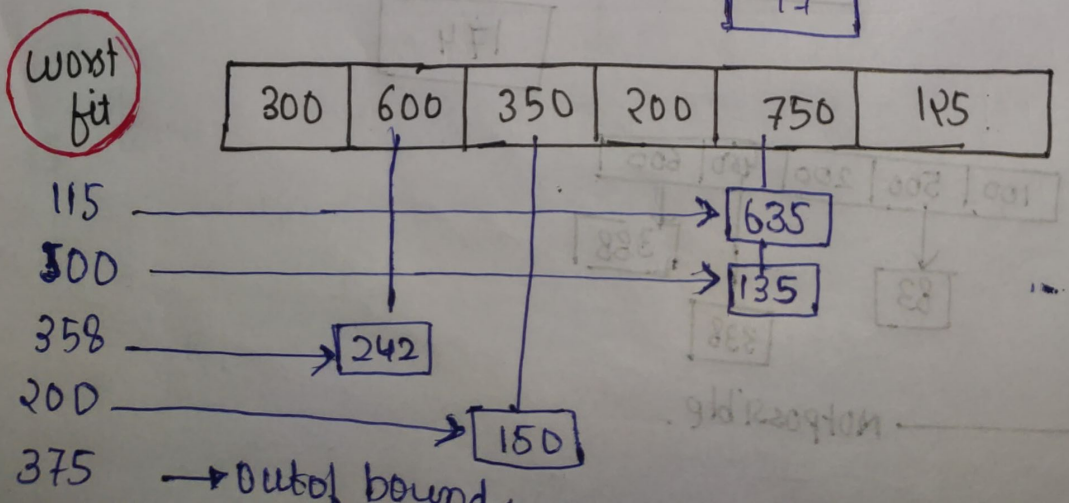
300KB, 600KB, 350 KB, 200 KB, 750 KB, 125 KB in order

How would the first fit, best fit and worst fit algo's place process of size 115 KB, 500KB, 358KB, 200KB, 375 KB





next fit
Homework.



Paging. [Because segmentation has problems in managing free space as memory becomes fragmented]

It is a another memory management scheme that avoids external fragmentation and the need for compaction whereas segmentation does not. It also solves the considerable problem of fitting memory chunks of varying size onto the backing store

Basic method: [Paging: → address space into fixed size units we call a page]

The Basic method for implementing paging involves breaking physical memory into fixed size blocks called frames and breaking logical memory (address space) into blocks of the same size called pages.

When a process is to be executed its pages are loaded into any available frames from there source. (from store)

The backing store is divided into fixed size blocks that are the same size as the memory frames or clusters of multiple frame.

Every address generated by the CPU is divided into two parts:

- page number (p)
- page Offset (d)

The page number is used as an index into a page table. The page table contains the base address of each page in physical memory.

This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

The page size is defined by the hardware. The size of the page is a ~~pow~~ power of 2 varying b/w 512 Bytes and 16 MB per page depending on the computer architecture.

page size = $2^{\square}$ → 512 bytes — 16 MB

(Q) What is the difference b/w 32 bit and 64 bit architecture?

The selection of power of 2 as a page size makes the translation of a logical address into a page number and a page offset. If the size of the logical address space is 2^m and a page size is 2^n, then the high order m-n bits of a logical addresses represents the page number and the n lower order bits represents the offset.

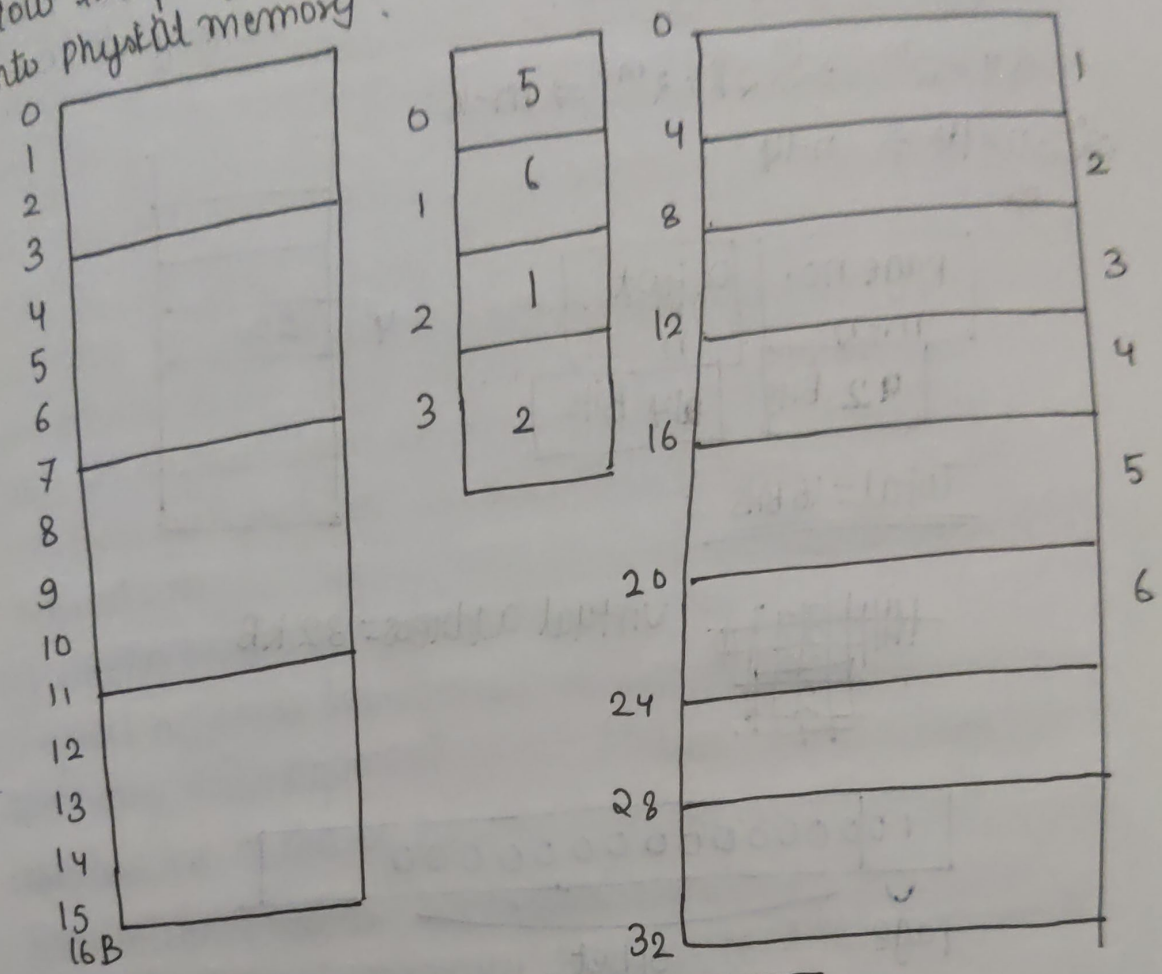
Page number	offset
m-n	n

Logical address = 16 KB
$= 2^4 \times 2^{10}$ B
$= 2^{14}$ B → m

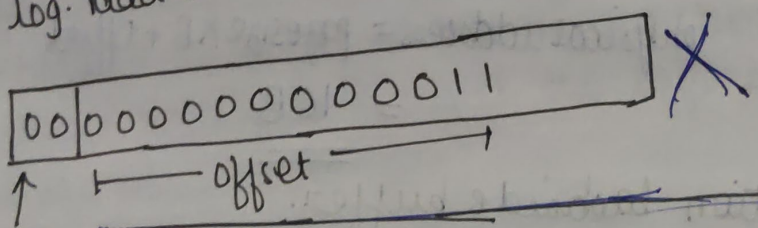
page size = 4 KB
$= 2^2 \times 2^{10}$ B
$= 2^{12}$ B → n

m - n = 2 bits

8 Consider the memory of 16KB size 4KB each, n=2, m=4 using a page size of 4 Bytes and physical memory of 32 Bytes. So How the programmers view of memory can be mapped into physical memory.



for log. add=3



physical address = page size × Base add + offset

$4 \times 5 + 3 = 23$

Virtual address = 8 bit

Page size = 16 Bytes

↳ $2^4 = 16$ $\boxed{4=n}$

Bits = 4

Total Pages = 16

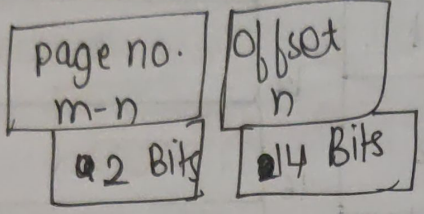
32X1024

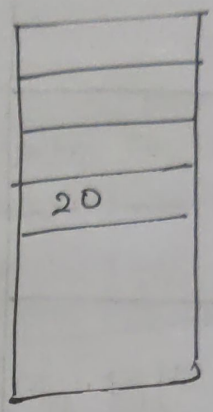
Q) Logical address size = 64 kB
Page size = 16 kB
physical memory size = 128.

virtual address = 32 kB.
find out the physical memory address

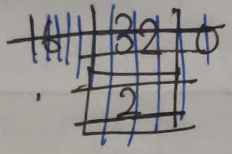
$64 = 2^m \Rightarrow 2^6 \times 2^{10} = 2^m \Rightarrow m = 16$

$2^n \times 2^{10} = 16 \Rightarrow n = 14$

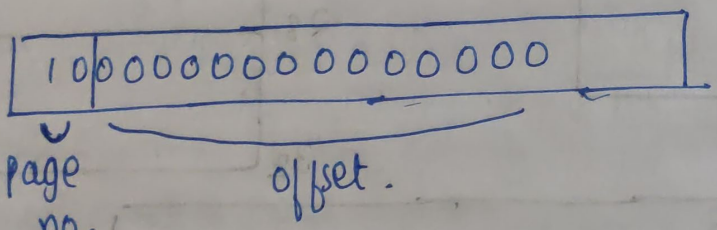




Total = 16 Bits



virtual address = 32 kB



physical address = page size × B + Offset
= 16 B

Interview point of view

TLB: Translation lookaside buffer.
- store extra info
- and with help of cache it boost up the speed of the process.

- TLB-hit
- TLB-Miss

[Best eg: Browser]

Using paging as a core mechanism to support virtual memory can lead to high performance overheads By chopping the addressspace into small fix size units (pages), paging requires a large amount of mapping information. Because that mapping information is generally stored in physical memory paginglogically requires an extra memory lookup for each virtual address generated by the program. Going to memory for translation information before every instruction fetch or explicit load or store is prohibitly slow. And thus our problem is how to speed up memory translation.

By using hardware translation lookaside buffer (TLB) To speed address translation a TLB is part of the memory management unit and it is simply a hardware cache of popular virtual to physical address translation. (Address translation cache). Upon each virtual memory reference the hardware first checks the TLB to see if the desired translation is held there in, if so the translation is performed without having to consult the page table they may get the physical address.

TLB Algorithm

↳ Assumption:

- Simple linear page table:
- Hardware managed TLB

The following steps are as follows:

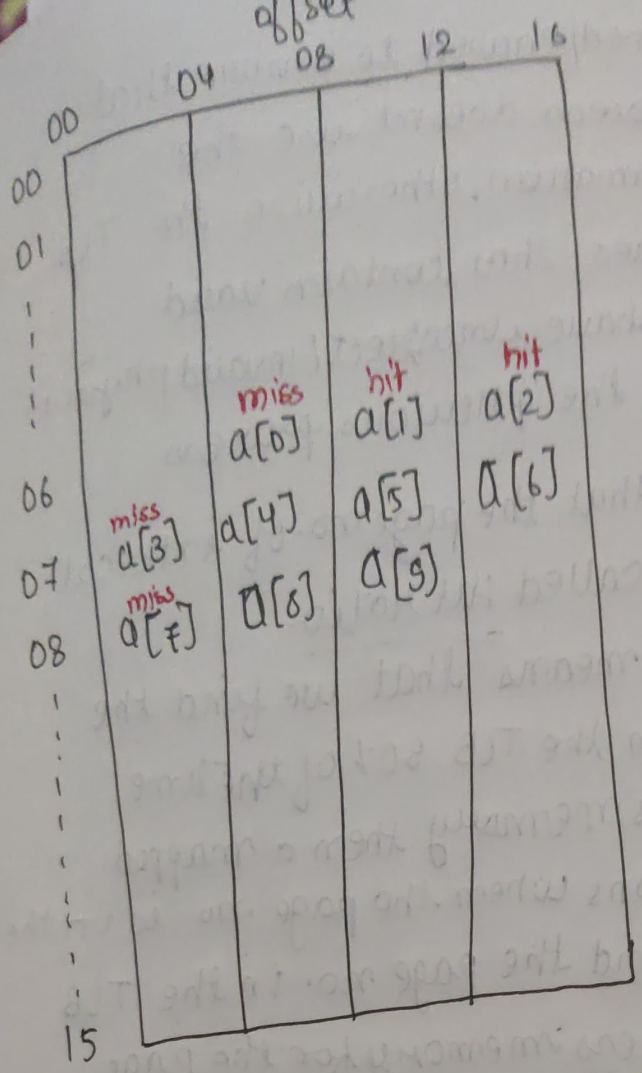
(1) Extract the virtual page number from the virtual address and check if the TLB holds the translation for this VPN (virtual page no.)

(2) If it does a TLB hit is success then we can now extract the page frame number from the relevant TLB entery, concatenate that onto the offset from the original virtual address, and from the desired physical address, and access memory, assuming protection checks do not fail

(3) If the CPU doesn't find the translation in the TLB then the hardware access the page table to find the translation and assuming that the virtual memory reference generated by the process is valid and accessible, updates the TLB with the translation.

Example: Accessing an Array.

We have an array of 10 four byte integers in memory starting at virtual address 100.
Assume further that we have a small 8-bit virtual address space, with 16 byte pages; Thus



Some TLB stores address space identifiers in each TLB entry. And ASID uniquely identifies each process and is used to provide address space protection for that process. When the TLB attempts to resolve virtual page numbers it ensures that the ASID for the currently running process matches the ASID associated with the virtual page. If the ASID do not match the attempt is treated as TLB miss. In addition to providing address space protection and an ASID allows the TLB to contain entries for several different process simultaneously. If the TLB does not support separate ASID's then every time a new page table is selected

, the TLB must be flushed/erased to ensure that the next executing process doesn't use the wrong Translation information, otherwise the TLB could include old entries that contain valid virtual addresses but have incorrect/invalid physical addresses left over from the previous process.

The percentage of times that the page no. of interest is found in the TLB is called Hit Ratio.

For eg: A 80% Hit ratio means that we find the desired page number in the TLB 80% of the time.

If it takes 100ns to access memory then a mapped memory access takes 100ns when the page no. is in the TLB. If we failed to find the page no. in the TLB then, we must first access memory for the page table and frame number, and ~~nanoseconds~~. and then access the desired byte in ~~memory~~ (100 nanoseconds) for a ~~total~~ total of 200 ns.

To find the effective memory access time, we must weigh each case by its probability

V.Imp

Effective access time = P × hit memory time + (1-P) × miss memory time

80% → Hit Ratio

$P = \frac{80}{100} = \frac{80}{100} \times 100 + \frac{20}{100} \times 200$

$= 120$ ns

Hit Ratio 99%.

$P = \frac{99}{100} \times 100 + \frac{1}{100} \times 200$

$P = 101$