M6 : CS Fundamentals

Q - What is an operating system?

A - Interface b/w hardware and software

Examples - windows, linux, unix, mac OS

Why learn operating systems?

Understand how OS executes code
Concepts of OS are asked in interview
Learn bash/shell scripting

Course outline:

Functions of OS
CPU virtualization
Memory virtualization
Concurrency
Bugs related to concurrency

Resource Management

What happens when we "run" a code?

Compiler: Translates code to binary file/machine level code
Fetch/Decode/Execute cycle starts
a, Fetch: Fetch next instruction from the machine level code
b. Decode: Interpret that line
c. Execute: Execute the instruction

In an interpreter based language, the two steps - compilation and fetch/decode/execution are intertwined - the initial code, is read line by line by the interpreter, and is converted into machine level code and executed.

Resources required to run our code

RAM (Memory - where the program is stored)

CPU (For fetching, decoding and execution)

Drives/HDD

How OS helps in running a program

Since these resources are to be shared among different processes, OS is required to allocate/manage these resources to facilitate the execution of code.

System calls: Commands that OS provides to the requesting process for getting access to the resources.

Example: mmap()-> Finds a particular memory location and gives it to the process.

CPU Virtualization

How many programs can CPU run at a time?

In a single core CPU, the CPU can execute only one program at a particular moment. But the OS can quickly switch between multiple programs: it can preempt an executing program and make another program execute. This gives an appearance to the user that all the programs are running simultaneously, but under the hood, it's the OS which is switching quickly between programs.

In the above diagram, 3 tasks - A,B and C are executed - without switching (top) and with switching (bottom)

Memory Virtualization

Execution of a program (machine level code) involves FDE - Fetch, Decode and Execution.

While fetching, the code is brought from the memory to CPU, for decoding and subsequent execution. If the CPU tries to fetch the code from hard disk, fetching will be very slow. Therefore, the OS first brings the code to RAM before it starts executing the program.

While bringing the code into RAM, can it happen that the code's memory is more than the RAM's capacity?

Yes. Imagine running a high graphic game (300 GB). RAM (typically 4-16 GB) is very less than the game's memory. In this case, the OS tries to manage the 16 GB RAM by moving the segment of code that is going to be executed, and removes the code that has been executed from the RAM. This is the idea behind virtual memorization - the game's process is able to access all the 300 GB of memory, whenever required - and the OS facilitates this by clever swapping of memory.

This gives the illusion to the process that a lot of RAM memory is available than there actually is.

Introduction to Processes

Q: What is a process?
A: A program in execution

Context switch:

When there are multiple processes available to run

1. OS selects a process

2. Converts it into a process

3. Gives it to CPU

4. After some time, the OS swaps out this process from CPU

5. Swaps in another process/program to CPU

The process of OS consistently swapping processes in and out is called context switching.

The concept of one actual CPU's time being shared by multiple processes is called "time sharing". This gives the illusion that each process has their own dedicated CPU.

Advantage of context switch: Interactivity

Disadvantage of context switch: CPU is unable to execute when context switch is taking place

In the above diagram, 3 tasks - A,B and C are executed - without switching (top) and with switching (bottom). The gray bands represent time wasted due to context switching.

Process Control Block

What all comes under PCB:

Memory Layout, Registers and Metadata

Memory Layout

Stack grows downwards, Heap grows upwards.If stack and heap meet each other, then we get the error -> MLE(Memory Limit Exceeded)/ Segmentation fault

2. Registers
a. Program Counter
Stores the current statement number that we are executing. When a program is swapped out and then swapped in later, PC register helps in identifying where should the execution resume from.

b. Stack Pointer
Lets say main() calls f1(), and f1() calls f2(). Each of the function calls: main, f1 and f2 will have their own stack pointers. Stack pointers store the "current activation record", which means the current line being executed in that particular function. If we just look at PC register, it will just show where were we in f2. To know from where was it called, we need to look up the SP register. SP register stores SP pointers which are available for every function.

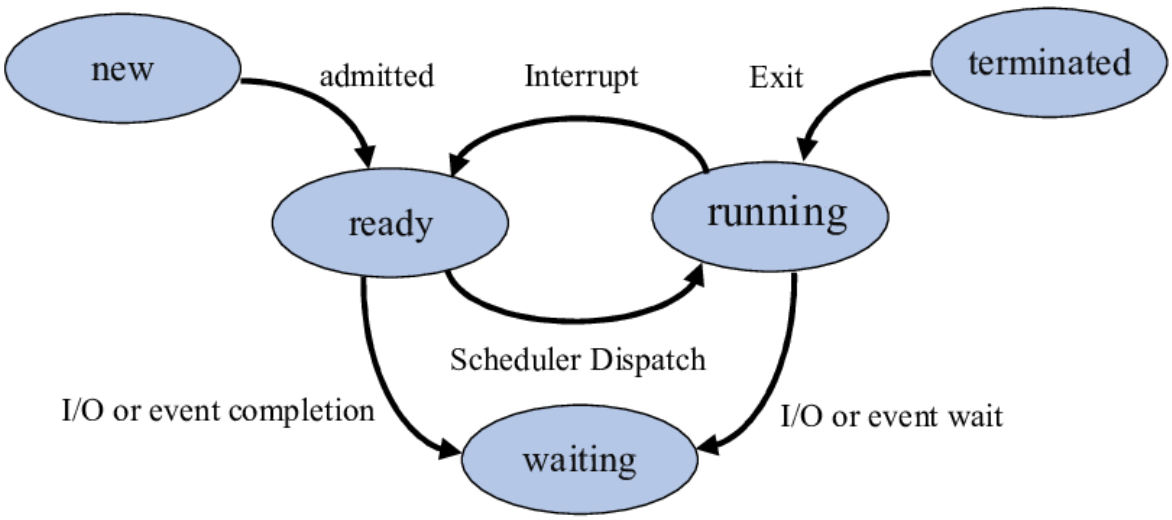
Lets say that main calls f1 at line #4, then the stack pointer of main() will store line #4 in itself and that stack pointer will be stored in the stack register.

c. Frame pointer
Frame pointer simply stores the beginning of the current function. Why is it needed? Say f1() called f2(). f2() is now at top of function stack and it has got memory allocated. When f2() is over, and f1() resumes, we need to delete the memory associated with f2() and we can do that by erasing every memory allocated above the frame pointer of f1(). Every function has its own frame pointer.

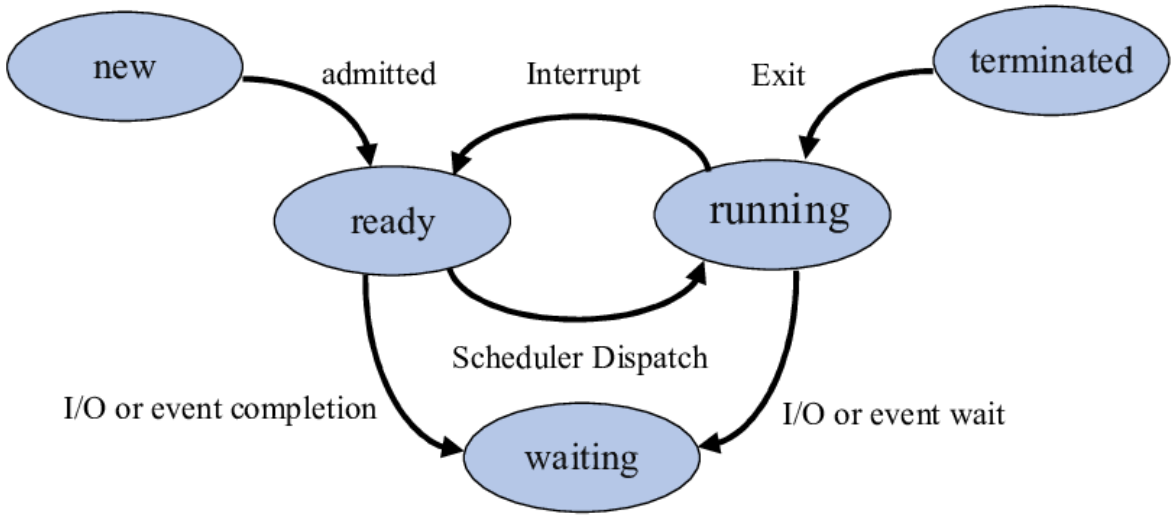
3. Metadata
Stores:
a. ID of current process
b. Priority
c. I/O status
d. Status of the process
e. Limits (of memory, time etc)
f. ID of parent process

Process Lifecycle

States of a process



New State (when OS identifies the process as something to be run)
Ready State (when its admitted by OS to run it)
Running State (when OS is executing it)
Terminated State (process completed or made to stop)



Q. What is the difference b/w new state and ready state 2?

A. In b/w new and ready state a few things happen:

Address space [load the code into RAM]
Allocate stack / heap space

After these 2 are done, a process moves from new to ready state.

Eager loading vs lazy loading

Eager loading: Memory for the process is eagerly loaded. It is taken and loaded into the RAM.

Lazy loading: You just reserve memory and say that this memory needs to be allocated to this process sometime later. This is just like reserving a table in restaurant for later.

Process Queues

Ready Queue: Contains admitted processes that are in line to run

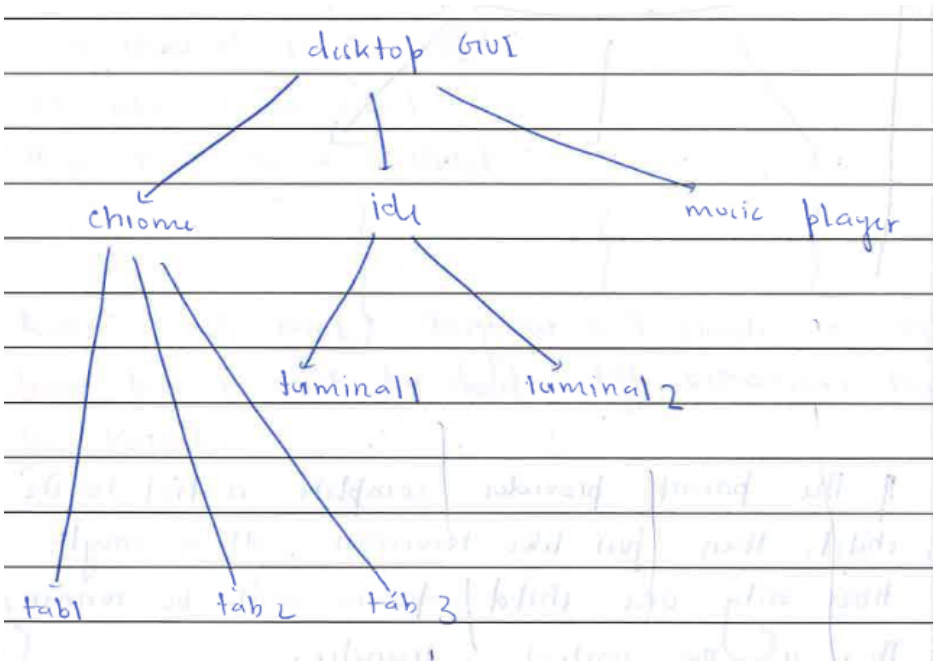
Job queue: Contains process in new state that are to be transformed into ready state. (processes waiting in line to be allocated space/memory)

Device Queue: Device's (I/O) in queue: Each of the i/o devices have their own device queue. In the waiting state of each process, they have some input or output they need/give from/to the device and that input/output is stored in device queue.

The PCB of each process can be found in either ready queue or job queue.

Creation of Processes

Existing processes spawns new processes. So all the processes can be imagined to be a part of huge tree.

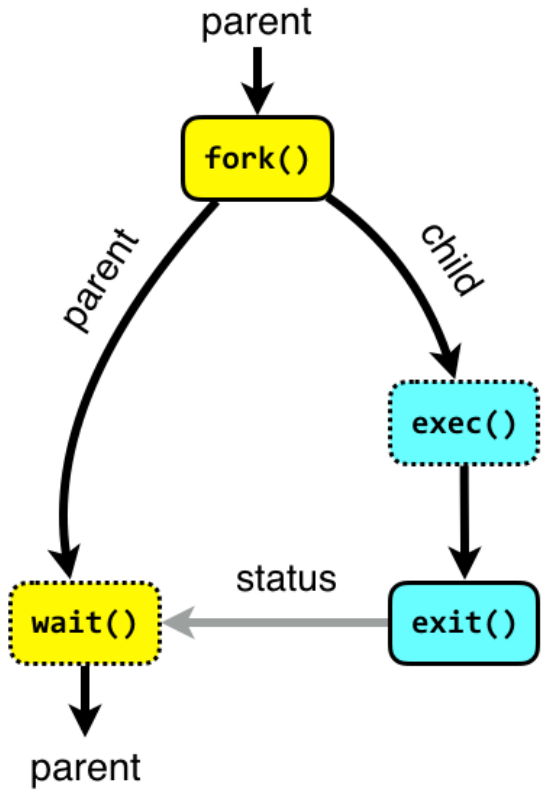


The scheduler process is the root of the tree, its pid is 0. It is the first process to run on the system, and it spawns the init process with pid 1. Each process in the system is spawned by init, or spawned by a process which in turn was spawned by init, and so on.

fork() system calls allows the parent process to spawn child process such that the parent and child can run independently (concurrently).

In some use cases, we want the parent to resume only after the child has completed. In those cases, we use the wait() system call within the parent.

When a process P uses the fork() system call, it spawns a child process which is a clone of the process P, i.e., it has the exact same code, program counter etc. To make the child process execute a different program, exec() system call is invoked from within the child process, which allows the child to run the desired program instead of just being the copy of the parent process.



A typical scenario, where the parent creates two processes using fork, and waits for the child to execute a program (using exec), and then the parent waits for the child to exit.

Termination of Processes

Ways a program gets terminated:

Voluntary exit: Process is completed & the process itself calls the exits system command: exit()
Involuntary exit:

a) System/OS takes decision to kill process because it's trying to do

something which it is not allowed to do (like unauthorized access)

b) Parent process kills the child process(for instance, if child process

exceeds the time limit)

What happens after termination:

Resources are freed
Return exit code() to parent process: If exit code is 0, it indicates successful completion of child, else it indicates erroneous exit.

Complications

Orhan processes:

Suppose the parent process P has spawned a child process C. Assume that P is not waiting for C's completion, and terminates before C does. In such situations C is called an orphan process. An orphan process is inherited by the init process(pid=1). The init process either kills the orphan process or lets it run and waits for completion.

2. Zombie processes:

Suppose the parent process P has spawned a child process C. Suppose P is not waiting for C and C terminates. Since P is not waiting for C, the exit code of C won't be read by P. The child will not be completely terminated because exit code() could not be returned (Parent is not waiting to accept it.). Such child processes which have completed but cannot be destroyed because their status has not been read by the parent are called zombie processes.

Introduction to Process Scheduling: FIFO

We start with these 4 assumptions to simplify the process scheduling algorithms:

All processes take finite amount of time
Once started (running), a process cannot be paused. [No interrupt]
Runtime of each process is known beforehand.
No process will be using and waiting for I/O (input/output).

Turnaround time: The time gap between the arrival of a process and its finishing time.

Avg turnaround time (T_avg): The average of turnaround times of all the processes.

First In/First Out (FIFO) scheduling algorithm:

Say at t=0 ms, two processes arrive with runtime 10 ms.

P1 runs from 0 to 10 ms.

P2 runs from 10 ms to 20 ms.

Turn around times for

P1: 10-0 = 10 ms

P2: 20-0 = 20 ms

Avg turnaround time = (10 ms + 20 ms)/ 2 = 15 ms

Avg turnaround time would have been the same had we taken P2 first to run followed by P1.

In case the processes have different run times, FIFO does NOT minimize the average turnaround time.

It is the reason why FIFO is not a good scheduling algorithm.

Shortest Job First Algorithm

Idea: Scheduling processes with smaller run time earlier than processes with large run time reduces the average turn around time.

Algorithm: If there are multiple processes in ready queue at a given time, then the process with the shortest run time is scheduled first.

Example:

Process	Arrival time	Running time
P1	0	2
P2	1	100
P3	2	1
P4	3	2

Average turnaround time with shortest job first algorithm:

From t = 0 to t = 2, P1 will run.

At t = 2, P2 and P3 are in queue.

We run the shorter job, i.e. P3

From t = 0 to t = 3, P3 will run.

At t = 3, P2 and P4 are in queue.

t = 2 to t = 3, P3 will run.

At t = 3, P2 and P4 are in queue

t = 3 to t = 5, P4 will run.

At t = 5, P2 is the only process in queue.

t = 5 to t =105 , P2 will run.

Avg turnaround time = ((2-0) + (3-2) + (5-3) + (105-1))/4 = 109/4 = 27.25

Average turnaround time with first come first serve algorithm:

t = 0 to t = 2, P1 will run

t = 2 to t =102, P2 will run

t = 102 to t = 103, P3 will run

t = 103 to t =105. P4 will run

Avg turnaround time = ((2-0) + (102-1) + (103-2) + (105-3))/4 = 309/4 = 77.25

Shortcoming of shortest job first algorithm:

If a process with large run time is already running in the CPU, then the smaller processes will have to wait till the large process finishes. This increases the average turn around time.

Example:

Process	Arrival time	Running time
P1	0	16
P2	1	1
P3	2	1

P1 arrives at t=0 and it is the shortest job of all so it will run initially till t=15 and P2 and P3 will have to wait till t =15.

Solution for the above : Pre-emption

A running job may be interrupted in the middle and some other job is given the CPU.

Shortest Time to Completion Algorithm

Idea: If a running process is going to take more time to complete (has more remaining time) than a process in the queue, then the running process is preempted, and the other process with shorter time to completion is scheduled first.

Example:

Process	Arrival time	Running time
P1	0	15
P2	1	1
P3	2	1
P4	3	2

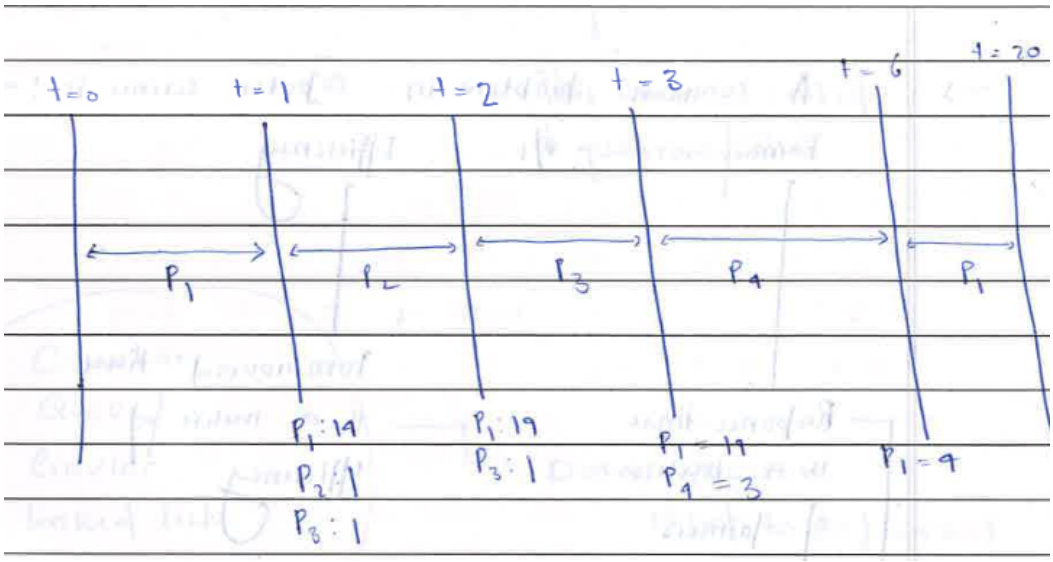
At t = 0s, there is only P1 in the queue, so P1 is scheduled.

At t = 1s, P2 is in the queue, but P1 is already running, and will run for remaining 14 seconds.

If we relax the constraint of no interruption, we can interrupt P1, and schedule process P2. This will drastically reduce the turn around time of P2 and the remaining process, P3 and P4.

The shortest time to completion algorithm interrupts a process in favor of another process whose *remaining time* is less as compared to the currently running process.

The schedule using the shortest remaining time first will look the following for the above scenario:



Issues with shortest time to completion algorithm: Starvation

Say P1 has run-time 1000 and it is followed by a continuous stream of small runtime processes. CPU won't give run time to P1, since all small time processes are running. This causes starvation of P1. If small runtime processes keep on coming indefinitely, P1 will be starved indefinitely. There is no upper bound on the response time.

Categories of scheduling algorithms

Pre-emptive: Allowed to remove a running process in middle of its execution
Non-preemptive: Algorithms in which a running process exits only after its completion, there is no pre-emption.

From now on, we also remove the assumption that processes are using only CPU - there may be I/O (input output) requests by the processes. Since I/O typically takes multiple CPU cycles to complete, and during that time the process would not utilise CPU but just wait for the I/O to complete, we move out the process which are waiting on I/O.

Note: Removal of process on account of I/O waiting is NOT considered preemption. A process will be moved out of CPU in case of wait for I/O irrespective of whether the algorithm is preemptive or not.

I/O burst vs CPU burst

If a process is working in CPU it is called a CPU burst. If a process is waiting in I/O, it is called an I/O burst.

A process is a sequence of CPU bursts & I/O bursts. And each CPU burst can be considered as one sub process. No matter whether the algorithm is preemptive or non preemptive, we will be scheduling the CPU bursts and not the whole process.

Interactive Processes:

Processes that do a lot of I/O.

Very frequent shift between CPU bursts and I/O bursts. Their CPU bursts (and hence subprocesses) have low runtime.

Round Robin Algorithm

Idea behind the algorithm:

A common problem in computer science is:

Fairness vs efficiency

Fairness - Response time is a metric of fairness

Efficiency - Turn around time is a metric of efficiency

Response time: The average time gap between the arrival of the process and the 1st instance when it is given CPU time.

There is a trade off between response time and turn around time. We have to balance both of them. Absence of efficiency and being completely fair would mean rotating the CPU for each process. Absence of fairness and being completely efficient would lead to starvation.

Round Robin Algorithm focuses on fairness but also tries to make sure that its somewhat efficient.

Algorithm: Every process gets 't' time of CPU and then next process gets CPU and this keeps happening in circle. This time is called t_quantum/slice.

Implementation: Circular queue/Circular linked list is used to implement round robin algorithm.

t_quantum - If it is very high (say 10 s), then it ceases to be round robin. It becomes more of a first come first serve because T_quantum is very high.

If it is very low, then much time will be spent in context switching only. It should be something of a middle value and experimentally that time has been found to be 10-100 ms.

Performance analysis:

If there are n processes, then in atleast t_quantum x n time, every process will have firstly interacted with CPU. Response time is quite good.

For turnaround time, the round-robin algorithm is not that good. The context switches and the fact that a process will probably be finished after t_quantum x n time makes it less efficient.We will try to find a more efficient algo and this time also give up our last assumption (i.e. we would not know the run time beforehand)

Modes of Process Execution

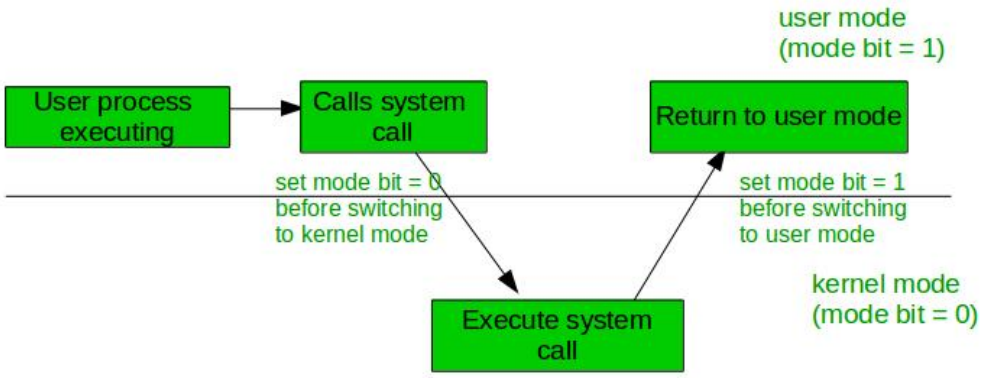
Since resources are to be shared among several processes, access to resources by user processes should be restricted.
On the other hand, OS should be given an unrestricted access to resources so that it can distribute it fairly and efficiently.

This distinction of when to give restricted access and when to grant unrestricted access is made through process execution modes.

There are two modes of process execution

Kernel mode - Infinite access to resources (Privileged modes)
User mode - Restrictions

The same process runs in user mode while carrying out its normal tasks and is elevated to kernel mode when it makes a system call.



Interrupt

The elevation from user to kernel mode happens via a software interrupt. Every software interrupt has an Id with it. This ID points to a specific code that will run under that particular software interrupt.

Interrupt vector: It is a table that specifies which code should be run for a given software interrupt ID.

Kernel stack: When we enter kernel mode, a separate stack is used to store its relevant information called kernel stack. The user processes cannot access and thus cannot corrupt the kernel stack.

Even though the software interrupt allows us to get into privileged mode, a malicious code can't damage the OS's crucial data structures in privileged mode or gain unrestricted access to resources.
This is because of the fact that in the privileged mode, only pre-defined code(through the interrupt vector table) is executed in the elevated mode and the malicious software does not have the freedom to run its own code in privileged mode.

Traps/Exceptions/Faults

When a process - in user mode - does something illegal, like trying to access a resource that is restricted (for example, tries to read a memory address outside of its address space), then it leads to traps/exceptions/faults - special type of software interrupts. What code the OS must execute to handle that particular illegal operation is determined by a trap table.

The trap table is nothing but a table containing the code for each trap id.

Handling hardware inputs (inputs from peripheral devices)

Two possible methods of handling hardware(e.g. keyboard/mouse) inputs:

Polling : Keep checking keyboard/mouse from time to time. If there is any new input.

Problem: High CPU wastage if number of input devices are high. The CPU will have to continuously keep checking the input even if there is no user input.

2. Hardware interrupt : Keyboard/mouse sends signal to CPU when some input is gathered. The CPU, depending on the priority of the current process it is running, preempts the process and processes the input.

Some interrupts are maskable, while other interrupts are non-maskable (can be ignored by the CPU). For each interrupt type, it is pre-defined whether it is maskable or non-maskable.

Context Switch

Say process P1 is running in CPU right now. And say P2, P3, P4, P5, ... are in ready queue.

If the scheduling is preemptive, it is possible that P1 could be removed and P2 starts running.

PCB(Process control block) of P1 removed from CPU.

PCB of P2 posted into CPU.

Also registers, cache etc are refreshed.

When P1 is running, then that means OS (which itself is a process) is not running. So how will OS remove P1?

Using co-operative way

Wait for P1 to make a system call/software interrupt. OS takes control in interrupt mode and then scheduler is run by OS to check if some other process has to be moved or not.

2. Using non-cooperative way

H/w interrupt is used to have OS take back control. OS starts a timer of some t time at the beginning of execution of P1. As soon as the timer expires, the hardware sends H/w interrupt to CPU & OS takes the control back.

Cooperative mode alone is not sufficient. If P1 runs into infinite loop and we do not have non-cooperative way, OS will never be able to take back control.

Interprocess Communication

Scenarios in which processes need to communicate with each other:

Producer-consumer problem

One process is creating some data, and the other process is using that data (data sharing)

E.g. the process which is downloading the video needs to inform the process which will play the video after downloading finishes.

2. To ensure parallelism

Suppose process Z needs to happen after processes X and Y have happened.

One way to do that would be to do them sequentially - Execute X, then Y, then Z.

A faster way to do it will be to execute X and Y simultaneously. When they complete, they will inform and then process Z will begin.

Ways to achieve Interprocess Communication:

Shared memory

Two processes can write on a shared memory to communicate with each other.

P1 allocated some shared memory.

P1 shares that memory with P2.

P2 attaches that memory space to its space.

Finally, read and write can happen through which the two processes can communicate.

2. Message passing system (Sending and receiving messages)

P1 can send message which P2 will receive

If only P1 can send and P2 can receive (or vice versa) its called asymmetric, else if both can send and receive, it is called symmetric.

Messaging passing system can be classified based on the following criteria:

Direct v/s indirect

Direct: P1 knows pid (process ID) of P2 and vice versa

Signals are sent b/w P2 and P1 because they know each other's pid's.

Indirect: P1 and P2 do not know each other's pid's but they use a secondary medium to communicate i.e. either through ports, named pipes or mail boxes.

2. Based on synchronous nature

Blocking process: If a process P1 gets itself blocked/paused till it receives a signal from another process P2, P1 is called a blocking process otherwise, its a non-blocking process.

3. Based on buffering

Buffer: Space of memory to store some info. Communication is happening between P1 and P2. There must be some kind of buffer to store the info being sent by sender till the time receiver consumes the information.

Buffer might also be required at the sender's end. Sender will keep on storing info in its buffer till the time receiver is ready to receive the information.

Classification based on buffering:

Zero buffer:

Neither P1 nor P2 have buffer. Then what happens if P2 wants to send some information but P1 is not ready to receive it?

In this case, P1 and P2 both need to be of blocking nature.

2. Bounded buffer:

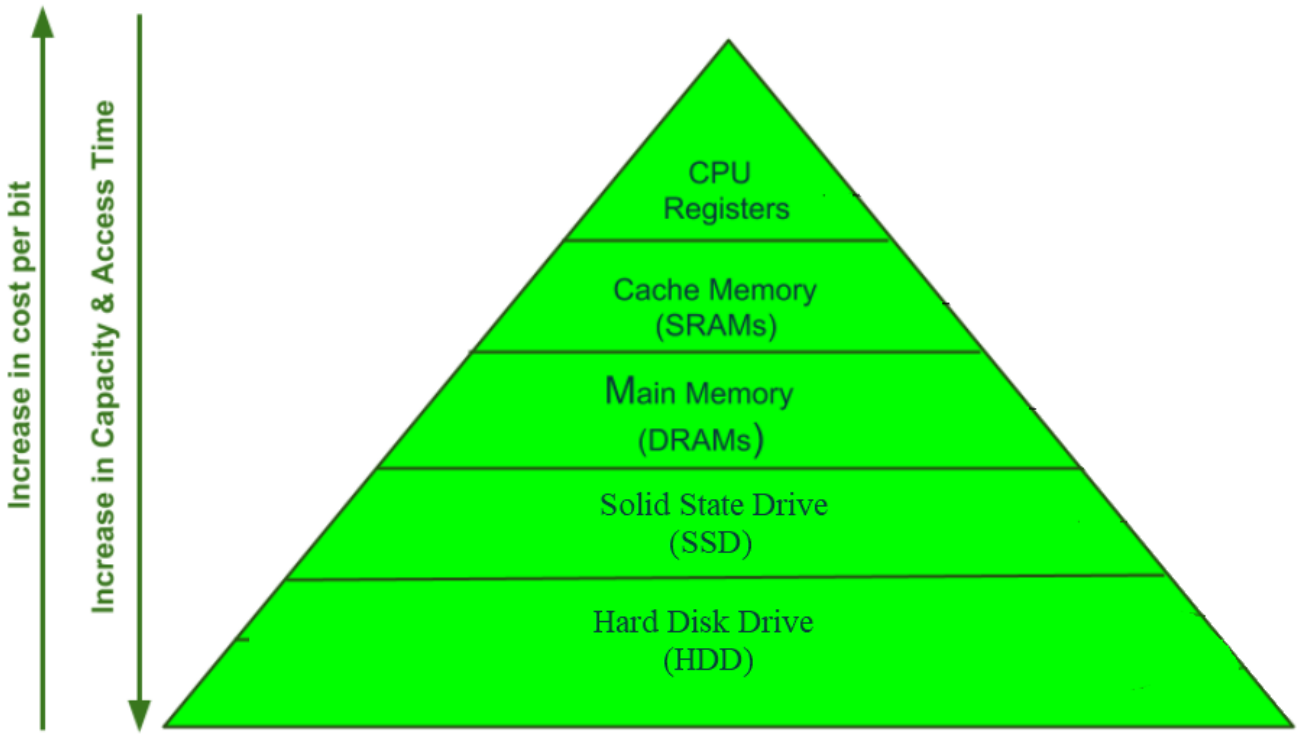
P1 and P2 have some fixed buffer. If P2's buffer is full, then P2 must wait for buffer to get empty before it can send more information to P1. Therefore, P2 must be of blocking nature. However if P2's buffer is empty, P1 can or cannot wait for P2's signal/info, it is under no obligation to wait. P1 can be blocking or non-blocking.

3. Unbounded buffer:Buffer size is infinite: Sounds impossible, but since hard disk's space is very large, we can theoretically call it infinity.Both sender or receiver can be blocking/non-blocking.

Memory Hierarchy Model

Memory Hierarchy:

HDD/Hard Disk Drive
SSD/Solid state drive
RAM/Main Memory/Random Access Memory
Cache
Registers



Accessing registers is fastest. Accessing HDD is slowest.

Cost of registers is highest, Cost of HDDs are lowest.

Volatile vs Non-volatile memory

Even if the machine is switched off, whatever is in non-volatile memory, is retained.

e.g. HDD, SSD - Non-volatile

RAM, Cache, Registers - Volatile

Enabling faster access

For storing important stuff (e.g, the processes that are running right now), we transfer them to main memory (RAM), so that CPU can access them faster.

On top of it, some part of that currently running process which are used a lot frequently are stored in cache.

On top of it, some variables which are right now in current instructions are put in registers.

Using RAM efficiently

It is not efficient to give complete RAM to the current running process and do it for each current running process. As during each context switch, the memory of the outgoing process has to be copied back to HDD, and the memory of the incoming process will need to be copied from HDD to RAM, which will consume a lot of time, as reads/writes in HDD is very slow.

The better idea is to share the RAM among processes. While sharing RAM, we do not want process P1 to access the memory region of process P2 and vice versa.

Address Translation

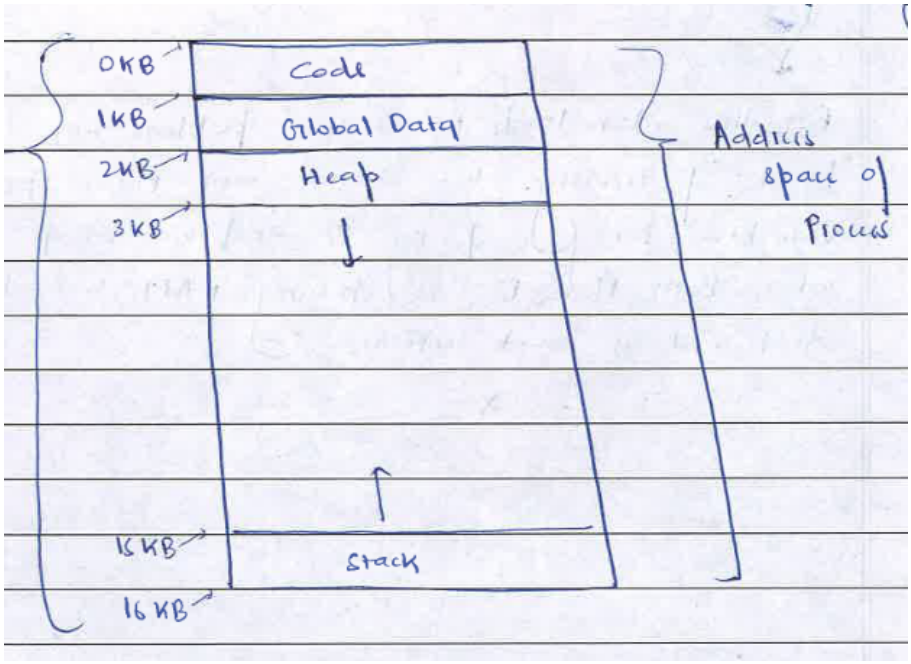
Virtual address: The addressing understood and used by the process to refer to its variable.
Physical address: The actual RAM address at which the variable is stored.

This mapping from virtual address to actual address in RAM is called address translation.

Physical address = Virtual address + Base address (Starting point of process in physical address)

Example:

Assume process P1 has been allocated 16 KBs of memory.



These 0Kb, 1KB, 2KB etc address milestones are virtual addresses actually which are in turn mapped to actual addresses in RAM.

Suppose X is a variable with virtual address (the addressing used by the process) 2.1 KB.
Let C be the starting address (Base address) of the process in the RAM
When the process tries to access the variable X (2.1 KB), this address is converted into physical address (2.1 KB + C), before it is accessed.

This base address C, is stored in a special register called base register. Every process has their own base register.

Memory protection:

Limit register: It stores the end point of the process's address space.

Whenever a process tries to access any variable, address translation will happen and will check whether the variable's physical address is within the scope of base and limit register. If we try to access memory outside the scope of LR, it leads to a trap and causes the segmentation fault error.

Two issues with the above memory management technique:

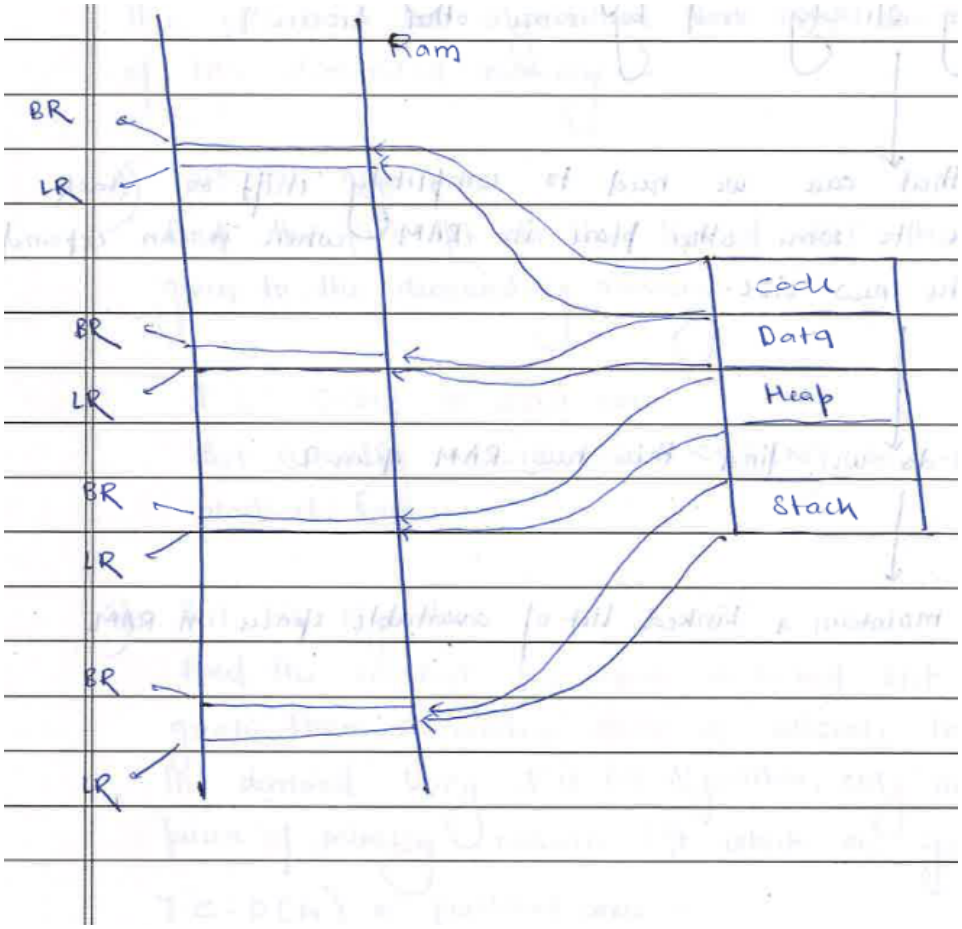
- Space between stack and heap is fixed but unutilized
- How to run processes that will take space greater than RAM size?

Segmentation

To manage the wasted space between the heap and stack space of a process, a technique called segmentation is used.

Rather than allocating one chunk of memory to a process, we segment out the different parts of memory and put them in different places in RAM. Each segment is identified by its own base and limit register.

So, the code part, data part, heap part and stack part can reside in different parts of the RAM.



The above technique solves the problem of gap between heap and stack.

But this technique does not actually solve the second problem (i.e. when the memory required by the process is more than the RAM itself).

Allocating space in RAM

Now post segmentation, suppose stack needs to expand. In that case, RAM will look for contiguous unused memory above/below stack memory and then add that memory to stack space.

In the case the memory above and below is being used by some other process, we need to completely shift our stack space to some other place in RAM where it can expand to the new size.

To enable searching for such available spaces in RAM, the OS maintains a linked list of free space in RAM.

Algorithms for memory allocation:

Now suppose we are demanding 1.5 MB from RAM (could be because of stack expansion or new process segmentation). There will be a few algorithms that take care of allocation of this demanded memory -

First fit algorithm

Find the first fit in the linked list that can be given to the demanding memory.

TC: .O(N) in worst case

But actually it will take much less than O(N) time for all practical purposes

2. Best fit algorithm

Find the smallest free space which is greater than demanded space and allocate that space to the demanding process. Using the best fit algorithm, only small pieces of memory remain left which are generally useless.
T.C. O(N)

3. Worst fit algorithm:

Find the largest free space in the linked list and allocate it to the demanded memory.

T.C. O(N)

Issues with the above algorithms: Fragmentation
Let's assume that there is 6.3 MB of unallocated space in RAM, but these are scattered in different portions of the RAM in small portions of 0.1-0.2 MB. If a process now requests for a contiguous chunk of 3 MB, the OS will not be able to allocate the memory: as there is no contiguous chunk of 3 MB. This happens despite the fact that the total free memory is more than the requested memory. This is due to fragmentation.

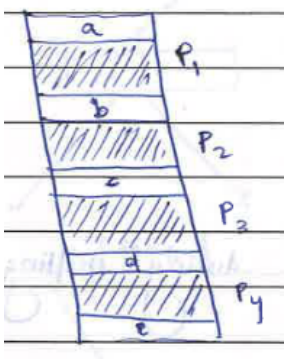
Fragmentation: Small portions of unallocated fragments in RAM - which are useless

Fragmentation

Fragments: Parts of RAM that are empty post segmentation (which are of no utility)

Types of fragmentation:

External fragmentation:



Suppose P2 wants allocation of 'm' size memory but a<m, b<m, c<m, d<m, e<m even though a+b+c+d+e > m. This is called external fragmentation

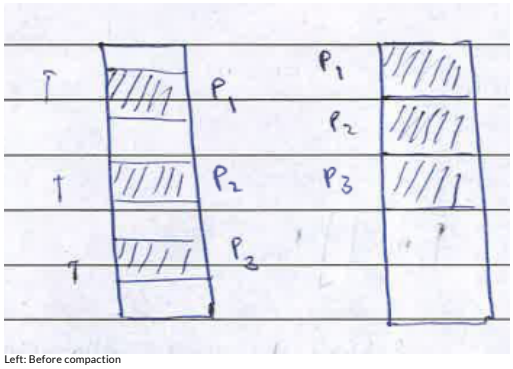
2. Internal fragmentation

The process itself asks for some extra memory which is wasted. Suppose P1 needs 2.01 MB, but it asks for 2.10 MB. The remaining 0.09 MB will be unutilized , but will still be unavailable to any other process. This is called internal fragmentation.

Internal fragmentation is not as big an issue because the extra memory that processes ask for is negligible.

Techniques for preventing external fragmentation:

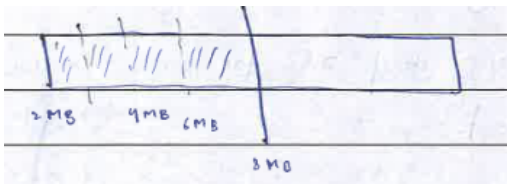
Compaction - After some time t, all processes are compacted to the top part of RAM, a procedure known as shifting up.



Left: Before compaction
Right: After compaction

Limitation: Compaction wastes CPU time.

2. Maintaining a separate RAM space for serving fixed size requests:
Suppose based on the history of memory allocation - OS deduces that most processes ask for 2 MB of data space. Now, the OS will keep a separate section of 2MB chunks only. All the free portion in this section of RAM will be in multiple of 2 MBs. And since most processes ask for 2 MB, if there is any space available in that section, the space will be allocated to the process - since it is guaranteed that the space will be in multiples of 2 MB in that section. This will eliminate the fragmentation at least in that section of the RAM.



Limitation: It's not necessary that all new requests are consistent with history.

3. Buddy allocation

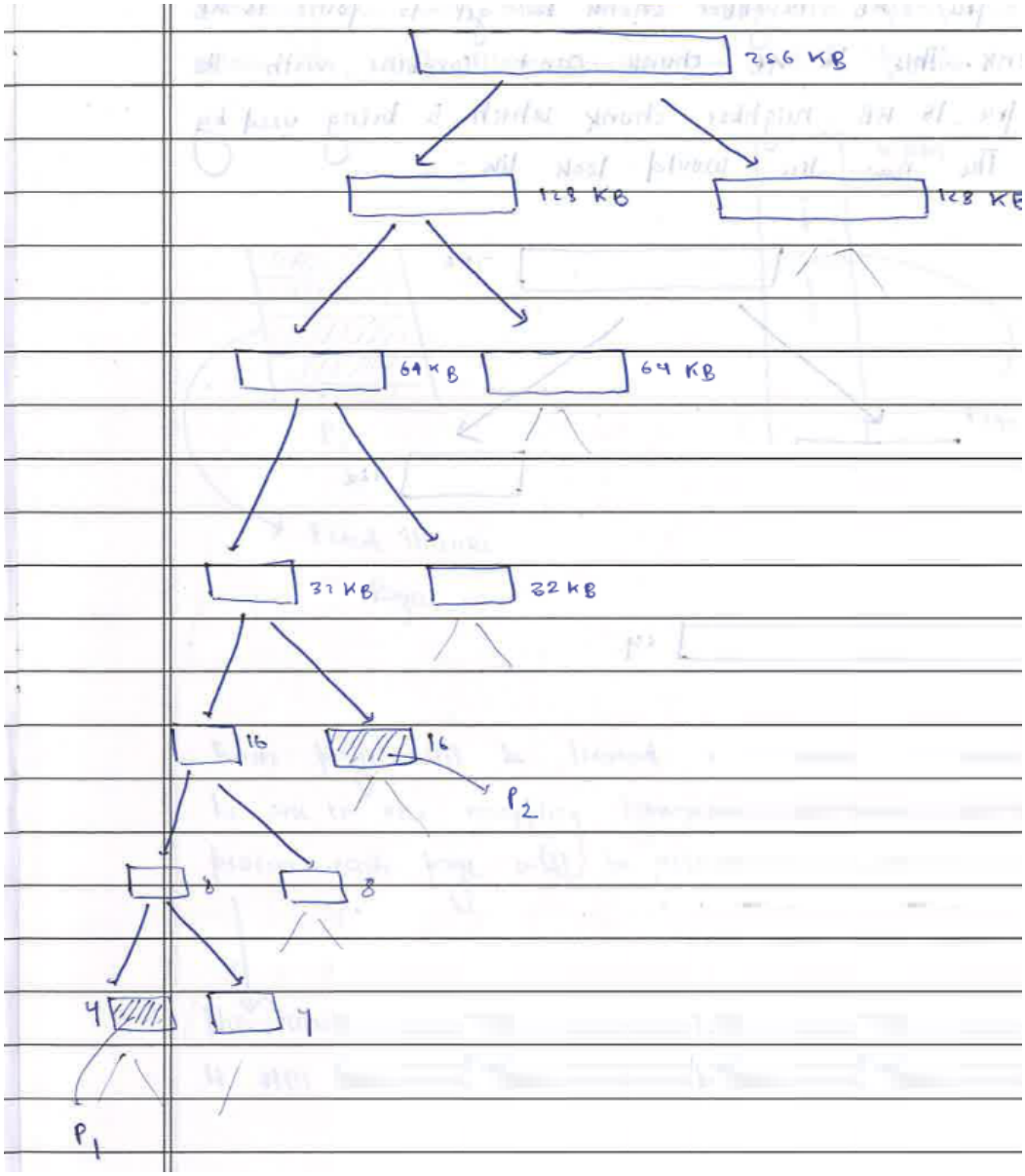
OS will only serve that space request such that space size is a power of 2.

When there's a memory request, the OS tries to break down the complete RAM into chunks (of power of 2) till it is able to find such a chunk that satisfies the request. When a process releases some memory, it is combined with the memory of the neighbor, if the neighbor is free.

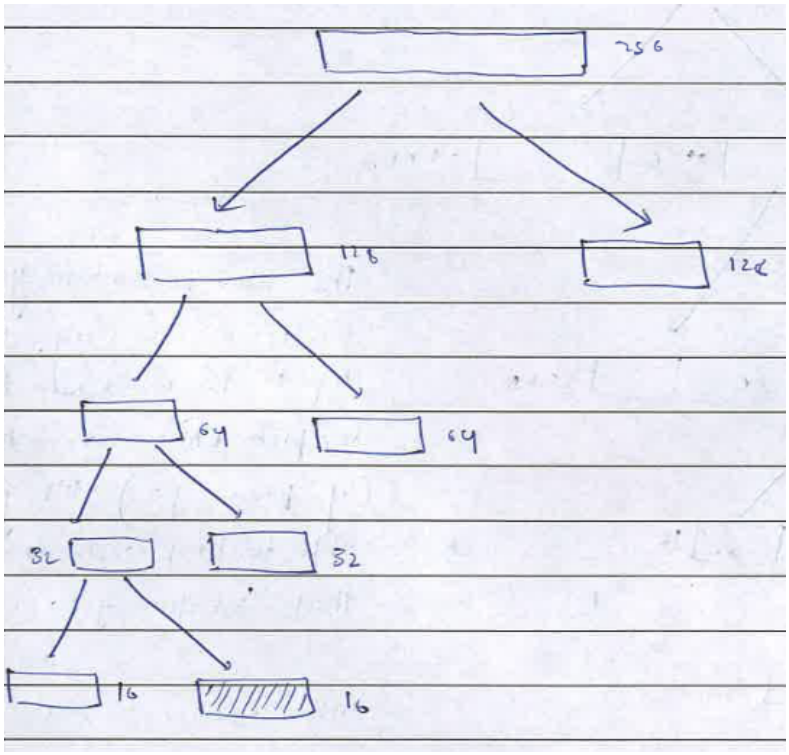
Illustration:

Suppose P1 wanted 4 KB. The whole RAM (of 25 KB) is divided into chunks until 4 KB is available. Now suppose P2 wants 16 KB. It can't get it from the first 16 KB chunk since it is not free. 4 KB of it being used by P1 so the second 16 KB chunk is allocated to P2.

Now suppose P1 releases itself. The 4 KB chunk gets free and it combines with its free 4 KB neighbor chunk to form 8 KB chunk. This 8 KB chunk combines with its free 8 KB neighbor chunk to get to form 16 KB chunk. This 16 KB chunk can't combine with its un-free 16 KB neighbor chunk which is being used by P2. The new tree would look like:



Before P1 releases memory

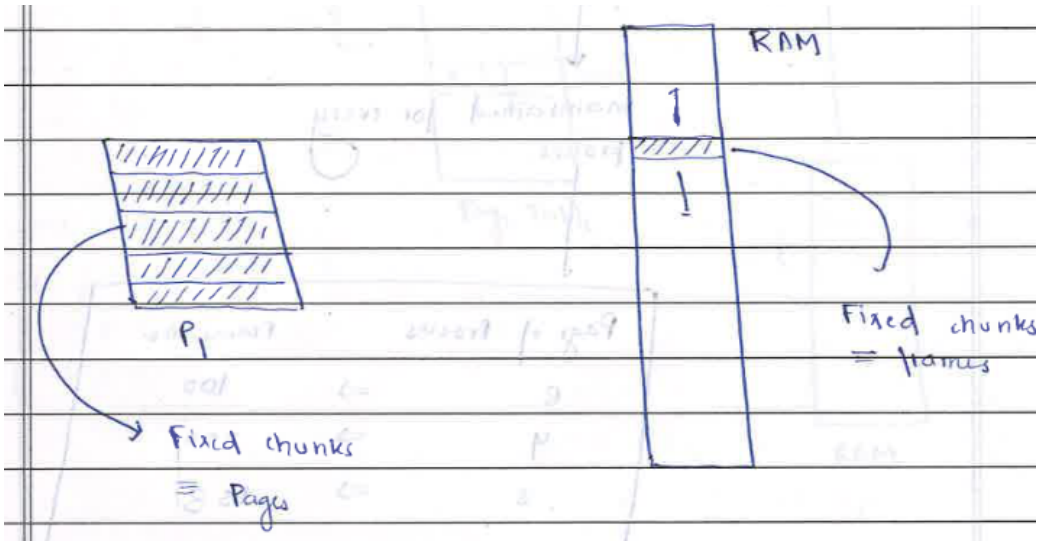


After P1 release memory.

Paging

Process's address space is divided into fixed size chunks. These chunks are called pages. The RAM space is also into chunks of the same size, and those chunks are called frames. Each page will be treated individually and there will be one to one mapping between pages and frames. Each page of a process will be allocated some frame in RAM. This completely solves the problem of external fragmentation, although it leads to internal fragmentation.

OS maintains a free frame list - list of frames that are free and from where frames will be given to processes.



Page Table:

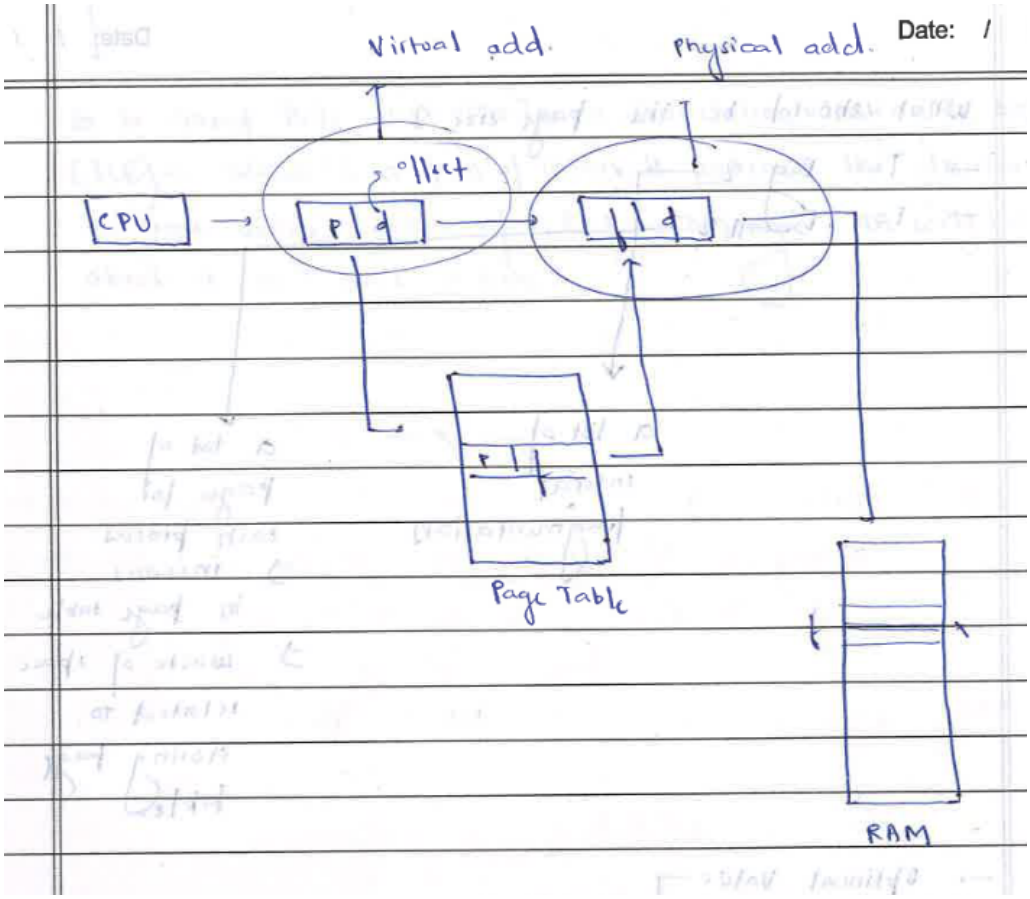
The page table is a table that maps the page number of a process to a corresponding frame number, i.e. it helps to determine where that page is actually stored in the RAM.
Each process has its own page table.

Address translation with paging:

Suppose a process wants to access variable X. Consider that X is present in page p at an offset of d. Then, p and d uniquely determine the variable's virtual address - the address understood by the process.

To be able to locate the variable in the physical RAM, we need to find the corresponding frame f in the RAM, and within that frame, we need to move to the offset value, d. The frame number is found through the page table of the process.

This gives us the physical address of the variable.



From physical address we can access where X variable is in RAM:
[(f-1) * page size + d]
(f-1) * Page size is the starting address of the frame f.

So now address translation becomes a 2 step process. We first access the frame number using page table and then with frame number and offset, we can get the position of X in RAM.

Ensuring memory protection:
Paging removes the concept of bases and bounds. Maintenance of the page table is sufficient to ensure memory protection.

Impact of page size:

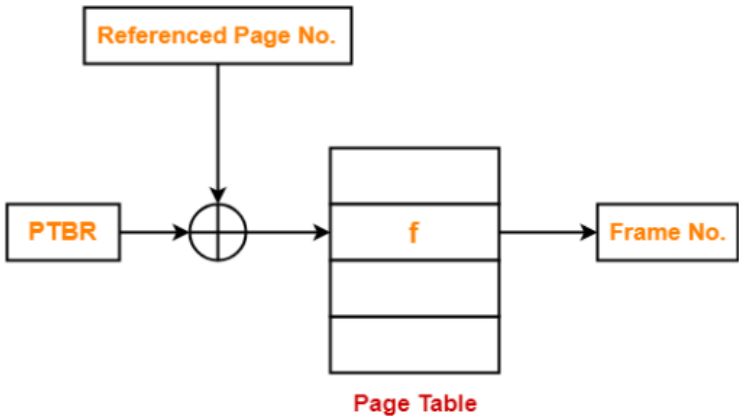
If page size is too large, then it leads to internal fragmentation

If it is too small, a lot of pages for each process leads to increase in page table and waste of space related to storing page value

Optimal value is selected by OS developer

Where does this page table reside?

In RAM. To know where a page table for a specific process resides in RAM, we have page table base register. The PTBR is a register that maintains the address of page table for the current process.



Based on this, you can observe that every variable access requires 2 RAM accesses (one to look up the frame number value from the page table and the other to read the value at the physical address). This is double the number of accesses required in non-paging address translation.

To avoid this, we introduce a "translation lookaside buffer" (TLB), which is a part of cache. It prevents the increase in time usage because of p-f mapping (page to frame).

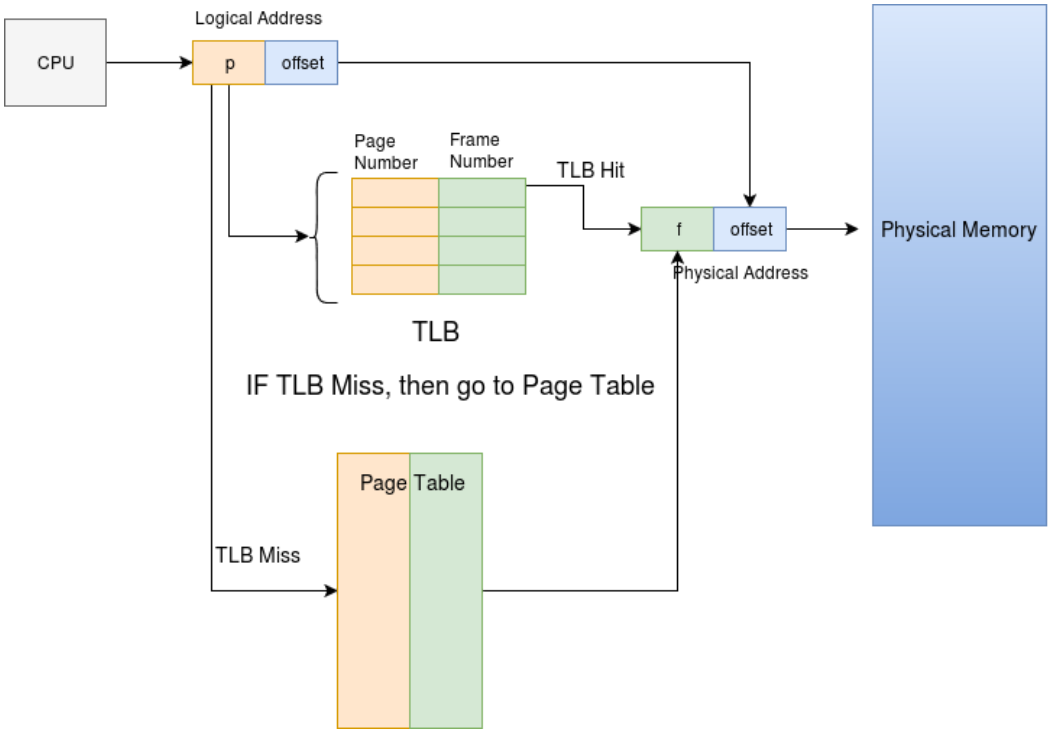
Translation Lookaside Buffer

The idea is to store the frame numbers of recently accessed pages in cache.

For example, 3 -> 150. This key value pair would be stored in the TLB, indicating page 3 is stored in frame 150.

The next time, page 3 is accessed, we don't have to do a lookup in page table, We can do a faster lookup in cache for the same (Access time of cache is much less than access time of RAM)

New address translation with TLB in place:



Note that in case of a TLB miss, along with looking up the page table to find the p-f value, we also need to cache this p-f value in the TLB cache, so that its available in the cache for future requests.

Cache replacement policy:

If TLB is full, then for storing a p-f in TLB, we remove an older (p-f) in TLB using "cache replacement policy".

One policy could be: LRU. (Remove the least recently used)

LRU in operation (example scenario)

Suppose TLB is as shown in the table:

Page	Frame
1	101
2	105
3	103

And suppose page 1,2,3 have been accessed till now. Now say page 4 is accessed and 4-109 is to be placed in TLB. The least recently used (p-f) in TLB is (2-105). So we replace it by (4-109).

Limitation of LRU:

Lets say that there are n slots in the TLB table. A hacker can create a process with (n+1) pages and then start accessing all those (n+1) pages sequentially in infinite loops:

```
for ( i = 1 to infinity )
for( k = 1 to n+ 1)

 access page k
```

Each time there will be a TLB miss and it will slow down the machine. So, in the worst case, TLB can make the address translation slower.

To avoid this, rather than using LRU, we use a random replacement policy.

Benefits of TLB:

Spatial locality

Suppose a process arr[1000] is part of some page: then for all arr[0], arr[1],... arr[999], once even if one of arr [0:999] is inserted into TLB (i.e. the process of arr[]), then accessing arr[i]'s will be short and quick because every time we access arr[i] for different i, the p-f value is already there in the TLB.

2. Temporal benefit:

The idea is that if variable X is being accessed at any given time, there is a high chance it will be accessed again after some time.

Stale values in TLB:

TLB is a common cache accessed by each process.So now, if a context switch happens, issues might arise:

Suppose a process (P1's) page no. 3 is in TLB. Now suppose context switch happens and P2 is in charge and tries to access page no.3. It will end up getting the response of the previous process.

To resolve it we can use either of the following techniques:

Flush TLB on context switch
We can use a valid bit in our TLB for each p-f value.

Valid bit	Tag (Virtual page #)	Data (Physical page #)
1	01AF4	FFF
0	0E45F	E03
0	012FF	2F0
1	01A37	788
1	02BB4	45C
0	03CA0	657

Bit 1 denotes that all these page nos are of the current process.

Now when the context switch happens, all valid bits are made 0 to ensure that all these page numbers are not part of the current process.

3. We can use address space identifiers (ASI) - process Ids in TLB to identify which process the p-f value belongs to.

Other uses of the page table

Permission bit: We store this information for every page-table entry, i.e. for each page in table, we store whether the page is read only or is it read-write.

Scenario where permission bit is useful:

If a process's page contains code(not data/variables) of the process, it need not be modified. So, it makes sense to make the page a read only page.

Similarly, if a page contains the stack of the process, then it should be a read-write page.

2. **Shared bit**: The shared bit helps in identifying read-only pages which are shared between processes .

Scenario where shared bit is useful:

Suppose 4 processes P1,P2,P3,P4 are accessing math.h library to get sqrt() function. Every process will have a separate page that would store the same code of sqrt() function which is read only. In such a case, the 3 extra copies of the sqrt function() are consuming extra RAM space and are wasteful. Shared bit helps in sharing this page across the processes.

Hierarchical & Inverted Page Tables

Imagine we have a 32 bit computer, this means that the virtual address space for each of the process it allocates is 4 GB (2^32 bytes).

Suppose the page size is around 4 KB for the process. 4 KB = 2^12 bytes

No. of pages the process can have = 2^32/ 2^12 = 2^20 pages

Suppose, corresponding to each page of the process, the page table entry is of size 4 bytes. Then for storing the page table entries for all the pages, 2^20 * 4 bytes = 4 MB space will be required.

And each process' page table will take up 4 MB memory space. This is a waste of RAM space.

Possible ways to reduce this wastage:

Increase page size
Increasing the page size will reduce the number of pages per process, and that will reduce the size of the page table entry for the process, which is directly proportional to the number of total pages
.
Issue with this technique: Results in increase of internal fragmentation

Avoid page table entries for empty pages
A process during its run won't be using all its 2^20 pages (and hence 4 GB of space) simultaneously. A lot of the pages at any given point of time won't be used.

But this is not a practical solution.

Issue with this technique:
1) If a process requests extra memory and uses a new page, a new page table entry will have to be created.
2) How will a process know which pages are empty and can be used?

3. Multi-level/tier page system

Page-Directory	Page
1-64	1
65-128	NULL

Page directory is like the page table for the page table. Now say if pages 65-128 are empty, then we can now not have pages 65-128 in the page table and the indication of emptiness can be done in the page directory itself. That is, no link from (65-128) to pages.

Issue with this technique: We would increase the count of RAM accesses by 1. (3 levels of indirection/3 RAM accesses needed to access any variable)

We can mitigate this problem to some extent by using TLB, but then TLB misses in this case are going to be very expensive.

4. Inverted page table

Rather than mapping being from page number to frame number, let us have mapping from frame number to page number

Because of this, all those pages which are not in memory will not have any entry in the page table.

Issue with this technique: If we want to access some variable, which is in the page 3 of PID 100, then we will have to do a linear access of the inverted page table to find the frame number for (pid: 100, page: 3).

Iteration of a complete inverted page table will be costly.

So, this technique saves space but at the expense of increased access time.

A mixture of all the above ideas are implemented to actually reduce the page table size.

Swap Space

Suppose there is a process P1 that has a huge memory requirement, and requires lots of pages.

RAM - having finite memory - will have limited frames only. RAM will need to serve other processes as well.

So, all the pages of the process may not be accommodated in the RAM at a given time.

The remaining pages of the process are stored in HDD.

Special memory is reserved in HDD for storing pages: swap space.

Processes are able to identify if a particular page is in RAM memory or in Swap Space by using the present bit.

Present bit = 1 means the page is in RAM

Present bit = 0 means the page is in swap space

Page fault:

If we try to access some particular page which is not there in the RAM, then it leads to page fault.

How page fault is handled:
If page fault happens, then the page fault handler runs.

Page fault handler will find the swapped space address of the page, which is stored in the page table, and will reach that address in HDD.
It will then shift the page from swap space to RAM.
It will then make the present bit 1 and update the frame number of the page.

Page replacement policy

If the RAM is already full and a page needs to be shifted from SWAP space to RAM, then we need to put one of the existing pages in RAM into swap space. So for this, we require a page replacement policy.

Page replacement policy: Policy required to remove which page so as to accommodate the incoming page. We will read about them in the next section.

Time analysis corresponding to page fault:

p=> probability of page fault

Tf=> time to process page fault

Tn=> Normal time to process

Avg Time of accessing an address (T_avg) = p*Tf + (1-p)* Tn

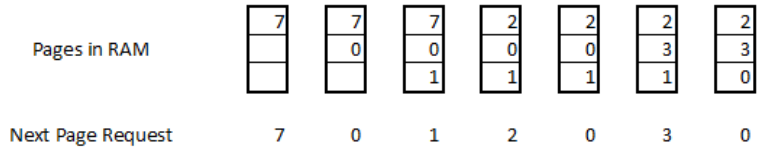
Suppose Tn = 200 ns, Tf = 8 ms = 8*10^6 ns.

If p = 0.001,

T_avg = 8.2 microseconds, which is not an acceptable time. We need to ensure that P is very small, 1/ 400,000, such that T avg is around 200 ns.

Page Replacement Policies

Compulsory faults vs capacity fault



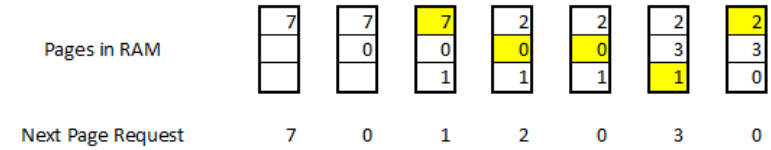
a) Compulsory faults: The faults which occur when all the frames in the RAM are not filled. For example, when we access page 7 for the first time, the frames in the RAM were empty. So page 7 had to be brought into the RAM from the hard disk. Such page faults cannot be avoided.

b) Capacity faults(cold misses): The page faults occurring when the RAM is full to its capacity are called capacity faults. The number of capacity faults depends on the page replacement policy (PRP).

We can only try to reduce cold misses/faults. Compulsory faults can't be avoided.

Some page replacement policies:

FIFO page replacement policy:



(Yellow ones indicate the next page to be replaced, i.e. the first page to occupy the RAM, among the other pages in the RAM)

In the above diagram, page 7 is the first page to be brought to the RAM. So,

when the RAM gets filled up and a new page has to be put into the RAM (when

page 2 is requested in the above diagram), page 7 has to be swapped out of the

RAM. This is the FIFO principle: the first page to be in, must be the first page to

go out in case the RAM is full and a new page has to be accommodated.

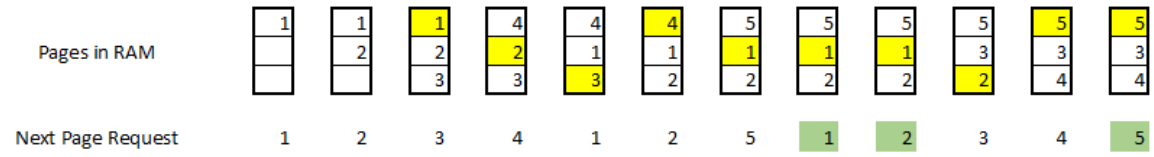
Cons of FIFO PRP:

1. Not at all intelligent: It does not consider which pages are more used/less used, used most recently etc - this causes more misses.

2. Belady's anomaly: If we increase the capacity in RAM, ideally the number of misses should decrease. But interestingly, in FIFO PRP (Page replacement policy), increasing RAM capacity increases misses.

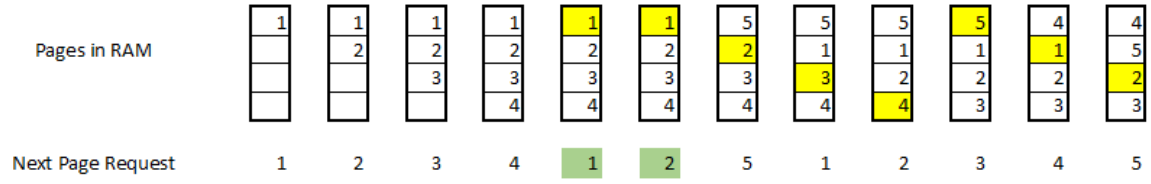
For example, consider the following sequence of page requests:
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

With 3 frames in RAM



We can see that with 3 frames in RAM, the number of page faults (every page request except the green ones) is 9.

With 4 frames in RAM (increased RAM capacity)



We can see that with 4 frames in RAM, the number of page faults (every page request except the green ones) is 10. Thus increasing the RAM capacity can lead to more page faults. This is an example of Belady's anomaly.

Random replacement - Statistically, it performs better than FIFO
Using history:

Most frequently used:

Idea: Replace most frequently used
Limitation: MFU does not take into account the importance of spatial and temporal locality. Hence MFUs do not perform that well

Least frequently used:

Idea: Replace least frequently used page
Since it is based on spatial and temporal locality, it performs better than MFU PRP
Limitation: We don't take into consideration 'when'. We are just making decisions based on frequency. If a variable was accessed very frequently, some time back, then probably we have actually moved away from that part of code which was accessing it and that page is not probably required.
A better strategy would be to implement Least Recently Used policy

LRU Page Replacement

LRU policy

Idea: Remove ones that were used last

Advantage: It is designed keeping in mind the principle of locality - it is more likely that the recently used page will be reused.

Pages in RAM	7	7	7	2	2	2	2	2	2	2
		0	0	0	0	0	0	0	1	1
			1	1	1	1	1	4	4	4
Next Page Request	7	0	1	2	0	0	2	4	1	0

(Yellow ones indicate the next page to be replaced, i.e. the least recently used page, among the other pages in the RAM)

For random cases, it might happen that LRU policy might perform worse than other page replacement policies.

Cons of LRU:

Implementing LRU is a problem in itself because its implementation takes a lot of time and space.

LRU approximation - an improvement over LRU

Idea: Reference bits need to be introduced in the page table. Initially, all reference bits are 0. They are changed to 1 when a page is accessed. Each page has its own reference bits.

Suppose RAM is filled with pages 1,2,3 and 4, and initially the reference bits are:

Pages	Bits
1	0
2	0
3	0
4	0

And now pages accessed are 3,1,1,3,5,2

Reference bits of pages 1-4, as page requests are made

1	0	1	1	1		
2	0	0	0	0		
3	1	1	1	1		
4	0	0	0	0		
	3	1	1	3	5	2

The moment we arrive at 5, we check pages with reference bits 0. We randomly replace any of those pages - which is an approximation of LRU (since pages with ref bits 1 are used recently and vice versa)

If every bit is 1, (ref bits of all pages in RAM are 1 and we need to do replacement). Then again, there will be a random replacement among them.

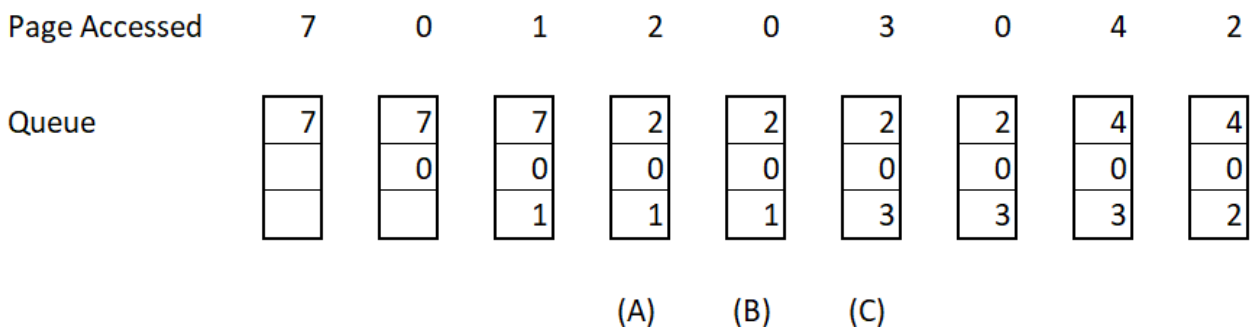
Algorithm:

Find the set of pages with reference bit 0
If none of the page's reference bit is 0, randomly replace any page from the RAM
Else, replace any of the page with reference bit 0.

Second Chance Algorithm

Idea: Hybrid algorithm of PRP that uses logic of FIFO and LRU approximation. We maintain a queue of all the pages currently in memory.

Example operation:



In the above diagram,

(A): FIFO is used to swap in 7 and swap out 2.

(B): Reference bit of 0 made to 1

(C): It was the chance of 0 to get out according to FIFO, but since the reference bit of 0 is 1, we give 0 a second chance. We give it a second chance by not removing it and then switching off its reference bit back to 0.

If a page has a reference bit equal to 1, it will be given one safety.

Possibile extensions:

This second chance algorithm can be extended and rather than giving one chance, we can give 2 chances or more (by using 2 bit references or more)

If the page has a non-zero reference and it is its chance to get out, we give it another chance and reduce its reference by 1.

Dirty bits

If a page P is modified after being in RAM, if it gets replaced by some other page, the Hard Disk's replica of the page should be updated, as the RAM's version of the page P is the correct one and the one in Hard Disk is stale. So, the modified page needs to be copied to Hard Disk, otherwise the hard disk's version of the page, if brought into RAM in future, would be incorrect.

But if the page is NOT modified (written on) at any point from the time it was fetched to RAM, it does not need to be copied back to the Hard Disk when being swapped out, as it is essentially the same copy which was fetched from the hard disk and is unmodified.

Dirty bits maintain the state of the page: if dirty bit is 0, it indicates that the page is unmodified, otherwise it is modified.
This bit is maintained by the OS - whenever some process writes on the page, the dirty bit is set to 1.
This dirty bit comes into consideration in second chance algorithm. Suppose a page P needs to be removed from RAM, then the page which has dirty bit 0 is more favorable for removal as compared to a page whose dirty bit is 1 (as the unmodified page need not be copied back to HDD, which will save time.)

Process	Ref	Dirty
P1	0	0
P2	0	1
P3	1	0
P4	1	1

Here, the reference bits for both P1 and P2 are 0, but the dirty bit of P1 is 0, whereas the dirty bit for P2 is 1. So, if the OS swaps out P2, it would need to copy P2's data back to HDD< but it would be saved from this copying effort if it decides to swap out P1.