**MEMORY PARTITIONING**

**Fixed Partitioning** - Split the physical memory into partitions into each a process may be assigned

**Two difficulties:-**

The programmer may have to rewrite a program in order to fit into even the largest available partition. Overlaying may be used.

Internal fragmentation typically results--there's memory that is unused within each partition.



**Dynamic Partitioning**

Dynamic parititioning has the OS allocate the space for a process when it enters the system.External fragmentation occurs here. Compaction may be used to help recover the space.

Placement algorithms

**best fit** - waste the least amount of space from the list of available blocks

**worst fit** - waste the most amount of space (hope that what's left over is useful)

**first fit** - performance gain

**next fit**– performance

**Simple Paging:-**

Main memory is partition into equal fixed-sized chunks (of relatively small size)

Trick: each process is also divided into chunks of the same size called pages. The process pages can thus be assigned to the available chunks in main memory called frames (or page frames)

**Simple Segmentation:-**

Each program is subdivided into blocks of non-equal size called segments. When a process gets loaded into main memory, its different segments can be located anywhere Each segment is fully packed with instructs/data: no internal fragmentation. There is external fragmentation; it is reduced when using small segments.



**PAGE REPLACEMENT METHOD :-**

**First In First Out (FIFO) algorithm**

Oldest page in main memory is the one which will be selected for replacement.

**Least Recently Used (LRU) algorithm**

Page which has not been used for the longest time in main memory is the one which will be selected for replacement.

**Least Frequently Used (LRU) algorithm**

if the page had not been used often in the past. This policy keeps count of the number of times that page is accessed. Pages with the lowest counts are replaced while pages with higher counts remain in primary memory

**Optimal page replacement algorithm**

It has lowest fault rate for any page reference stream

- idea: replace the page that will not be used for longest time in the future

-Replace the page that will not be used for the longest period of time

- we can’t predict the future usage. So it will not be implemented practically.

**THRASHING** :-

Continuously swapping pages is called thrashing. CPU spends most of the time in it. Reason : Running too many program simultaneously.

**THRASHING :-**

In a virtual storage system (an operating system that manages its logical storage or memory in units called pages), thrashing is a condition in which excessive paging operations are taking place. A system that is thrashing can be perceived as either a very slow system or one that has come to a halt.

**LOCALITY REFERENCE:-**

**Temporal Locality (locality in time) –**

If an item is referenced, it will tend to be referenced again soon.

**Spatial Locality (locality in space) –**

If an item is referenced, items whose addresses are close by will tend to be referenced soon.

**Sequential locality:-**

Here storage is accessed sequentially, in descending or ascending order.

**Working set:-**

Working set (t, k) at an instant of time, t, is the set of pages that have been referenced in the last k time units.

**Virtual Memory:-**

Physical memory is broken into fixed-sized blocks called frames.

Logical memory is also broken into blocks of the same size called pages.

**Belady's Anomaly:-**

When adding more page space we'll have more page faults. Eg. FIFO and Random page replacement policy

**TRANSLATION LOOK ASIDE BUFFER:-**

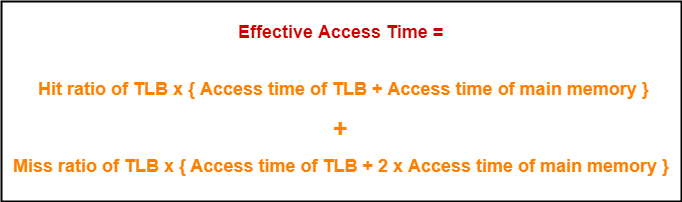
In Virtual memory systems, the cpu generates virtual memory addresses. But, the data is stored in actual physical memory i.e. we need to place a physical memory address on the memory bus to fetch the data from the memory circuitry. So, a special table is maintained by the operating system called the Page table. This table contains a mapping between the virtual addresses and physical addresses. So, every time a cpu generates a virtual address, the operating system page table has to be looked up to find the corresponding physical address. To speed this up, there is hardware support called the TLB. The TLB is a high speed cache of the page table i.e. contains recently accessed virtual to physical translations. TLB hit ratio- A TLB hit is the no of times a virtual-to-physical address translation was already found in the TLB, instead of going all the way to the page table which is located in slower physical memory. TLB hit ratio is nothing but the ratio of TLB hits/Total no of queries into TLB. In the case that the page is found in the TLB (TLB hit) the total time would be the time of search in the TLB plus the time to access memory, so

TLB\_hit\_time := TLB\_search\_time + memory\_access\_time

In the case that the page is not found in the TLB (TLB miss) the total time would be the time to search the TLB (you don’t find anything, but searched nontheless) plus the time to access memory to get the page table and frame, plus the time to access memory to get the data, so

TLB\_miss\_time := TLB\_search\_time + memory\_access\_time + memory\_access\_time

But this is in individual cases, when you want to know an average measure of the TLB performance, you use the Effective Access Time, that is the weighted average of the previous measures



**SCHEDULER**

**Long Term Scheduler :-**

It is also called a job scheduler. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.

**Short Term Scheduler :-**

It is also called as CPU scheduler. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.

Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

**Medium Term Scheduler :**

Medium-term scheduling is a part of swapping. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the swapped out-processes. This process is called swapping, and the process is said to be swapped out or rolled out. It is called swapper.

**SCHEDULING ALGORITHM**

**First Come First Serve (FCFS)**

Jobs are executed on first come, first serve basis.

Easy to understand and implement.

Its implementation is based on FIFO queue.

Poor in performance as average wait time is high.

**Shortest Job Next (SJN)**

This is also known as shortest job first, or SJF

This is a non-preemptive scheduling algorithm.

Best approach to minimize waiting time.

The processer should know in advance how much time process will take.

**Priority Based Scheduling**

Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.

Each process is assigned a priority. Process with highest priority is to be executed first and so on.

Processes with same priority are executed on first come first served basis.

Priority can be decided based on memory requirements, time requirements or any other resource requirement.

**Shortest Remaining Time**

Shortest remaining time (SRT) is the preemptive version of the SJN algorithm. The processor is allocated to the job closest to completion but it can be preempted by a newer ready job with shorter time to completion. It is often used in batch environments where short jobs need to give preference.

**Round Robin Scheduling**

Round Robin is the preemptive process scheduling algorithm.

Each process is provided a fix time to execute, it is called a quantum.

Once a process is executed for a given time period, it is preempted and other process executes for a given time period.

Context switching is used to save states of preempted processes.

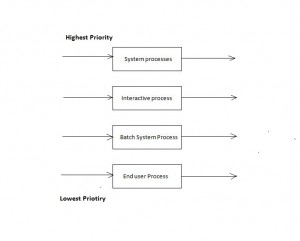
**Multilevel Queue scheduling:-**

Multi level queue scheduling was created for situation in which processes are easily classified into different groups.

1.Processes are divided into different queue based on their type. Process are permanently assigned to one queue, generally based on some property of process i.e. system process, interactive, batch system, end user process, memory size, process priority and process type.

2.Each queue has its own scheduling algorithm. For example interactive process may use round robin scheduling method, while batch job use the FCFS method.

In addition, there must be scheduling among the queue and is generally implemented as fixed priority preemptive scheduling. Foreground process may have higher priority over the background process



**Multilevel feedback Queue scheduling:-**

It is an enhancement of multilevel queue scheduling where process can move between the queues. In approach, the ready queue is partitioned into multiple queues of different priorities. The system use to assign processes to queue based on their CPU burst characteristic. If a process consumes too much CPU time, it is placed into a lower priority queue. It favors I/O bound jobs to get good input/output device utilization. A technique called aging promotes lower priority processes to the next higher priority queue after a suitable interval of time.

In general, a multilevel feedback queue scheduler is defined by the following parameters:

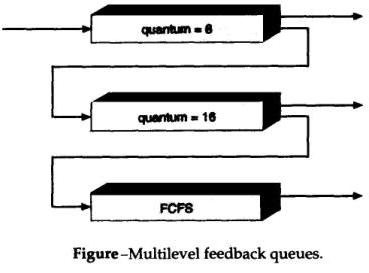
-> The number of queues

-> The scheduling algorithm for each queue

-> The method used to determine when to upgrade a process to a higher-priority queue

-> The method used to determine when to demote a process to a lower-priority queue

-> The method used to determine which queue a process will enter when that process needs service



**Spooling:-**

Spooling is an acronym for simultaneous peripheral operations on line. Spooling refers to putting data of various I/O jobs in a buffer. This buffer is a special area in memory or hard disk which is accessible to I/O devices

**DEAD LOCK**

Mutual Exclusion (It means making all resources sharable)

Hold and Wait

No Pre-emption

Circular Waiting

**Deadlock Prevention:-**

We can prevent Deadlock by eliminating any of the above four condition.

**Eliminate Mutual Exclusion**

It is not possible to dis-satisfy the mutual exclusion because some resources, such as the tape drive and printer, are inherently non-shareable.

**Eliminate Hold and wait**

1. Allocate all required resources to the process before start of its execution, this way hold and wait condition is eliminated but it will lead to low device utilization. for example, if a process requires printer at a later time and we have allocated printer before the start of its execution printer will remained blocked till it has completed its execution.

2. Process will make new request for resources after releasing the current set of resources. This solution may lead to starvation.

**Eliminate No Preemption:-**

Preempt resources from process when resources required by other high priority process.

**Eliminate Circular Wait:-**

Each resource will be assigned with a numerical number. A process can request for the resources only in increasing order of numbering.

For Example, if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.

**Deadlock Avoidance:-**

Deadlock avoidance can be done with Banker’s Algorithm.

**Banker’s Algorithm**

Bankers’s Algorithm is resource allocation and deadlock avoidance algorithm which test all the request made by processes for resources, it check for safe state, if after granting request system remains in the safe state it allows the request and if there is no safe state it don’t allow the request made by the process.

Inputs to Banker’s Algorithm

1. Max need of resources by each process.

2. Currently allocated resources by each process.

3. Max free available resources in the system.

Request will only be granted under below condition.

1. If request made by process is less than equal to max need to that process.

2. If request made by process is less than equal to freely available resource in the system.

**Types of Synchronization Problems:-**

1. locks

2. condition variables (implemented as pthread\_cond\_\* under pthreads)

3. semaphores

**Binary Semaphore :-**

* the semaphore value is restricted to 0 and 1.
* P succeeds only when the semaphore value is 1.
* V does not change the semaphore value when it is 1. (Thus successive Vs are lost.)

**Counting Semaphore :-**

**P and V operator:-**

1.The wait operation (P) decrements the semaphore

2.The Signal operation(v) increments the semaphore

**Example)** At a particular time of computation the value of a counting semaphore is 7.Then 20 P operations and 15 V operations were completed on this semaphore.The resulting value of the semaphore is : (GATE 1987)

(Ans : 2)

**Classical IPC Problems:-**

The operating systems literature is full of interprocess communication problems that have been widely discussed using a variety of synchronization methods. In the following sections we will examine three of the better-known problems.

1. Dining Philosophers Problem

2. The Readers and Writers Problem

3. The Sleeping Barber Problem

**Dirty Read:-**

Dirty read occurs when one transaction is changing the record, and the other transaction can read this record before the first transaction has been committed or rolled back. This is known as a dirty read scenario because there is always the possibility that the first transaction may rollback the change, resulting in the second transaction having read an invalid data.

**FILE ALLOCATION METHOD/DISK ALLOCATION METHOD**

**Contiguous Allocation**

Each file occupies a contiguous address space on disk.

Assigned disk address is in linear order.

Easy to implement.

External fragmentation is a major issue with this type of allocation technique(first bit, best bit)

**Linked Allocation**

Each file carries a list of links to disk blocks.

Directory contains link / pointer to first block of a file.

No external fragmentation

Effectively used in sequential access file.

Inefficient in case of direct access file.

In linked allocation, each file is a linked list of disk blocks. The directory contains a pointer to the first and (optionally the last) block of the file. For example, a file of 5 blocks which starts at block 4, might continue at block 7, then block 16, block 10, and finally block 27. Each block contains a pointer to the next block and the last block contains a NIL pointer. The value -1 may be used for NIL to differentiate it from block 0.

**Indexed Allocation**

Provides solutions to problems of contigous and linked allocation.

A index block is created having all pointers to files.

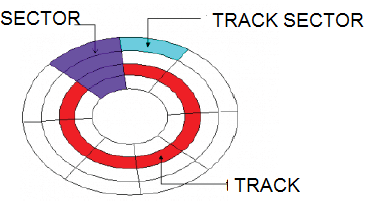
Each file has its own index block which stores the addresses of disk space occupied by the file.

Directory contains the addresses of index blocks of files

**Multi Level Indexed Allocation :-**

This scheme supports very large file sizes

**DISK ARM SCHEDULING**

****

**Seek Time**:

Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.

**Rotational Latency**:

Rotational Latency is the time taken by the desired sector of disk to rotate into a position so that it can access the read/write heads. So the disk scheduling algorithm that gives minimum rotational latency is better.

**Transfer Time**:

Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.

**Disk Access Time**: Disk Access Time is:

Disk Access Time = Seek Time + Rotational Latency + Transfer Time

**Disk Response Time**: Response Time is the average of time spent by a request waiting to perform its I/O operation. Average Response time is the response time of the all requests. Variance Response Time is measure of how individual request are serviced with respect to average response time. So the disk scheduling algorithm that gives minimum variance response time is better.

**Disk Scheduling Algorithms**

**FCFS:** FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.

**Advantages:**

Every request gets a fair chance

No indefinite postponement

**Disadvantages:**

Does not try to optimize seek time

May not provide the best possible service

**SSTF:** In SSTF (Shortest Seek Time First), requests having shortest seek time are executed first. So, the seek time of every request is calculated in advance in queue and then they are scheduled according to their calculated seek time. As a result, the request near the disk arm will get executed first. SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of system.

**Advantages:**

Average Response Time decreases

Throughput increases

**Disadvantages:**

Overhead to calculate seek time in advance

Can cause Starvation for a request if it has higher seek time as compared to incoming requests

High variance of response time as SSTF favours only some requests

**SCAN:** In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works like an elevator and hence also known as elevator algorithm. As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

**Advantages:**

High throughput

Low variance of response time

Average response time

**Disadvantages**:

Long waiting time for requests for locations just visited by disk arm

**CSCAN:** In SCAN algorithm, the disk arm again scans the path that has been scanned, after reversing its direction. So, it may be possible that too many requests are waiting at the other end or there may be zero or few requests pending at the scanned area.

These situations are avoided in CSCAN algorithm in which the disk arm instead of reversing its direction goes to the other end of the disk and starts servicing the requests from there. So, the disk arm moves in a circular fashion and this algorithm is also similar to SCAN algorithm and hence it is known as C-SCAN (Circular SCAN).

**Advantages:**

Provides more uniform wait time compared to SCAN

**LOOK:** It is similar to the SCAN disk scheduling algorithm except the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus it prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

**CLOOK**: As LOOK is similar to SCAN algorithm, in similar way, CLOOK is similar to CSCAN disk scheduling algorithm. In CLOOK, the disk arm inspite of going to the end goes only to the last request to be serviced in front of the head and then from there goes to the other end’s last request. Thus, it also prevents the extra delay which occurred due to unnecessary traversal to the end of the disk**.**

**Selecting a Disk-Scheduling Algorithm:**

\*SSTF is common and has a natural appeal

\*SCAN and C-SCAN perform better for systems that place a heavy load on the disk.

\*Performance depends on the number and types of requests.

\*Requests for disk service can be influenced by the file-allocation method.

\*The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary.

\*Either SSTF or LOOK is a reasonable choice for the default algorithm.

----------------------------------------------------------------------------------------------------

**WRITE-THROUGH AND WRITE-BACK:-**

In a **write-through** cache, every write to the cache causes a synchronous write to the backing store.

In a **write-back** (or write-behind) cache, writes are not immediately mirrored to the backing store. Instead, the cache tracks which of its locations have been written over and marks these locations as dirty. The data in these locations is written back to the backing store when those data are evicted from the cache, an effect referred to as a lazy write.

**SERVER AVAILABILITY(**MTTF, MMTR and MTBF):-

MTTF is mean time to failure

MTTR is mean time to repair

1/MTTF is failure rate

MTTBF, the mean time between failures, is MTBF = MTTF + MTTR

**SERVER AVAILABILITY:-**

Availability Measured by (MTTF)/(MTTF + MTTR) = MTTF/MTBF – MTTR is very important A good MTTR requires that we detect quickly the failure

**EXAMPLE:-**

A server crashes on the average once a month. When this happens, it takes 12 hours to reboot it. What is the server availability ?

ANSWER:-

• MTBF = 30 days

• MTTR = 12 hours = ½ day

• MTTF = 29 ½ days

• Availability is 29.5/30 \* 100 =98.3 %

**HIT RATIO:-**

Hit ratio = Total number of Hit Counts / Total number of Reference Counts

To represent it as a percentage:

Hit % = Hit ratio \* 100

**EFFECTIVE ACCESS TIME:-**

access time = time for associative memory \* hit ratio + time for main memory (1-hit ratio)

---------------------------------------------------------------------------------

Let P be the page fault rate

Page Fault Rate 0 ≤p≤ 1.0

if p = 0 no page faults; if p = 1, every reference is a fault

Effective Memory Access Time = p \* (page fault service time) +

(1 - p) \* (Memory access time)

---------------------------------------------------------------------------------

p is the probability that the page fault occurs,

hence probability that page is found in Main Memory is (1-p).

Hence,

Effective Memory Access Time = (probability that pagefault occurs) \* (page fault service time) + (probability of no page fault) \* (Memory access time)

Effective Memory Access Time = p \* (page fault service time) + (1 - p) \* (Memory access time)

**Unix Inodes:-**

- Unix uses an indexed allocation structure

– An inode (index node) stores both metadata and the pointers to disk blocks

- Each inode contains 15 block pointers

– First 12 are direct blocks(e.g., 4 KB disk blocks)

– Then single, double, triple indirect blocks

**THREE STANDARD FILES:-**

One reason why UNIX is so flexible is that each program is automatically assigned three standard files: the standard input file, the standard output file, and the standard error file

**Different Types of Unix files**

**Regular files**

Regular files hold data and executable programs. Executable programs are the commands (ls) that you enter on the prompt. The data can be anything and there is no specific format enforced in the way the data is stored. The regular files can be visualized as the leaves in the UNIX tree.

**Directories**

The kernel alone can write the directory file. When a file is added to or deleted from this directory, the kernel makes an entry. A directory file can be visualized as the branch of the UNIX tree.

**Special Or Device Files(Block Special files)**

These files represent the physical devices. Files can also refer to computer hardware such as terminals and printers. These device files can also refer to tape and disk drives, CD-ROM players, modems, network interfaces, scanners, and any other piece of computer hardware. When a process writes to a special file, the data is sent to the physical device associated with it. Special files are not literally files, but are pointers that point to the device drivers located in the kernel. The protection applicable to files is also applicable to physical devices.

**Each filesystem contains:**

1.**A boot block** located in the first few sectors of a file system. The boot block contains the initial bootstrap program used to load the operating system.

Typically, the first sector contains a bootstrap program that reads in a larger bootstrap program from the next few sectors, and so forth.

2.A **super block** describes the state of the file system: the total size of the partition, the block size, pointers to a list of free blocks, the inode number of the root directory, magic number, etc.

3. A linear array of **inodes** (short for “index nodes”). There is a one to one mapping of files to inodes and vice versa.

An i-node contains accounting information as well as enough information to locate all the disk blocks that holds the file’s data.

An **inode** is the “handle” to a file and contains the following information:

* file ownership indication
* file type (e.g., regular, directory, special device, pipes, etc.)
* file access permissions. May have setuid (sticky) bit set.
* time of last access, and modification
* number of links (aliases) to the file
* pointers to the data blocks for the file
* size of the file in bytes (for regular files), major and minor device numbers for special devices.

4. **data blocks** blocks containing the actual contents of files

**Fork command :-**

How could I calculate the number of processes generated using N fork() statements?

e.g. if N=3

main()

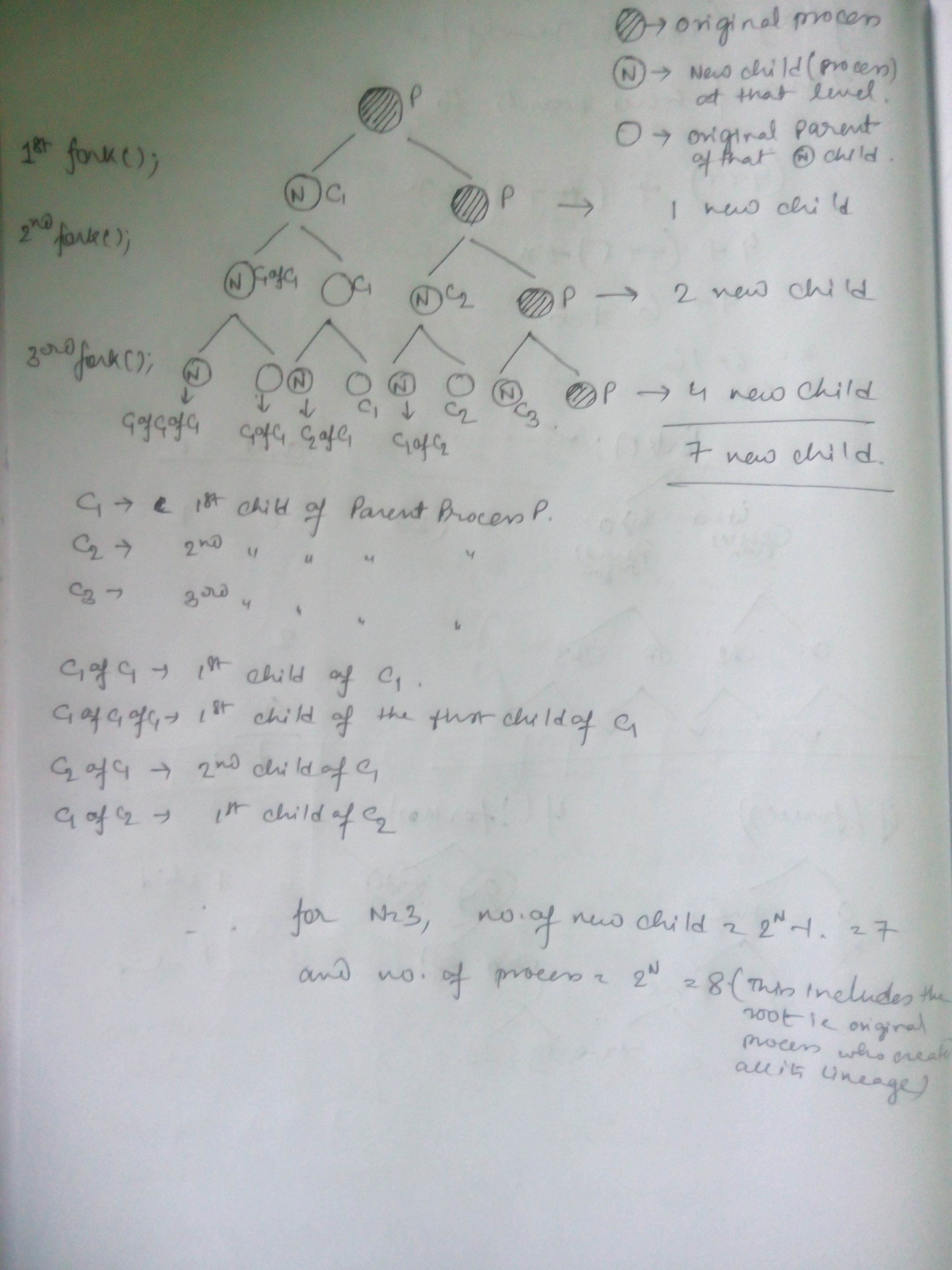
{ fork(); fork(); fork(); }

If you just have a series of N fork statements one after the other, then the total number of processes formed are 2^N

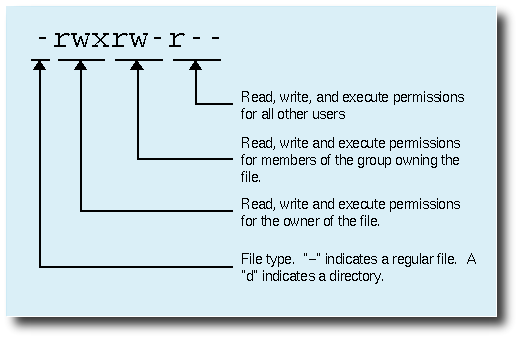
In general,

f(n) = 2\*f(n-1),n>1

= 1,n=1



**CHMOD :-**



It is easy to think of the permission settings as a series of bits (which is how the computer thinks about them). Here's how it works:

rwx rwx rwx = 111 111 111

rw- rw- rw- = 110 110 110

rwx --- --- = 111 000 000

and so on...

rwx = 111 in binary = 7

rw- = 110 in binary= 6

r-x = 101 in binary = 5

r-- = 100 in binary = 4

**Nice command:-**

You can launch a programe with your required priority.

You can also see the process priority(nice value) using "top" command

nice -10 <command name> Will set a process with the priority of "10".

Process priority values range from -20 to 19.

A process with the nice value of -20 is considered to be on top of the priority.

And a process with nice value of 19 is considered to be low on the priority list.

**Renice command :-**

you can also change the priority of an already running process.

**wait() and waitpid() system call:-**

The wait() system call suspends execution of the current process until one of its children terminates.

The waitpid() system call suspends execution of the current process until a child specified by pid argument has changed state

**brk() system call:-**

The amount of space allocated for the calling process. e brk() and sbrk() functions are used to change

**lseek system call:-**

The lseek( ) system call is provided to set the file position or repositon of a file descriptor to a given value.

**exec command:-**

The exec command replaces the current shell process with the specified command. Normally, when you run a command a new process is spawned (forked). The exec command does not spawn a new process. Instead, the current process is overlaid with the new command. In other words the exec command is executed in place of the current shell without creating a new process.

**JOIN command :-**

It is for joining lines of two files on a common field. It can be used to join two files by selecting fields within the line and joining the files on them. The result is written to standard output.

**SECTION IN LINUX :-**

(.bss) => segment for uninitialized data.

(.data) => segment for initialized data here by default.

(.rodata) => segment for constant data,

(.text) => machine instruction code

**DIFFERENT TYPE OF PROCESS:-**

**Zombie Process:-**

A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process. A child process always first becomes a zombie before being removed from the process table. The parent process reads the exit status of the child process which reaps off the child process entry from the process table.

**Orphan Process:-**

A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process. It is achieved by nohup command

**Daemon Process:-**

In Unix and other multitasking computer operating systems, a daemon is a computer program that runs as a background process, rather than being under the direct control of an interactive user.

**Child process:-**

It is a process created by another process (the parent process). For eg Fork command

**Sleeping process :-**

A process enters a Sleeping state when it needs resources that are not currently available. At that point, it either goes voluntarily into Sleep state or the kernel puts it into Sleep state.

**There are two tasks with specially distinguished process IDs:**

**Process ID 0 :-**

swapper or sched has process ID 0 and is responsible for paging, and is actually part of the kernel rather than a normal user-mode process.

**Process ID 1 :-**

Process ID 1 is usually the init process primarily responsible for starting and shutting down the system. Process ID 1 was not specifically reserved for init by any technical measures.

**The Unix "parameters", $0, $#, $1, $2, $3, ..., $?, $@**

**$0 => Command-name**

$ cat test.sh

echo "This is : "$0

This is : test.sh

**$# => contains the number of parameters for this shell is started**

$ ./test2.sh a b

The number of parameters : 2

**$1 $2 ... => parameter 1 parameter 2 ...**

$? => check the return code

$ date

Thu Apr 15 17:12:05 DFT 1999

$ echo $?

0

**$@ => includes all the parameters that this shell is started**

$ Makenewusers piet /home/piet

$echo $@

piet /home/piet

**The /dev directory :-**

The /dev directory contains the special device files for all the devices.

The /dev directory contains special device files that control access to peripheral devices.

There are several subdirectories to the /dev directory. Each of these subdirectories holds special device files related to a certain type of device. For example, the /dev/dsk directory contains device files for floppy and hard disks.

**ASSEMBLER**

**Introduction**

- Convert mnemonic operation codes to their machine language equivalents

- Convert symbolic operands to their equivalent machine addresses

- Build the machine instructions in the proper format

- Convert the data constants to internal machine representations

- Write the object program and the assembly listing

**Pass 1:**

* Separate the Symbol, Mnemonic opcode, and operand fields
* Build the symbol table
* Determine the storage-required for every assembly language statement and update the location counter
* Construct Intermediate Representation

**Pass 2:**

* Synthesize the target program

**Data Structures Used In Pass I**

* OPTAB
* SYMTAB
* LITTAB
* POOLTAB