# UNIX Concepts

SECTION - I

FILE MANAGEMENT IN UNIX

1. *How are devices represented in UNIX?*

All devices are represented by files called *special files* that are located in/dev directory. Thus, device files and other files are named and accessed in the same way. A 'regular file' is just an ordinary data file in the disk. A 'block special file' represents a device with characteristics similar to a disk (data transfer in terms of blocks). A 'character special file' represents a device with characteristics similar to a keyboard (data transfer is by stream of bits in sequential order).

2. *What is 'inode'?*

All UNIX files have its description stored in a structure called 'inode'. The inode contains info about the file-size, its location, time of last access, time of last modification, permission and so on. Directories are also represented as files and have an associated inode. In addition to descriptions about the file, the inode contains pointers to the data blocks of the file. If the file is large, inode has indirect pointer to a block of pointers to additional data blocks (this further aggregates for larger files). A block is typically 8k.

Inode consists of the following fields:

- File owner identifier
- File type
- File access permissions
- File access times
- Number of links
- File size
- Location of the file data

3. *Brief about the directory representation in UNIX*

A Unix directory is a file containing a correspondence between filenames and inodes. A directory is a special file that the kernel maintains. Only kernel modifies directories, but processes can read directories. The contents of a directory are a list of filename and inode number pairs. When new directories are created, kernel makes two entries named '.' (refers to the directory itself) and '..' (refers to parent directory). System call for creating directory is mkdir (pathname, mode).

4. *What are the Unix system calls for I/O?*

- open(pathname,flag,mode) - open file
- creat(pathname,mode) - create file
- close(filedes) - close an open file
- read(filedes,buffer,bytes) - read data from an open file
- write(filedes,buffer,bytes) - write data to an open file
- lseek(filedes,offset,from) - position an open file
- dup(filedes) - duplicate an existing file descriptor

- dup2(oldfd,newfd) - duplicate to a desired file descriptor
- fcntl(filedes,cmd,arg) - change properties of an open file
- ioctl(filedes,request,arg) - change the behaviour of an open file

The difference between fcntl anf ioctl is that the former is intended for any open file, while the latter is for device-specific operations.

5. *How do you change File Access Permissions?*

Every file has following attributes:

- owner's user ID ( 16 bit integer )
- owner's group ID ( 16 bit integer )
- File access mode word

'r w x -r w x- r w x'
(user permission-group permission-others permission)
r-read, w-write, x-execute
To change the access mode, we use chmod(filename,mode).
Example 1:

To change mode of myfile to 'rw-rw-r--' (ie. read, write permission for user - read,write permission for group - only read permission for others) we give the args as:

chmod(myfile,0664) .

Each operation is represented by discrete values

'r' is 4
'w' is 2
'x' is 1

Therefore, for 'rw' the value is 6(4+2).
Example 2:

To change mode of myfile to 'rwxr--r--' we give the args as:
chmod(myfile,0744).

6. *What are links and symbolic links in UNIX file system?*

A link is a second name (not a file) for a file. Links can be used to assign more than one name to a file, but cannot be used to assign a directory more than one name or link filenames on different computers.

Symbolic link 'is' a file that only contains the name of another file.Operation on the symbolic link is directed to the file pointed by the it.Both the limitations of links are eliminated in symbolic links.

Commands for linking files are:

| | |
|---|---|
| Link | ln filename1 filename2 |
| Symbolic link | ln -s filename1 filename2 |

7. *What is a FIFO?*

FIFO are otherwise called as 'named pipes'. FIFO (first-in-first-out) is a special file which is said to be data transient. Once data is read from named pipe, it cannot be read again. Also, data can be read only in the order written. It is used in interprocess communication where a process writes to one end of the pipe (producer) and the other reads from the other end (consumer).

*8. How do you create special files like named pipes and device files?*

The system call mknod creates special files in the following sequence.

1. kernel assigns new inode,
2. sets the file type to indicate that the file is a pipe, directory or special file,
3. If it is a device file, it makes the other entries like major, minor device numbers.

For example:

If the device is a disk, major device number refers to the disk controller and minor device number is the disk.

*9. Discuss the mount and unmount system calls*

The privileged mount system call is used to attach a file system to a directory of another file system; the unmount system call detaches a file system. When you mount another file system on to your directory, you are essentially splicing one directory tree onto a branch in another directory tree. The first argument to mount call is the mount point, that is , a directory in the current file naming system. The second argument is the file system to mount to that point. When you insert a cdrom to your unix system's drive, the file system in the cdrom automatically mounts to /dev/cdrom in your system.

*10. How does the inode map to data block of a file?*

Inode has 13 block addresses. The first 10 are direct block addresses of the first 10 data blocks in the file. The 11th address points to a one-level index block. The 12th address points to a two-level (double in-direction) index block. The 13th address points to a three-level(triple in-direction)index block. This provides a very large maximum file size with efficient access to large files, but also small files are accessed directly in one disk read.

*11. What is a shell?*

A shell is an interactive user interface to an operating system services that allows an user to enter commands as character strings or through a graphical user interface. The shell converts them to system calls to the OS or forks off a process to execute the command. System call results and other information from the OS are presented to the user through an interactive interface. Commonly used shells are sh,csh,ks etc.

SECTION - II

PROCESS MODEL and IPC

*1. Brief about the initial process sequence while the system boots up.*

While booting, special process called the 'swapper' or 'scheduler' is created with Process-ID 0. The swapper manages memory allocation for processes and influences CPU allocation. The swapper inturn creates 3 children:

- the process dispatcher,
- vhand and
- dbflush

with IDs 1,2 and 3 respectively.

This is done by executing the file /etc/init. Process dispatcher gives birth to the shell. Unix keeps track of all the processes in an internal data structure called the Process Table (listing command is ps -el).

*2. What are various IDs associated with a process?*

Unix identifies each process with a unique integer called ProcessID. The process that executes the request for creation of a process is called the 'parent process' whose PID is 'Parent Process ID'. Every process is associated with a particular user called the 'owner' who has privileges over the process. The identification for the user is 'UserID'. Owner is the user who executes the process. Process also has 'Effective User ID' which determines the access privileges for accessing resources like files.

getpid() -process id
getppid() -parent process id
getuid() -user id
geteuid() -effective user id

*3. Explain fork() system call.*

The `fork()' used to create a new process from an existing process. The new process is called the child process, and the existing process is called the parent. We can tell which is which by checking the return value from `fork()'. The parent gets the child's pid returned to him, but the child gets 0 returned to him.

*4. Predict the output of the following program code*

```
main()
{
        fork();
        printf("Hello World!");
}
```

***Answer:***

Hello World!Hello World!

***Explanation***:

The fork creates a child that is a duplicate of the parent process. The child begins from the fork().All the statements after the call to fork() will be executed twice.(once by the parent process and other by child). The statement before fork() is executed only by the parent process.

*5. Predict the output of the following program code*

```
main()
{
        fork(); fork(); fork();
        printf("Hello World!");
}
```

***Answer:***

"Hello World" will be printed 8 times.

***Explanation***:

2^n times where n is the number of calls to fork()

*6. List the system calls used for process management:*

| System calls | Description |
|---|---|
| fork() | To create a new process |
| exec() | To execute a new program in a process |
| wait() | To wait until a created process completes its execution |
| exit() | To exit from a process execution |
| getpid() | To get a process identifier of the current process |
| getppid() | To get parent process identifier |
| nice() | To bias the existing priority of a process |
| brk() | To increase/decrease the data segment size of a process |

*7. How can you get/set an environment variable from a program?*

Getting the value of an environment variable is done by using `getenv()'.
Setting the value of an environment variable is done by using `putenv()'.

*8. How can a parent and child process communicate?*

A parent and child can communicate through any of the normal inter-process communication schemes (pipes, sockets, message queues, shared memory), but also have some special ways to communicate that take advantage of their relationship as a parent and child. One of the most obvious is that the parent can get the exit status of the child.

*9. What is a zombie?*

When a program forks and the child finishes before the parent, the kernel still keeps some of its information about the child in case the parent might need it - for example, the parent may need to check the child's exit status. To be able to get this information, the parent calls `wait()'; In the interval between the child terminating and the parent calling `wait()', the child is said to be a `zombie' (If you do `ps', the child will have a `Z' in its status field to indicate this.)

*10. What are the process states in Unix?*

As a process executes it changes state according to its circumstances. Unix processes have the following states:

Running : The process is either running or it is ready to run .
Waiting : The process is waiting for an event or for a resource.
Stopped : The process has been stopped, usually by receiving a signal.
Zombie : The process is dead but have not been removed from the process table.

*11. What Happens when you execute a program?*

When you execute a program on your UNIX system, the system creates a special environment for that program. This environment contains everything needed for the system to run the program as if no other program were running on the system. Each process has process context, which is everything that is unique about the state of the program you are currently running. Every time you execute a program the UNIX

system does a fork, which performs a series of operations to create a process context and then execute your program in that context. The steps include the following:

- Allocate a slot in the process table, a list of currently running programs kept by UNIX.
- Assign a unique process identifier (PID) to the process.
- iCopy the context of the parent, the process that requested the spawning of the new process.
- Return the new PID to the parent process. This enables the parent process to examine or control the process directly.

After the fork is complete, UNIX runs your program.

*12. What Happens when you execute a command?*

When you enter 'ls' command to look at the contents of your current working directory, UNIX does a series of things to create an environment for ls and the run it: The shell has UNIX perform a fork. This creates a new process that the shell will use to run the ls program. The shell has UNIX perform an exec of the ls program. This replaces the shell program and data with the program and data for ls and then starts running that new program. The ls program is loaded into the new process context, replacing the text and data of the shell. The ls program performs its task, listing the contents of the current directory.

*13. What is a Daemon?*

A daemon is a process that detaches itself from the terminal and runs, disconnected, in the background, waiting for requests and responding to them. It can also be defined as the background process that does not belong to a terminal session. Many system functions are commonly performed by daemons, including the sendmail daemon, which handles mail, and the NNTP daemon, which handles USENET news. Many other daemons may exist. Some of the most common daemons are:

- init: Takes over the basic running of the system when the kernel has finished the boot process.
- inetd: Responsible for starting network services that do not have their own stand-alone daemons. For example, inetd usually takes care of incoming rlogin, telnet, and ftp connections.
- cron: Responsible for running repetitive tasks on a regular schedule.

*14. What is 'ps' command for?*

The ps command prints the process status for some or all of the running processes. The information given are the process identification number (PID),the amount of time that the process has taken to execute so far etc.

*15. How would you kill a process?*

The kill command takes the PID as one argument; this identifies which process to terminate. The PID of a process can be got using 'ps' command.

*16. What is an advantage of executing a process in background?*

The most common reason to put a process in the background is to allow you to

do something else interactively without waiting for the process to complete. At the end of the command you add the special background symbol, &. This symbol tells your shell to execute the given command in the background.

Example: cp *.* ../backup& (cp is for copy)

*17. How do you execute one program from within another?*

The system calls used for low-level process creation are execlp() and execvp(). The execlp call overlays the existing program with the new one , runs that and exits. The original program gets back control only when an error occurs.

execlp(path,file_name,arguments..); //last argument must be NULL

A variant of execlp called execvp is used when the number of arguments is not known in advance.

execvp(path,argument_array); //argument array should be terminated by NULL

*18. What is IPC? What are the various schemes available?*

The term IPC (Inter-Process Communication) describes various ways by which different process running on some operating system communicate between each other. Various schemes available are as follows:

*Pipes:*

One-way communication scheme through which different process can communicate. The problem is that the two processes should have a common ancestor (parent-child relationship). However this problem was fixed with the introduction of named-pipes (FIFO).

*Message Queues :*

Message queues can be used between related and unrelated processes running on a machine.

*Shared Memory:*

This is the fastest of all IPC schemes. The memory to be shared is mapped into the address space of the processes (that are sharing). The speed achieved is attributed to the fact that there is no kernel involvement. But this scheme needs synchronization.

Various forms of synchronisation are mutexes, condition-variables, read-write locks, record-locks, and semaphores.

SECTION - III

MEMORY MANAGEMENT

*1. What is the difference between Swapping and Paging?*

*Swapping:*

Whole process is moved from the swap device to the main memory for execution. Process size must be less than or equal to the available main memory. It is easier to implementation and overhead to the system. Swapping systems does not handle the memory more flexibly as compared to the paging systems.

*Paging:*

Only the required memory pages are moved to main memory from the swap device for execution. Process size does not matter. Gives the concept of the virtual memory.

It provides greater flexibility in mapping the virtual address space into the physical memory of the machine. Allows more number of processes to fit in the main memory simultaneously. Allows the greater process size than the available physical memory. Demand paging systems handle the memory more flexibly.

2. *What is major difference between the Historic Unix and the new BSD release of Unix System V in terms of Memory Management?*

Historic Unix uses Swapping – entire process is transferred to the main memory from the swap device, whereas the Unix System V uses Demand Paging – only the part of the process is moved to the main memory. Historic Unix uses one Swap Device and Unix System V allow multiple Swap Devices.

3. *What is the main goal of the Memory Management?*

- It decides which process should reside in the main memory,
- Manages the parts of the virtual address space of a process which is non-core resident,
- Monitors the available main memory and periodically write the processes into the swap device to provide more processes fit in the main memory simultaneously.

4. *What is a Map?*

A Map is an Array, which contains the addresses of the free space in the swap device that are allocatable resources, and the number of the resource units available there.

| Address | Units |
|---|---|
| 1 | 10,000 |

This allows First-Fit allocation of contiguous blocks of a resource. Initially the Map contains one entry – address (block offset from the starting of the swap area) and the total number of resources.

Kernel treats each unit of Map as a group of disk blocks. On the allocation and freeing of the resources Kernel updates the Map for accurate information.

5. *What scheme does the Kernel in Unix System V follow while choosing a swap device among the multiple swap devices?*

Kernel follows Round Robin scheme choosing a swap device among the multiple swap devices in Unix System V.

*6. What is a Region?*

A Region is a continuous area of a process's address space (such as text, data and stack). The kernel in a 'Region Table' that is local to the process maintains region. Regions are sharable among the process.

*7. What are the events done by the Kernel after a process is being swapped out from the main memory?*

When Kernel swaps the process out of the primary memory, it performs the following:

- Kernel decrements the Reference Count of each region of the process. If the reference count becomes zero, swaps the region out of the main memory,
- Kernel allocates the space for the swapping process in the swap device,
- Kernel locks the other swapping process while the current swapping operation is going on,
- The Kernel saves the swap address of the region in the region table.

*8. Is the Process before and after the swap are the same? Give reason.*

Process before swapping is residing in the primary memory in its original form. The regions (text, data and stack) may not be occupied fully by the process, there may be few empty slots in any of the regions and while swapping Kernel do not bother about the empty slots while swapping the process out.

After swapping the process resides in the swap (secondary memory) device. The regions swapped out will be present but only the occupied region slots but not the empty slots that were present before assigning.

While swapping the process once again into the main memory, the Kernel referring to the Process Memory Map, it assigns the main memory accordingly taking care of the empty slots in the regions.

*9. What do you mean by u-area (user area) or u-block?*

This contains the private data that is manipulated only by the Kernel. This is local to the Process, i.e. each process is allocated a u-area.

*10. What are the entities that are swapped out of the main memory while swapping the process out of the main memory?*

All memory space occupied by the process, process's u-area, and Kernel stack are swapped out, theoretically.

Practically, if the process's u-area contains the Address Translation Tables for the process then Kernel implementations do not swap the u-area.

*11. What is Fork swap?*

fork() is a system call to create a child process. When the parent process calls fork() system call, the child process is created and if there is short of memory then the child process is sent to the read-to-run state in the swap device, and return to the user state without swapping the parent process. When the memory will be available the child process will be swapped into the main memory.

*12. What is Expansion swap?*

At the time when any process requires more memory than it is currently allocated, the Kernel performs Expansion swap. To do this Kernel reserves enough space in the swap device. Then the address translation mapping is adjusted for the new virtual address space but the physical memory is not allocated. At last Kernel swaps the process into the assigned space in the swap device. Later when the Kernel swaps the process into the main memory this assigns memory according to the new address translation mapping.

*13. How the Swapper works?*

The swapper is the only process that swaps the processes. The Swapper operates only in the Kernel mode and it does not uses System calls instead it uses internal Kernel functions for swapping. It is the archetype of all kernel process.

*14. What are the processes that are not bothered by the swapper? Give Reason.*

- Zombie process: They do not take any up physical memory.
- Processes locked in memories that are updating the region of the process.
- Kernel swaps only the sleeping processes rather than the 'ready-to-run' processes, as they have the higher probability of being scheduled than the Sleeping processes.

*15. What are the requirements for a swapper to work?*

The swapper works on the highest scheduling priority. Firstly it will look for any sleeping process, if not found then it will look for the ready-to-run process for swapping. But the major requirement for the swapper to work the ready-to-run process must be core-resident for at least 2 seconds before swapping out. And for swapping in the process must have been resided in the swap device for at least 2 seconds. If the requirement is not satisfied then the swapper will go into the wait state on that event and it is awaken once in a second by the Kernel.

*16. What are the criteria for choosing a process for swapping into memory from the swap device?*

The resident time of the processes in the swap device, the priority of the processes and the amount of time the processes had been swapped out.

*17. What are the criteria for choosing a process for swapping out of the memory to the swap device?*

- The process's memory resident time,
- Priority of the process and
- The nice value.

*18. What do you mean by nice value?*

Nice value is the value that controls {increments or decrements} the priority of the process. This value that is returned by the nice () system call. The equation for using nice value is:

*Priority = ("recent CPU usage"/constant) + (base- priority) + (nice value)*

Only the administrator can supply the nice value. The nice () system call works for the running process only. Nice value of one process cannot affect the nice value of the other process.

*19. What are conditions on which deadlock can occur while swapping the processes?*

- All processes in the main memory are asleep.
- All 'ready-to-run' processes are swapped out.
- There is no space in the swap device for the new incoming process that are swapped out of the main memory.
- There is no space in the main memory for the new incoming process.

*20. What are conditions for a machine to support Demand Paging?*

- Memory architecture must based on Pages,
- The machine must support the 'restartable' instructions.

*21. What is 'the principle of locality'?*

It's the nature of the processes that they refer only to the small subset of the total data space of the process. i.e. the process frequently calls the same subroutines or executes the loop instructions.

*22. What is the working set of a process?*

The set of pages that are referred by the process in the last 'n', references, where 'n' is called the *window* of the working set of the process.

*23. What is the window of the working set of a process?*

The window of the working set of a process is the total number in which the process had referred the set of pages in the working set of the process.

*24. What is called a page fault?*

Page fault is referred to the situation when the process addresses a page in the working set of the process but the process fails to locate the page in the working set. And on a page fault the kernel updates the working set by reading the page from the secondary device.

*25. What are data structures that are used for Demand Paging?*

Kernel contains 4 data structures for Demand paging. They are,

- Page table entries,
- Disk block descriptors,
- Page frame data table (pfdata),
- Swap-use table.

*26. What are the bits that support the demand paging?*

Valid, Reference, Modify, Copy on write, Age. These bits are the part of the page table entry, which includes physical address of the page and protection bits.

| Page address | Age | Copy on write | Modify | Reference | Valid | Protection |
|---|---|---|---|---|---|---|

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | | |

*27. How the Kernel handles the fork() system call in traditional Unix and in the System V Unix, while swapping?*

Kernel in traditional Unix, makes the duplicate copy of the parent's address space and attaches it to the child's process, while swapping. Kernel in System V Unix, manipulates the region tables, page table, and pfdata table entries, by incrementing the reference count of the region table of shared regions.

*28. Difference between the fork() and vfork() system call?*

During the fork() system call the Kernel makes a copy of the parent process's address space and attaches it to the child process.

But the vfork() system call do not makes any copy of the parent's address space, so it is faster than the fork() system call. The child process as a result of the vfork() system call executes exec() system call. The child process from vfork() system call executes in the parent's address space (this can overwrite the parent's data and stack ) which suspends the parent process until the child process exits.

*29. What is BSS(Block Started by Symbol)?*

A data representation at the machine level, that has initial values when a program starts and tells about how much space the kernel allocates for the un-initialized data. Kernel initializes it to zero at run-time.

*30. What is Page-Stealer process?*

This is the Kernel process that makes rooms for the incoming pages, by swapping the memory pages that are not the part of the working set of a process. Page-Stealer is created by the Kernel at the system initialization and invokes it throughout the lifetime of the system. Kernel locks a region when a process faults on a page in the region, so that page stealer cannot steal the page, which is being faulted in.

*31. Name two paging states for a page in memory?*

The two paging states are:

- The page is aging and is not yet eligible for swapping,
- The page is eligible for swapping but not yet eligible for reassignment to other virtual address space.

*32. What are the phases of swapping a page from the memory?*

- Page stealer finds the page eligible for swapping and places the page number in the list of pages to be swapped.
- Kernel copies the page to a swap device when necessary and clears the *valid* bit in the page table entry, decrements the pfdata reference count, and places the pfdata table entry at the end of the free list if its reference count is 0.

*33. What is page fault? Its types?*

Page fault refers to the situation of not having a page in the main memory when any process references it.

There are two types of page fault :

- Validity fault,
- Protection fault.

*34. In what way the Fault Handlers and the Interrupt handlers are different?*

Fault handlers are also an interrupt handler with an exception that the interrupt handlers cannot sleep. Fault handlers sleep in the context of the process that caused the memory fault. The fault refers to the running process and no arbitrary processes are put to sleep.

*35. What is validity fault?*

If a process referring a page in the main memory whose valid bit is not set, it results in validity fault.

The valid bit is not set for those pages:

- that are outside the virtual address space of a process,
- that are the part of the virtual address space of the process but no physical address is assigned to it.

*36. What does the swapping system do if it identifies the illegal page for swapping?*

If the disk block descriptor does not contain any record of the faulted page, then this causes the attempted memory reference is invalid and the kernel sends a *"Segmentation violation"* signal to the offending process. This happens when the swapping system identifies any invalid memory reference.

*37. What are states that the page can be in, after causing a page fault?*

- On a swap device and not in memory,
- On the free page list in the main memory,
- In an executable file,
- Marked "demand zero",
- Marked "demand fill".

38. *In what way the validity fault handler concludes?*

- It sets the valid bit of the page by clearing the modify bit.
- It recalculates the process priority.

39. *At what mode the fault handler executes?*

At the Kernel Mode.

40. *What do you mean by the protection fault?*

Protection fault refers to the process accessing the pages, which do not have the access permission. A process also incur the protection fault when it attempts to write a page whose *copy on write* bit was set during the fork() system call.

41. *How the Kernel handles the copy on write bit of a page, when the bit is set?*

In situations like, where the copy on write bit of a page is set and that page is shared by more than one process, the Kernel allocates new page and copies the content to the new page and the other processes retain their references to the old page. After copying the Kernel updates the page table entry with the new page number. Then Kernel decrements the reference count of the old pfdata table entry.

In cases like, where the copy on write bit is set and no processes are sharing the page, the Kernel allows the physical page to be reused by the processes. By doing so, it clears the copy on write bit and disassociates the page from its disk copy (if one exists), because other process may share the disk copy. Then it removes the pfdata table entry from the page-queue as the new copy of the virtual page is not on the swap device. It decrements the swap-use count for the page and if count drops to 0, frees the swap space.

42. *For which kind of fault the page is checked first?*

The page is first checked for the validity fault, as soon as it is found that the page is invalid (valid bit is clear), the validity fault handler returns immediately, and the process incur the validity page fault. Kernel handles the validity fault and the process will incur the protection fault if any one is present.

43. *In what way the protection fault handler concludes?*

After finishing the execution of the fault handler, it sets the *modify* and *protection* bits and clears the *copy on write* bit. It recalculates the process-priority and checks for signals.

44. *How the Kernel handles both the page stealer and the fault handler?*

The page stealer and the fault handler thrash because of the shortage of the memory. If the sum of the working sets of all processes is greater that the physical memory then the fault handler will usually sleep because it cannot allocate pages for a process. This results in the reduction of the system throughput because Kernel spends too much time in overhead, rearranging the memory in the frantic pace.