Unit V

Algorithm for booting the UNIX system :

As we've noted, the boot process begins when the instructions stored in the computer's permanent, nonvolatile memory (referred to colloquially as the BIOS, ROM,NVRAM, and so on) are executed. This storage location for the initial boot instructions is generically referred to as *firmware* (in contrast to "software," but reflecting the fact that the instructions constitute a program[2]).

These instructions are executed automatically when the power is turned on or the system is reset, although the exact sequence of events may vary according to the values of stored parameters.[3] The firmware instructions may also begin executing in response to a command entered on the system console (as we'll see in a bit). However they are initiated, these instructions are used to locate and start up the system's *boot program* , which in turn starts the Unix operating system.

The boot program is stored in a standard location on a bootable device. For a normal boot from disk, for example, the boot program might be located in block 0 of the root disk or, less commonly, in a special partition on the root disk. In the same way, the boot program may be the second file on a bootable tape or in a designated location on a remote file server in the case of a network boot of a diskless workstation.

There is usually more than one bootable device on a system. The firmware program may include logic for selecting the device to boot from, often in the form of a list of potential devices to examine. In the absence of other instructions, the first bootable device that is found is usually the one that is used. Some systems allow for several variations on this theme. For example, the RS/6000 NVRAM contains separate default device search lists for normal and service boots; it also allows the system administrator to add customized search lists for either or both boot types using the bootlist command.

The boot program is responsible for loading the Unix kernel into memory and passing control of the system to it. Some systems have two or more levels of intermediate boot programs between the firmware instructions and the independently-executing Unix kernel. Other systems use different boot programs depending on the type of boot.

Even PC systems follow this same basic procedure. When the power comes on or the system is reset, the BIOS starts the master boot program, located in the first 512 bytes of the system disk. This program then typically loads the boot program located in the first 512 bytes of the active partition on that disk, which then loads the kernel. Sometimes, the master boot program loads the kernel itself. The boot process from other media is similar.

The firmware program is basically just smart enough to figure out if the hardware devices it needs are accessible (e.g., can it find the system disk or the network) and to load and initiate the boot program. This first-stage boot program often performs additional hardware status verification, checking for the presence of expected system memory and major peripheral

devices. Some systems do much more elaborate hardware checks, verifying the status of virtually every device and detecting new ones added since the last boot.

Algorithm for init process :

In Unix-based computer operating systems , **init** (short for *initialization*) is the first process started during booting of the computer system. Init is a daemon process that continues running until the system is shut down. It is the direct or indirect ancestor of all other processes and automatically adopts all orphaned processes. Init is started by the kernel during the booting process; a kernel panic will occur if the kernel is unable to start it. Init is typically assigned process identifier 1.

In Unix systems such as System III and System V, the design of init has diverged from the functionality provided by the init in Research Unix and its BSD derivatives. Up until recently, most Linux distributions employed a traditional init that is somewhat compatible with System V, while some distributions such as Slackware use BSD-style startup scripts, and others such as Gentoo have their own customized versions.

Since then, several additional init implementations have been created, attempting to address design limitations in the traditional versions. These include launchd, the Service Management Facility, systemd, Runit and OpenRC

Process Scheduling algorithms

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms which we are going to discuss in this chapter −

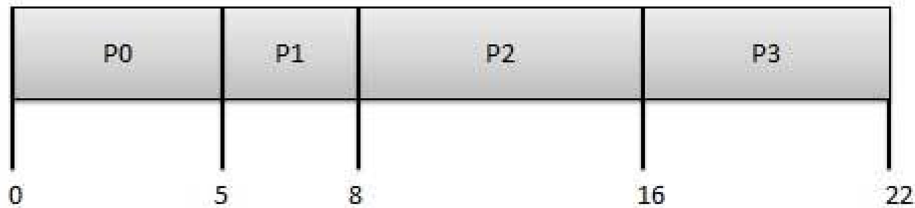
- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-Next (SJN) Scheduling
- Priority Scheduling
- Shortest Remaining Time
- Round Robin(RR) Scheduling
- Multiple-Level Queues Scheduling

These algorithms are either **non-preemptive or preemptive**. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

First Come First Serve (FCFS)

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.

Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16



Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	8 - 2 = 6
P3	16 - 3 = 13

Average Wait Time: (0+4+6+13) / 4 = 5.75

Shortest Job Next (SJN)

- This is also known as **shortest job first**, or SJF
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known.
- The processer should know in advance how much time process will take.

Given: Table of processes, and their Arrival time, Execution time

Waiting time of each process is as follows –

Process	**Waiting Time**
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	14 - 2 = 12
P3	8 - 3 = 5

Average Wait Time: (0 + 4 + 12 + 5)/4 = 21 / 4 = 5.25

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.

Given: Table of processes, and their Arrival time, Execution time, and priority. Here we are considering 1 is the lowest priority.

Process	Arrival Time	Execution Time	Priority	Service Time
P0	0	5	1	0
P1	1	3	2	11
P2	2	8	1	14
P3	3	6	3	5

Waiting time of each process is as follows −

Process	Waiting Time
P0	0 - 0 = 0
P1	11 - 1 = 10
P2	14 - 2 = 12
P3	5 - 3 = 2

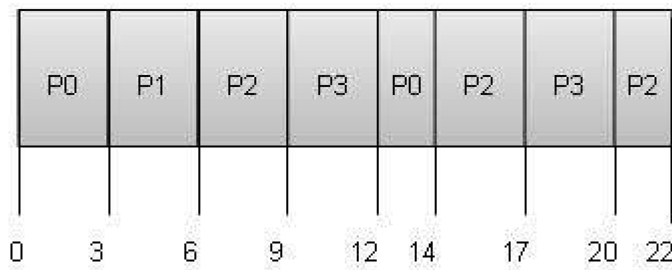
Average Wait Time: (0 + 10 + 12 + 2)/4 = 24 / 4 = 6

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.
- Impossible to implement in interactive systems where required CPU time is not known.
- 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.



Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	(0 - 0) + (12 - 3) = 9
P1	(3 - 1) = 2
P2	(6 - 2) + (14 - 9) + (20 - 17) = 12
P3	(9 - 3) + (17 - 12) = 11

Average Wait Time: (9+2+12+11) / 4 = 8.5

Multiple-Level Queues Scheduling

Multiple-level queues are not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.

- Multiple queues are maintained for processes with common characteristics.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue.

For example, CPU-bound jobs can be scheduled in one queue and all I/O-bound jobs in another queue. The Process Scheduler then alternately selects jobs from each queue and assigns them to the CPU based on the algorithm assigned to the queue.

Unix Architecture

The Unix operating system is a set of programs that act as a link between the computer and the user.

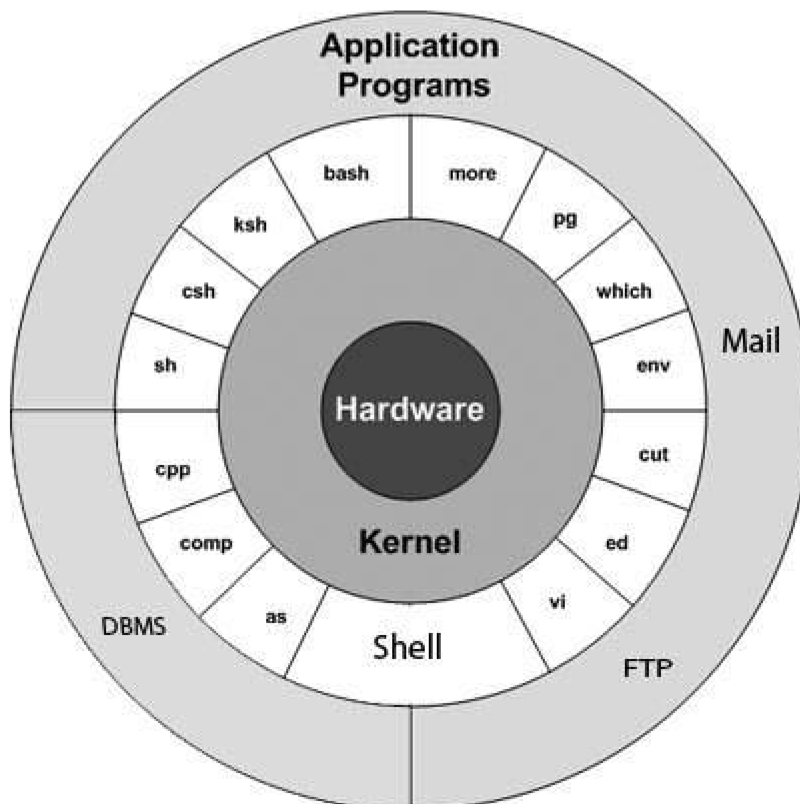
The computer programs that allocate the system resources and coordinate all the details of the computer's internals is called the **operating system** or the **kernel**.

Users communicate with the kernel through a program known as the **shell**. The shell is a command line interpreter; it translates commands entered by the user and converts them into a language that is understood by the kernel.

- Unix was originally developed in 1969 by a group of AT&T employees Ken Thompson, Dennis Ritchie, Douglas McIlroy, and Joe Ossanna at Bell Labs.
- There are various Unix variants available in the market. Solaris Unix, AIX, HP Unix and BSD are a few examples. Linux is also a flavor of Unix which is freely available.
- Several people can use a Unix computer at the same time; hence Unix is called a multiuser system.
- A user can also run multiple programs at the same time; hence Unix is a multitasking environment.

Unix Architecture

Here is a basic block diagram of a Unix system −



The main concept that unites all the versions of Unix is the following four basics −

- **Kernel** − The kernel is the heart of the operating system. It interacts with the hardware and most of the tasks like memory management, task scheduling and file management.
- **Shell** − The shell is the utility that processes your requests. When you type in a command at your terminal, the shell interprets the command and calls the program that you want. The shell uses standard syntax for all commands. C Shell, Bourne Shell and Korn Shell are the most famous shells which are available with most of the Unix variants.
- **Commands and Utilities** − There are various commands and utilities which you can make use of in your day to day activities. **cp**, **mv**, **cat** and **grep**, etc. are few examples of commands and utilities. There are over 250 standard commands plus numerous others provided through 3rd party software. All the commands come along with various options.
- **Files and Directories** − All the data of Unix is organized into files. All files are then organized into directories. These directories are further organized into a tree-like structure called the **filesystem**.

System Bootup

If you have a computer which has the Unix operating system installed in it, then you simply need to turn on the system to make it live.

As soon as you turn on the system, it starts booting up and finally it prompts you to log into the system, which is an activity to log into the system and use it for your day-to-day activities.

Login Unix

When you first connect to a Unix system, you usually see a prompt such as the following −

login:

To log in

- Have your userid (user identification) and password ready. Contact your system administrator if you don't have these yet.
- Type your userid at the login prompt, then press **ENTER**. Your userid is **case-sensitive**, so be sure you type it exactly as your system administrator has instructed.
- Type your password at the password prompt, then press **ENTER**. Your password is also case-sensitive.
- If you provide the correct userid and password, then you will be allowed to enter into the system. Read the information and messages that comes up on the screen, which is as follows

fork(), vfork(), wait() and exec() system calls

It is found that in any Linux/Unix based Operating Systems it is good to understand **fork** and **vfork** system calls, how they behave, how we can use them and differences between them. Along with these **wait** and **exec** system calls are used for process spawning and various other related tasks.

Most of these concepts are explained using programming examples. In this article, I will be covering what are fork, vfork, exec and wait system calls, their distinguishing characters and how they can be better used.

fork()

fork(): System call to create a child process.

shashi@linuxtechi ~}$ man fork

This will yield output mentioning what is fork used for, syntax and along with all the required details.

The syntax used for the fork system call is as below,

pid_t fork(void);

Fork system call creates a child that differs from its parent process only in **pid(process ID)** and **ppid(parent process ID)**. Resource utilization is set to zero. File locks and pending signals are not inherited. (In Linux "fork" is implemented as "**copy-on-write()**").

Note:- "**Copy on write**" -> Whenever a fork() system call is called, a copy of all the pages(memory) related to the parent process is created and loaded into a separate memory location by the Operating System for the child process. But this is not needed in all cases and may be required only when some process writes to this address space or memory area, then only separate copy is created/provided.

Return values:- PID (process ID) of the child process is returned in parents thread of execution and "**zero**" is returned in child's thread of execution. Following is the c-programming example which explains how fork system call works.

shashi@linuxtechi ~}$ vim 1_fork.c

```
#include<stdio.h>

#include<unistd.h>

Int main(void)

{

printf("Before fork\n");

fork();

printf("after fork\n");
```

}

shashi@linuxtechi ~}$

shashi@linuxtechi ~}$ cc 1_fork.c

shashi@linuxtechi ~}$ ./a.out

Before fork

After fork

shashi@linuxtechi ~}$

Whenever any system call is made there are plenty of things that take place behind the scene in any unix/linux machines.

First of all context switch happens from user mode to kernel(system) mode. This is based on the process priority and unix/linux operating system that we are using. In the above C example code we are using "{" opening curly brace which is the entry of the context and "}" closing curly brace is for exiting the context. The following table explains context switching very clearly.

vfork()

vfork –> create a child process and block parent process.

Note:- In vfork, signal handlers are inherited but not shared.

shashi@linuxtechi ~}$ man vfork

This will yield output mentioning what is vfork used for, syntax and along with all the required details.

vfork is as same as fork except that behavior is undefined if process created by vfork either modifies any data other than a variable of type pid_t used to store the return value p of vfork or calls any other function between calling _exit() or one of the exec() family.

Note: vfork is sometimes referred to as special case of clone.

Following is the C programming example for vfork() how it works.

shashi@linuxtechi ~}$ vim 1.vfork.c

```
#include<stdio.h>

#include<unistd.h>

Int main(void)

{

printf("Before fork\n");

vfork();

printf("after fork\n");

}
```

```
shashi@linuxtechi ~}$ vim 1.vfork.c

shashi@linuxtechi ~}$ cc 1.vfork.c

shashi@linuxtechi ~}$ ./a.out

Before vfork

after vfork

after vfork

a.out: cxa_atexit.c:100: __new_exitfn: Assertion `l != NULL' failed.

Aborted
```

Note:– As explained earlier, many a times the behaviour of the vfork system call is not predictable. As in the above case it had printed before once and after twice but aborted the call with _exit() function. It is better to use fork system call unless otherwise and avoid using vfork as much as possible.

Differences between fork() and vfork()

Vfork() behaviour explained in more details in the below program.

shashi@linuxtechi ~}$ cat vfork_advanced.c

```
#include <sys/types.h>

#include <unistd.h>

#include <stdio.h>

int main()

{

   int n =10;

   pid_t pid = vfork(); //creating the child process

   if (pid == 0)          //if this is a chile process

   {

      printf("Child process started\n");

   }

   else//parent process execution

   {

      printf("Now i am coming back to parent process\n");

   }

   printf("value of n: %d \n",n); //sample printing to check "n" value

   return 0;

}
```

shashi@linuxtechi ~}$ cc vfork_advanced.c

shashi@linuxtechi ~}$ ./a.out

Child process started

value of n: 10

Now i am coming back to parent process

value of n: 594325573

a.out: cxa_atexit.c:100: __new_exitfn: Assertion `l != NULL' failed.

Aborted

Note: Again if you observe the outcome of vfork is not defined. Value of "n" has been printed first time as 10, which is expected. But the next time in the parent process it has printed some garbage value.

wait()

wait() system call suspends execution of current process until a child has exited or until a signal has delivered whose action is to terminate the current process or call signal handler.

pid_t wait(int * status);

There are other system calls related to wait as below,

1) **waitpid()**: suspends execution of current process until a child as specified by pid arguments has exited or until a signal is delivered.

pid_t waitpid (pid_t pid, int *status, int options);

2) **wait3()**: Suspends execution of current process until a child has exited or until signal is delivered.

pid_t wait3(int *status, int options, struct rusage *rusage);

3) **wait4()**: As same as wait3() but includes pid_t pid value.

pid_t wait3(pid_t pid, int *status, int options, struct rusage *rusage);

exec()

exec() family of functions or sys calls replaces current process image with new process image.

There are functions like **execl**, **execlp**,**execle**,**execv**, **execvp** and **execvpe** are used to execute a file.

These functions are combinations of array of pointers to null terminated strings that represent the argument list , this will have path variable with some environment variable combinations.

exit()

This function is used for normal process termination. The status of the process is captured for future reference. There are other similar functions **exit(3)** and **_exit().**, which are used based on the exiting process that one is interested to use or capture.

Conclusion:-

The combinations of all these system calls/functions are used for process creation, execution and modification. Also these are called "shell" spawning set-of functions. One has to use these functions cautiously keeping in mind the outcome and behaviour.

Memory Management In Unix

Introduction to UNIX

According to Leon, 2007, UNIX is an operating system (OS) is software that manages hardware and software resources of a computer. UNIX was first developed in the 1960s and has been constant development ever since. UNIX is one of the most widely used operating systems in industry, government and education. It is a stable, multi-user, multi-tasking system for servers, desktops and laptops.

UNIX Memory Management

Memory is an important resource in computer. Memory management is the process of managing the computer memory which consists of primary memory and secondary memory. The goal for memory management is to keep track of which parts of memory are in use and which parts are not in use, to allocate memory to processes when they need it and de-allocate it when they are done. UNIX memory management scheme includes swapping and demand paging.

Memory Partitioning

The simplest form of memory management is splitting up the main memory into multiple logical spaces called partition. Each partition is used for separate program. There are 2 types of memory partitioning:-

Single Partition Allocation

Single partition allocation only separates the main memory into operating system and one user process area. Operating system will not able to have virtual memory using single partition. Using single partition is very ineffective because it only allows one process to run in the memory at one time.

Multiple Partition Allocation

Most of the operating system nowadays is using multiple partitions because it is more flexible. Multiple partition allocation enabled multiple programs run in the main memory at once. Each partition is used for one process.

There are two different forms of multiple partition allocation, which is fixed partitioning and variable partitioning. Fixed partitioning divides memory up into many fixed

partitions which cannot be change. However, variable partitioning is more flexible because the partitions vary dynamically in the later as processes come and go. Variable partitioning (Variable memory) has been used in UNIX.

UNIX Memory Management Strategies

Swapping

Swapping consists of bringing in each process in physical memory entirely and running it. When the process is no longer in use, the process will be terminated or is swapped out to disk.

Initially only process A is in memory. Then process B is swapped into memory from disk. After that, process A terminates or swapped out to disk. Then process C is swapped into the free space.

External Fragmentation Problem

The size of each process is different, therefore when the processes is been swapped in and out, there will be a multiple holes in the memory because UNIX is using variable partitioning.

Solution

There are two techniques to solve this problem, which are memory compaction and fit in the process using algorithms

Memory compaction moves all the processes upward as far as possible, so that all the free memory is placed in one large block. However, it is not a good idea because it requires a lots of CPU time.

Most processes will grow as they run, and the processes data segments can grow, as in many programming languages, the process will grow. If there is a hole is next to the process, it can be allocated and the process is allowed to grow into the hole. Therefore it is good to allocate some extra memory whenever a process is swapped in or out.

Algorithms

There are three different types of algorithm can be used to loads the program wherever the memory space is unused, which is first fit, best fit and worst fit.

First Fit

The memory manager scans along the list and allocates the first space to fit the process. First fit is a fast algorithm because it searches as little as possible.

Best Fit

The memory manager scans the whole list and takes the smallest hole that will fit the process. Best fit is slower than first fit because it must search the whole list every time it is called.

Worst Fit

The memory manager scans the whole list and takes the largest available hole, so that the hole broken will be big enough to be useful.

Virtual Memory

UNIX operating system allows user to fully utilize the physical memory installed in the system as well as part of the hard disk called swap space which have been designated for use by the kernel while the physical memory is insufficient to handle the tasks. Virtual memory managers will create a virtual address space in secondary memory (hard disk) and it will determine the part of address space to be loaded into physical memory at any given time. The benefit of virtual memory relies on separation of logical and physical memory.

Demand Paging

Paging is a memory allocation strategy by transferring a fixed-sized unit of the virtual address space called virtual page whenever the page is needed to execute a program. As the size of frames and pages are the same, any logical page can be placed in any physical frame of memory.

Find out how UKEssays.com can help you!

Our academic experts are ready and waiting to assist with any writing project you may have. From simple essay plans, through to full dissertations, you can guarantee we have a service perfectly matched to your needs.

Every processes will be logical divided and allocate in the virtual address space. There is a page table in the virtual memory to allocate and keep tracking of the pages to map into

the frames.

UNIX will perform page swapping only when the program needs a certain page. This procedure is called demand paging. The page will be paged into the memory only when it is needed to execute. The whole process will not be paged into the memory, only the pages needed are swapped in.

Demand paging decreases the paging time and physical memory needed because only the needed pages will be paged and the reading time of the unused pages can be avoided.