- Accounting information and limits

- Add the process to the ready list

- Initial allocation of memory and resources must be a subset of parent's and be assigned as shared Initial priority of the process can be greater than the parent's

**Management of processes in UNIX**

How processes are managed after creation in UNIX

1. Suspend - Change process state to suspended
    - A process may suspend only its descendants
    - May include cascaded suspension
    - Stop the process if the process is in running state and save the state of the processor in the process control block
    - If process is already in blocked state, then leave it blocked, else change its state to ready state
    - If need be, call the scheduler to schedule the processor to some other process
2. Activate - Change process state to active
    - Change one of the descendant processes to ready state
    - Add the process to the ready list
3. Destroy - Remove one or more processes
    - Cascaded destruction
    - Only descendant processes may be destroyed

- If the process to be “killed" is running, stop its execution
- Free all the resources currently allocated to the process
- Remove the process control block associated with the killed process

4. Change priority - Set a new priority for the process
    - Change the priority in the process control block
    - Move the process to a different queue to reflect the new priority

### 3.4.2 Scheduling in UNIX

Scheduler decides the process to run first by using a scheduling algorithm

#### 3.4.2.1 Type of scheduling used in UNIX

**Pre-emptibility**

In UNIX, Processes in user space are *pre-emptible* - what this means is that a process may have the CPU taken away from it arbitrarily. This is how pre-emptive multitasking works: the scheduling routine will periodically suspend the currently executing process, and possibly schedule another task to run on that CPU. This means that theoretically, a process can be in a situation where it never gets the CPU back. In reality the scheduling code has an interest in fairness and will try to give the CPU to each process with a weak level of fairness, but there are no guarantees

**Algorithms are:**

- Shortest Remaining Time Scheduling
    - Preemptive version of shortest job next scheduling
    - Preemptive in nature (only at arrival time)
    - Highest priority to process that need least time to complete
    - Priority function P
    - Schedule for execution
    - Average waiting time calculations
- Round-Robin Scheduling
    - Preemptive in nature
    - Preemption based on time slices or time quanta
    - Time quantum between 10 and 100 milliseconds
    - All user processes treated to be at the same priority
    - Ready queue treated as a circular queue

**Desirable features of a scheduling algorithm**

1. Fairness: Make sure each process gets its fair share of the CPU
2. Efficiency: Keep the CPU busy 100% of the time
3. Response time: Minimize response time for interactive users
4. Turnaround: Minimize the time batch users must wait for output
5. Throughput: Maximize the number of jobs processed per hour

## 3.5 DEVICE MANAGEMENT

To perform useful functions, processes need access to the peripherals connected to the computer, which are controlled by the kernel through device drivers. For example, to show the user something on the screen, an application would make a request to the kernel, which would forward the request to its display driver, which is then responsible for actually plotting the character/pixel.

### 3.5.1 Special features of Device management in UNIX

Device drivers run as part of the kernel, either compiled in or as run-time loadable modules. The kernel architectures, Monolithic kernel does this and it have the advantage of speed and efficiency.

- **Device manager**

Device manager will be the interface between the device drivers and the both the rest of the kernel and user applications.

The device manager needs to do two things:

1. Isolate devices drivers from the kernel so that driver writers can worry about interfacing to the hardware and not about interfacing to the kernel
2. Isolate user applications from the hardware so that applications can work on the majority of devices the user might connect to their system

In most operating systems, the device manager is the only part of the kernel that programmers really see. Writing a good interface will make the difference between an efficient and reliable OS which works with a variety of devices and an OS which you spend all your own time writing and debugger drivers for.

**Capabilities of device manager**

1. Asynchronous I/O: that is, applications will be able to start an I/O operation and continue to run until it terminates.
2. Plug and Play: drivers will be able to be loaded and unloaded as devices are added to and removed from the system. Devices will be detected automatically on system startup, if possible.

- **Drivers**

Because we want our kernel to be plug-and-play capable, it isn't enough for drivers to be added to the kernel at compile time, as Minix and old Linux do. We must be able to load and unload them at run time. This isn't difficult: it just means we have to extend the executable file interface to kernel mode.

- **Interfaces**

Once we've detected the devices installed in the system we need to keep a record of them somewhere. The standard Unix model, employed by Minix and Linux, is to keep directory somewhere in the file system. This directory is filled with special directory entries, directory entries which don't point to any data, each of which refers to a specific device via major and minor device numbers. The major device number specifies the device type or driver to use and the minor number specifies a particular device implemented by that drivers.

## 3.6 Security

An important kernel design decision is the choice of the abstraction levels where the security mechanisms and policies should be implemented. Kernel security mechanisms play a critical role in supporting security at higher levels.

One approach is to use firmware and kernel support for fault tolerance (see above), and build the security policy for malicious behavior on top of that (adding features such as cryptography mechanisms where necessary), delegating some responsibility to the compiler. Approaches that delegate enforcement of security

policy to the compiler and/or the application level are often called *language-based security*.

The lack of many critical security mechanisms in current mainstream operating systems impedes the implementation of adequate security policies at the application abstraction level. In fact, a common misconception in computer security is that any security policy can be implemented in an application regardless of kernel support.

**4.0 ADVANTAGES OF UNIX O/S**

- Unix is more flexible and can be installed on many different types of machines, including main-frame computers, supercomputers and micro-computers.

- Unix is more stable and does not go down as often as Windows does, therefore requires less administration and maintenance.

- Unix has greater built-in security and permissions features than Windows.

- Unix possesses much greater processing power than Windows.

- Unix is the leader in serving the Web. About 90% of the Internet relies on Unix operating systems running Apache, the world's most widely used Web server.

- Software upgrades from Microsoft often require the user to purchase new or more hardware or prerequisite software. That is not the case with Unix.

- The mostly free or inexpensive open-source operating systems, such as Linux and BSD, with their flexibility and control, are very attractive to (aspiring) computer

wizards. Many of the smartest programmers are developing state-of-the-art software free of charge for the fast growing "open-source movement”.

- Unix also inspires novel approaches to software design, such as solving problems by interconnecting simpler tools instead of creating large monolithic application programs.

**CHAPTER TWO - LINUX**

# Introduction

What is Linux?
Linux is a UNIX-like operating system that runs on many different computers. Linux was first released in 1991 by its author Linus Torvalds at the University of Helsinki and developed by Linus Torvalds (author) and Andrew Morton. Linux is the operating system *kernel*, which comes with a *distribution* of software The Linux kernel is an operating system kernel used by a family of Unix-like operating system. It started out as a personal computer system used by individuals, and has since gained the support of several large operations such as HP, IBM, and Sun microsystem. It now used mostly as the server operating system. It's a prime example of open source development system. It's written in C

Since then it has grown tremendously in popularity as programmers around the world embraced his project of building a free operating system, adding features, and fixing problems. Linux is portable, which means you'll find versions running on name-brand or clone PCs, Apple Macintoshes, Sun workstations, or Digital Equipment Corporation Alpha-based computers. Linux also comes with source code, so you can change or customize the software to adapt to your needs. Finally, Linux is a great operating system, rich in features adopted from other versions of UNIX. The term Linux distribution is used to refer to the various operating systems that run on top of the Linux kernel. Linux is one of the most prominent examples of free/open source software. Today, the Linux kernel has received contributions from thousands of programmers.

## Event Leading To the Creation

The UNIX operating system was conceived and implemented in 1960 and first released in 1970. Its portability and availability caused it to the widely adopted and modified by academic institutions and businesses. In 1983, Richard Stallman started the GNU project with the goal of creating a free UNIX like operating system. As part of the work, he wrote the GNU general public license (GPL). By the early 1990's there was almost enough available software to create a full operating system. However, the GNU kernel called HURD, failed to attract attention from developers leaving GNU incomplete. A solution seemed to appear in form of MINIX. It was released by Andrew S Tanenbaum in 1987, as an operating system, MINIX was not a superb one while source code was available, modification and retribution was restricted. This factors and lack of widely adopted free kernel made Torvalds start is project.

# Processes

The concept of a *process* is fundamental to any multiprogramming operating system. A process is usually defined as an instance of a program in execution; thus, if 16 users are running vi at once, there are 16 separate processes (although they can share the same executable code). Processes are often called "tasks" in Linux source code.

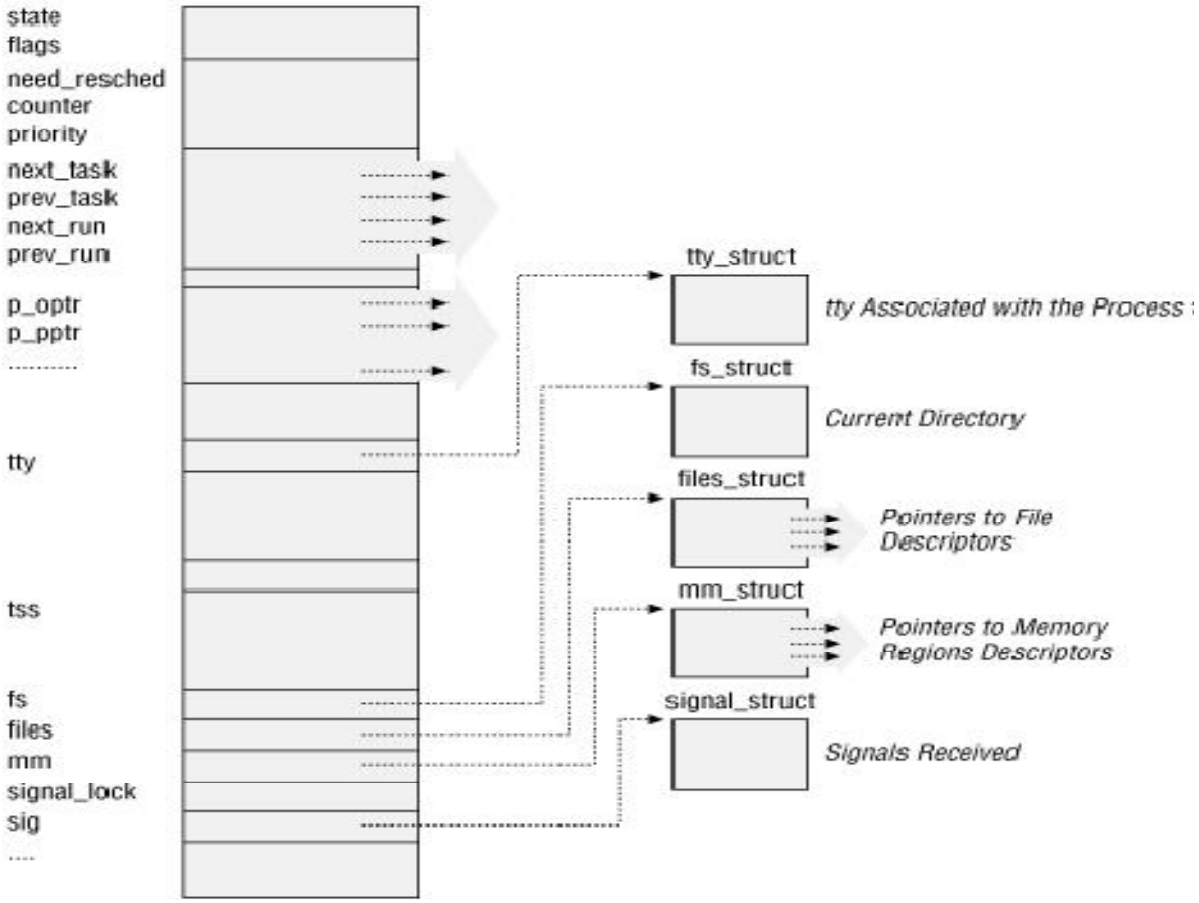
Properties of processes

- Static
- Dynamic

## Process Descriptor

In order to manage processes, the kernel must have a clear picture of what each process is doing. It must know, for instance, the process's priority, whether it is running on the CPU or blocked on some event, what address space has been assigned to it, which files it is allowed to address, and so on. This is the role of the *process descriptor*, that is, of a task_struct type structure whose fields contain all the information related to a single process. As the repository of so much information, the process descriptor is rather complex. Not only does it contain many fields itself, but some contain pointers to other data structures that, in turn, contain pointers to other structures. The figure below describes the Linux process descriptor schematically.

Figure 1 The Linux Process Descriptor




The five data structures on the right side of the figure refer to specific resources owned by the process. These resources will be covered in future chapters. This chapter will focus on two types of fields that refer to the process state and to process parent/child relationships.

## Process State

As its name implies, the 'state' field of the process descriptor describes what is currently happening to the process. It consists of an array of flags, each of which describes a possible process state. In the current Linux version these states are mutually exclusive, and hence exactly one flag of state is set; the remaining flags are cleared. The following are thepossible process states:

### TASK_RUNNING

The process is either executing on the CPU or waiting to be executed.

### TASK_INTERRUPTIBLE

The process is suspended (sleeping) until some condition becomes true. Raising a hardware interrupt, releasing a system resource the process is waiting for, or delivering a signal are examples of conditions that might wake up the process, that is, put its state back to TASK_RUNNING.

### TASK_UNINTERRUPTIBLE

Like the previous state, except that delivering a signal to the sleeping process leaves its state unchanged. This process state is seldom used. It is valuable, however, under certain specific conditions in which a process must wait until a given event occurs without being interrupted. For instance, this state may be used when a process opens a device file and the corresponding device driver starts probing for a corresponding hardware device. The device driver must not be interrupted until the probing is complete, or the hardware device could be left in an unpredictable state.

### TASK_STOPPED

Process execution has been stopped: the process enters this state after receiving a SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal. When a process is being monitored by another (such as when a debugger executes a ptrace( ) system call to monitor a test program), any signal may put the process in the TASK_STOPPED state.

### TASK_ZOMBIE

Process execution is terminated, but the parent process has not yet issued a wait( )-like system call (wait2( ), wait3( ), wait4( ), or waitpid( )) to return information about the dead process. Before the wait( )-like call is issued, the kernel cannot discard the data contained in the dead process descriptor because the parent could need it

## **Identifying A Process**

Any Unix-like operating system, on the other hand, allows users to identify processes by means of a number called the *Process ID* (or *PID*). The PID is a 32-bit unsigned integer stored in the PID field of the process descriptor. PIDs are

numbered sequentially: the PID of a newly created process is normally the PID of the previously created process incremented by one. However, for compatibility with traditional Unix systems developed for 16-bit hardware platforms, the maximum PID number allowed on Linux is 32767. When the kernel creates the 32768th process in the system, it must start recycling the lower unused PIDs.

## Memory Management

The memory management subsystem is one of the most important parts of the operating system. Since the early days of computing, there has been a need for more memory than exists physically in a system. Strategies have been developed to overcome this limitation and the most successful of these is virtual memory. Virtual memory makes the system appear to have more memory than is physically present by sharing it among competing processes as they need it. Virtual memory does more than just make your computer's memory go farther. The memory management subsystem provides:

### Large Address Spaces

The operating system makes the system appear as if it has a larger amount of memory than it actually has. The virtual memory can be many times larger than the physical memory in the system.

### Protection

Each process in the system has its own virtual address space. These virtual address spaces are completely separate from each other and so a process running one application cannot affect another. Also, the hardware virtual memory mechanisms allow areas of memory to be protected against writing. This protects code and data from being overwritten by rogue applications.

### Memory Mapping

Memory mapping is used to map image and data files into a process' address space. In memory mapping, the contents of a file are linked directly into the virtual address space of a process.