**Kernel Space/User Space**

In Linux, user memory and kernel memory are independent and implemented in separate address spaces. The address spaces are virtualized, meaning that the addresses are abstracted from physical memory (through a process detailed shortly). Because the address spaces are virtualized, many can exist. In fact, the kernel itself resides in one address space, and each process resides in its own address space. These address spaces consist of virtual memory addresses, permitting many processes with independent address spaces to refer to a considerably smaller physical address space (the physical memory in the machine).

The mapping of virtual memory to physical memory occurs through page tables. The hardware itself provides the mapping, but the kernel manages the tables and their configuration.

So following are the differences(this is also the difference between kernel module/user program):

* **Kernel modules have separate address space.** A module runs in **kernel space**. An application runs in **user space**. System software is protected from user programs. Kernel space and user space have their own memory address spaces.
* **Kernel modules have higher execution privilege.** Code that runs in kernel space has greater privilege than code that runs in user space. Driver modules potentially have a much greater impact on the system than user programs.
* **Kernel modules must be preemptable.**

**Interface of user space to kernel space and vice versa**

* **Socket based mechanism**
* **ioctl(discussed later)**
* **system calls**
* **Using signals**
* **mmap.(discussed later)**
* **procfs**
* **copy\_to\_user,copy\_from\_user,put\_user,get\_user**

**Procfs(explanation of proc)**

The procfs, located in /proc, is the best known interface of this class. It was originally designed to export all kind of process information such as the current status of the process, or all open file descriptors to the user space. Despite its initial purposes, the procfs has been used for a lot of other purposes:

* provide information about the running system such as cpu information, information about interrupts, about the available memory or the version of the kernel.
* information about "ide devices", "scsi devices" and "tty's".
* networking information such as the arp table, network statistics or lists of used sockets

The proc file system is a logical file system used by the kernel to export its information to the external world. The /proc filesystem was originally developed to provide information on the processes in a system. The effort of the /proc filesystem is to provide an easy way to view kernel and information about currently running processes. The /proc filesystem acts as an interface to internal data structures in the kernel. It can be used to obtain information about the system and to change certain kernel parameters at runtime (sysctl). The /proc fs can be used for system related tasks such as:

* finding out hardware information
* modifying runtime parameters
* viewing and modifying network and host parameters
* memory and performance information

**How to system calls work**

When a user mode process invokes a system call the CPU switches from user mode to kernel mode,and starts executing functions. In Linux system calls are invoked using the "int 0x80" assembly language instruction. "int 0x80" acts as an gateway to all the system calls implemented by the Linux kernel.

Kernel implements many system call. Parameters are passed to identify the system call number and arguments to the system call. This is done by storing values in the process register before calling the "int 0x80" instruction.

Normally the user will always call a library function.This library function(generated from \_\_syscall macros in asm/unistd.h) writes its arguements & no. Of system call(i.e sys\_call\_num & sys\_call\_args).to be defines transfer register and then triggers the 0x80 interrupt.when the appropiate ISR returns,the return value reads from the transfer register and library function terminates.

**Monolithic/Micro kernel**

**Monolithic kernels** - that is a kernel which take care of almost all the [system](http://linuxhelp.blogspot.com/2006/05/monolithic-kernel-vs-microkernel-which.html) tasks like interfacing between the sound card and the audio software, the graphics card [drivers](http://linuxhelp.blogspot.com/2006/05/monolithic-kernel-vs-microkernel-which.html) and so on. Linux is a perfect example of a monolithic kernel where most of these functions take place in the kernel space.(Linux is Monolithic). **microkernel architecture**, wherein device drivers and other code are loaded and executed on demand and are not necessarily always in memory.

The Linux kernel is mostly a monolithic kernel: i.e., all device drivers are part of the kernel proper. Unlike BSD, a Linux kernel's device drivers can be “loadable”, i.e., they can be loaded and unloaded from memory through user commands.

**2.4/2.6 kernel difference**

1. Dynamical loading of kernel module i.e there is no concept of .ko in 2.4.
2. İnsmod concept is newly implemented in 2.6 with also some new modutils package.
3. 2.6 kernel is preemptive while 2.4 in not

**Pre-emptive/non-Preemptive kernel**

A pre-emptive kernel forcibly timeslots processes, regardless of the processes desire to continue processing. A non-preemptive kernel allows processes to utilize the processor until they are ready to give it up. In the scenario of a non-preemptive kernel, a single process could easily hog the machine's processing time, and in effect, bring the machine to a halt.

Linux processes are *preemptive*. If a process enters the TASK\_RUNNING state, the kernel checks whether its dynamic priority is greater than the priority of the currently running process. If it is, the execution of current is interrupted and the scheduler is invoked to select another process to run (usually the process that just became runnable). Of course, a process may also be preempted when its time quantum expires. When this occurs, the need\_resched field of the current process is set, so the scheduler is invoked when the timer interrupt handler terminates.

**Types of scheduling**

* First-come, first-served scheduling (FCFS) algorithm
* Shortest Job First Scheduling (SJF) algorithm
* Shortest Remaining time (SRT) algorithm
* Non-preemptive priority Scheduling algorithm
* Preemptive priority Scheduling algorithm
* Round-Robin Scheduling algorithm
* Highest Response Ratio Next (HRRN) algorithm
* Multilevel Feedback Queue Scheduling algorithm
* Multilevel Queue Scheduling algorithm

First-come First-served(FCFS) Scheduling follow first in first out method. As each process becomes ready, it joins the ready queue. It is non-preemptive.

SJF-The process with the shortest expected processing time is selected for execution, among the available processes in the ready queue. SJF can be premptive or non-preemptive.

Priority-A Priority (an integer) is associated with each process.The CPU is allocated to the process with the highest priority

**Round-Robin**: It is similar to FCFS with preemption added.Round-Robin Scheduling is also called as time-slicing scheduling and it is a preemptive version based on a clock. That is a clock interrupt is generated at periodic intervals usually 10-100ms. When the interrupt occurs, the currently running process is placed in the ready queue and the next ready job is selected on a First-come, First-serve basis. This process is known as time-slicing, because each process is given a slice of time before being preempted.

Using round-robin scheduling allots a slice of time to each process that is running. In a [computer](http://www.wisegeek.com/what-is-a-computer.htm) for example, the user starts three applications, [Email](http://www.wisegeek.com/what-is-email.htm), a [web browser](http://www.wisegeek.com/what-is-a-web-browser.htm), and a [word processor](http://www.wisegeek.com/what-is-a-word-processor.htm). These applications are loaded into system memory as processes and each is allowed to run without the user considering which applications are running in the background.

Round-robin scheduling handles the sharing of resources between the three application processes (and the countless others running in the background completely invisible to the user). This scheduling works well because each application gets a certain amount of time per processor cycle. A processor cycle is the amount of time it takes the [CPU](http://www.wisegeek.com/how-does-a-cpu-work.htm) to manage each process running, one time.The running applications in the earlier example provide a short cycle for the processor and more time would be allotted to each of these three processes, making them appear to perform better to the end user.

**Process States**

**A Process state** is the stage of execution that a process is in. It is these states which determine which processes are eligible to receive CPU time.Following are the states:

1. executing in user mode
2. executing in [kernel](http://www.linux-tutorial.info/modules.php?name=MContent&obj=glossary&term=kernel) mode
3. ready to run
4. sleeping
5. newly created, not ready to run, and not sleeping
6. Zombie- process terminated, but information is still there in the process table.

**Context switch**

A **context switch** is the [computing](http://en.wikipedia.org/wiki/Computing) process of storing and restoring [state](http://en.wikipedia.org/wiki/State_(computer_science)) ([context](http://en.wikipedia.org/wiki/Context_(computing))) of a [CPU](http://en.wikipedia.org/wiki/Central_processing_unit) so that execution can be resumed from the same point at a later time. This enables multiple [processes](http://en.wikipedia.org/wiki/Process_(computing)) to share a single [CPU](http://en.wikipedia.org/wiki/Central_processing_unit)

The contexts from which driver code can execute are:

*user context*

Driver is called in the context of the user-level process that is requesting service. The driver can block and can do operations such as [**copyout**(D3)](http://docsrv.sco.com/cgi-bin/man/man?copyout+D3) that require access to the requesting process's user-level address space. The **ioctl**( ), **read**( ), and **write**( ) entry point routines execute in user context.

*interrupt context*

Driver is called outside the normal flow of execution, such as in response to an interrupt or because of the expiration of a timer. It is a special instance of the non-blockable context, which is functionally equivalent in a multiprocessor-safe driver

*Process context*: Each time a process is removed from access to the processor, sufficient information on its current operating state must be stored such that when it is again scheduled to run on the processor it can resume its operation from an identical position. This operational state data is known as its *context* and the act of removing the process's thread of execution from the processor (and replacing it with another) is known as a *process switch* or *context switch.*

**Fork()/vfork()**

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.

**Zombie/Orphan**

A zombie process or defunct process is a process that has completed execution but still has an entry in the process table, allowing the process that started it to read its exit status. An orphan process is a computer process whose parent process has finished or terminated.

**IPCs**

* Sockets
* Message queues
* Pipes/Fifos
* Semaphores/Mutex
* Signals
* Shared Memory

**Pipes/Fifos**

A simple example of using a pipe is the command:

ls | grep x

The **ls** program produces a list of files in the current directory, while the **grep** program reads the output of **ls** and prints only those lines containing the letter **x**.

**Named pipes-FIFO**

The named pipe has a directory entry. The directory entry allows using the FIFO for anyprocess which knows its name, unlike unnamed pipes which are restricted only to related processes.

int mkfifo(char \* path, mode\_t mode)

path points to the pathname of a file

mode access rights

**Unnamed pipes**

Unnamed pipes may be only used with related processes (parent/child or child/child having the same parent). They exist as long as their descriptors are open.

#include <unistd.h>

intpipe(int fd[2])

RETURNS: success : 0

fd [0] – descriptor used for reading

fd [1] - descriptor used for writing

Function creates the pipe and opens it for reading and writing

**Signals**

Signals offer another way to transition between Kernel and User Space. While system call are *synchronous* call originating *from User Space*, signals are *asynchronous* messages coming *from Kernel space*. Signals are always *delivered* by the Kernel but they can be *initiated* by:

* **other processes** on the system (using the [kill](http://docs.sun.com/app/docs/doc/816-5167/kill-2?a=view) command/system call)
* **the process itself**. This includes hardware exceptions triggered by a process: when a program executes an illegal instruction, such as dividing a number by zero or attempting to access a memory zone that has not been allocated yet, the hardware detects it and a signal is sent to the faulty program.
* **the Kernel**. The Kernel also use signals to notify a process of some system events, such as the arrival of out-of-band data. In the same way, when a program sets a system alarm, the Kernel sends a signal to the process every time a timer expires (e.g. every 10 seconds).

**Note**:There are 31 signals in unix/Linux like

SIGHUP - Hangup detected on controlling terminal or death of controlling process

SIGKILL - Sure way to terminate (kill) a process. Cannot be caught or ignored.

SIGALRM - Timer signal from alarm(2)

SIGIO - Asynchronous I/O now event

SIGINT - Interrupt from keyboard. Usually terminate the process. Can be triggered by Ctrl-C

**Semaphore/Mutex**

A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number. Threads can request access to the resource (decrementing the semaphore), and can signal that they have finished using the resource (incrementing the semaphore)."

Mutexes are typically used to serialise access to a section of  re-entrant code that cannot be executed concurrently by more than one thread. A mutex object only allows one thread into a controlled section, forcing other threads which attempt to gain access to that section to wait until the first thread has exited from that section

(A mutex is really a semaphore with value 1.)

Mutex can be released only by thread that had acquired it, while you can signal semaphore from any other thread (or process), so semaphores are more suitable for some synchronization problems like producer-consumer.(ownership)

Semaphores use a counter that counts how many tasks are waiting, a mutex is just a binary value. Semaphores can be used to organise queues while a mutex is grabbed by the first task that gets there.

So some of the more difference are

1) Mutex can be used only for mutual exclusion, while binary can be used of mutual exclusion as well as synchronisation.  
2) Mutex can be given only by the task that took it.  
3) Mutex cannot be given from an ISR.  
4) Mutual-exclusion semaphores can be taken recursively. This means that the semaphore can be taken more than once by the task that holds it before finally being released.  
5) Mutex provides a options for making the task that took it as DELETE\_SAFE. This means, that the task cannot be deleted when it holds mutex.

**Bottom Halves**

When the interrupt was asserted, the processor stopped what it was doing and the operating system delivered the interrupt to the appropriate device driver. Device drivers should not spend too much time handling interrupts as, during this time, nothing else in the system can run. There is often some work that could just as well be done later on. Linux's bottom half handlers were invented so that device drivers and other parts of the Linux kernel could queue work to be done later on.

There can be up to 32 different bottom half handlers, which are referenced through a vector of pointers called bh\_base. These pointers point to each of the kernel's bottom half handling routines. bh\_active and bh\_mask have their bits set according to what handlers have been installed and are active. If bit N of bh\_mask is set then the Nth element of bh\_base contains the address of a bottom half routine. If bit N of bh\_active is set then the Nth bottom half handler routine should be called as soon as the scheduler deems reasonable. These indices are statically defined. The timer bottom half handler (index 0) is the highest priority, the console bottom half handler (index 1) is next in priority and so on. Typically the bottom half handling routines have lists of tasks associated with them. For example, the *immediate* bottom half handler works its way through the immediate tasks queue (tq\_immediate), which contains tasks that need to be performed immediately.

Some of the kernel's bottom half handers are device specific, but others are more generic:

**TIMER**

This handler is marked as active each time the system's periodic timer interrupts and is used to drive the kernel's timer queue mechanisms,

**CONSOLE**

This handler is used to process console messages,

**TQUEUE**

This handler is used to process tty messages,

**NET**

This handler handles general network processing,

**IMMEDIATE**

This is a generic handler used by several device drivers to queue work to be done later.

Whenever a device driver, or some other part of the kernel, needs to schedule work to be done later, it adds work to the appropriate system queue, for example the timer queue, and then signals the kernel that some bottom half handling needs to be done. It does this by setting the appropriate bit in bh\_active. Bit 8 is set if the driver has queued something on the immediate queue and wishes the immediate bottom half handler to run and process it. The bh\_active bitmask is checked at the end of each system call, just before control is returned to the calling process. If it has any bits set, the bottom half handler routines that are active are called. Bit 0 is checked first, then 1 and so on until bit 31.

The bit in bh\_active is cleared as each bottom half handling routine is called. bh\_active is transient; it only has meaning between calls to the scheduler and is a way of not calling bottom half handling routines when there is no work for them to do.

**Softlink/Hardlink**

Soft or symbolic is more of a short cut to the original file....if you delete the original the shortcut fails and if you only delete the short cut nothing happens to the original.  
  
Hard link is more of a mirror copy....do something to file1 and it appears in file 2 deleting one still keeps the other ok.

Soft links will have a different inode value.Hardlink shares the same inode.

ls –l will show links.

Using ln –sf softlink is being created.

**Important system calls/commands**

System calls

Open, read, write, close, ioctl, mmap, select, poll,

Commands

grep,make,mknod,ctags,ps,ls,chmod,mount,kill

**Linux File Systems structure**

The following lists the most common directories and their intended contents.

* **/** - root directory
* **/home** - where directories are contained for each user, example:
* **/usr** - pronounced 'user' and contains Linux commands and utilities
  + **/bin** - binary executable programs
  + **/lib** - program libraries, similar to Windows 'dll' files
  + **/sbin** - more executable programs and Linux utilities for administrative purposes
  + **/doc** - documentation
  + **/src** - source code to programs
* **/tmp** - temporary work files
* **/etc** - configuration files
  + **/rc.d** - scripts used during boot and shutdown process
  + **/sysconfig** - default configuration files
  + **/sysconfig/network-scripts** - network scripts
  + **/sysconfig/daemons** - special programs that run in background, such as print spooling
* **/bin** - binary executable programs that all users need
* **/dev** - device files that control drives, terminals and any equipment attached to the server
* **/var** - user specific files
  + **/log** - log files containing system usage and errors
  + **/spool** - where spooled files are stored during print spooling process
  + **/mail** - where Email files are stored until retrieved by client Email program
* **/proc** - system files
* **/root** - root's home directory
* **/opt** - other options
* **/sbin** - more executable programs and utilities

**Inode**

The **inode (index node)** is a fundamental concept in the Linux and UNIX filesystem. Each object in the filesystem is represented by an inode. But what are the objects? Let us try to understand it in simple words. Each and every file under Linux (and UNIX) has following attributes:

=> File type (executable, block special etc)  
=> Permissions (read, write etc)  
=> Owner  
=> Group  
=> File Size  
=> File access, change and modification time (remember UNIX or Linux never stores file creation time, this is favorite question asked in UNIX/Linux sys admin job interview)  
=> File deletion time  
=> Number of links (soft/hard)  
=> Extended attribute such as append only or [no one can delete file](http://www.cyberciti.biz/tips/linux-password-trick.html) including [root user (immutability)](http://www.cyberciti.biz/tips/howto-write-protect-file-with-immutable-bit.html)  
=> Access Control List (ACLs)

All the above information stored in an inode. In short the inode identifies the file and its attributes (as above) . Each inode is identified by a unique inode number within the file system. Inode is also known as index number.

ls –i

**Types of files in unix/Linux**

Regular file

b - block special file (/devices/isa/fdc@1,3f0/fd@0,0:c)

c - character special file (/dev/fd/0)

d - directory (/)

D - door (/var/run/name\_service\_door)

p - pipe (/etc/cron.d/FIFO)

l - link (/proc/449/path/4)

s - socket file (/var/run/.inetd.uds)

f - fifo (/export/home/xtreme4/.bash\_history)