UNIT II: Open source operating systems: LINUX:

Introduction – General Overview – Kernel Mode and user mode. Process – Advanced Concepts – Scheduling – Personalities – Cloning – Signals.

LINUX:

Introduction: Linux is an Operating System that was first created at the University of Helsinki in Finland by a young student named Linus Torvalds. At this time the student was working on a UNIX system that was running on an expensive platform. Because of his low budget, and his need to work at home, he decided to create a copy of the UNIX system in order to run it on a less expensive platform, such as an IBM PC.

He began his work in 1991 when he released version 0.02 and worked steadily until 1994 when version 1.0 of the Linux Kernel was released. The current full-featured version at this time is 2.2.X; released January 25, 1999, and development continues.

Reasons to use Linux

- **Configurability**

Linux distributions give the user full access to configure just about any aspect of their system. Options range from the simple and straightforward.

- **Convenience**

While Linux takes some effort to get set up, once it is set up, it is surprisingly low-maintenance.

- **Stability**

Linux is based on the UNIX kernel. It providespreemptive multitasking and protected memory.

- **Community**

Linux is part of the greater open-source community. This consists of thousands of developers and many more users world-wide whosupport open software.This user and developer base is also a support base.

- **Freedom**

Linux is free. This means more than just costingnothing. This means that you are allowed to do whatever you want to with the software. This is why Redhat, Mandrake, and Suse are all allowed to sell their own distributions of Linux

General Overview:

The Linux Kernel:

The Linux Kernel is an operating system, which runs on a wide variety of hardware and for a variety of purposes. Linux is capable of running on devices as simple as a wrist watch, or a cell phone, but it can also run on a home computer using, for example Intel, or AMD processors, and its even capable of running on high end servers using Sun Sparc CPU's or IBM power PC processors. Some Linux distro's can only run one processor, while others can run many at once.

The Linux kernel is an operating system kernel used by the Linux family of Unix-likeoperating systems. It is one of the most prominent examples of free and open source software.

The Linux kernel was initially conceived and created by Finnishcomputer science student Linus Torvalds in 1991. Linux rapidly accumulated developers and users who adopted code from other free

software projects for use with the new operating system. The Linux kernel has received contributions from thousands of programmers. Many Linux distributions have been released based upon the Linux kernel.

Properties of Linux:

Linux Pros:

A lot of the advantages of Linux are a consequence of Linux' origins, deeply rooted in UNIX, except for the first advantage, of course:

Linux is free:

Linux is portable to any hardware platform:

Linux was made to keep on running:

Linux is secure and versatile:

Linux is scalable:

The Linux OS and most Linux applications have very short debug-times:

Linux Cons

There are far too many different distributions:

Linux is not very user friendly and confusing for beginners:

Is an Open Source product trustworthy?

Linux vs. Windows

TOPIC	LINUX	WINDOWS
Price	The majority of Linux variants are available for free or at a much lower price than Windows.	Microsoft Windows can run between $20.00 - $150.00 US dollars per each license copy.
Reliability	The majority of Linux variants and versions are notoriously reliable and can often run for months and years without needing to be rebooted.	Although Microsoft Windows has made great improvements in reliability over the last few versions of Windows, it still cannot match the reliability of Linux.
Software	Linux has a large variety of available software programs, utilities, and games. However, Windows has a much larger selection of available software	Because of the large amount of Microsoft Windows users, there is a much larger selection of available software programs, utilities, and games for Windows.

Software Cost	Many of the available software programs,utilities, and games available on Linux arefreeware and/or open source. Even such complex programs such as Gimp, OpenOffice, StarOffice, and wine are available for free or at a low cost.	Although Windows does have software programs, utilities, and games for free, the majority of the programs will cost anywhere between $20.00 - $200.00+ US dollars per copy.
Hardware	Linux companies and hardware manufacturers have made great advancements in hardwaresupport for Linux and today Linux will support most hardware devices. However, many companies still do not offer drivers or support for their hardware in Linux.	Because of the amount of Microsoft Windows users and the broader driver support, Windows has a much larger support for hardware devices and a good majority of hardware manufacturers will support their products in Microsoft Windows.
Security	Linux is and has always been a very secure operating system. Although it still can be attacked when compared to Windows, it muchmore secure.	Although Microsoft has made great improvements over the years with security on their operating system, their operating system continues to be the most vulnerableto viruses and other attacks.
Open Source	Many of the Linux variants and many Linux programs are open source and enable users to customize or modify the code however they wish to.	Microsoft Windows is not open source and the majority of Windows programs are not open source.
Support	Although it may be more difficult to find users familiar with all Linux variants, there are vast amounts of available online documentation and help available for Linux.	Microsoft Windows includes its own help section, has vast amount of available online documentation and help, as well as books oneach of the versions of Windows.

Kernel Mode and user:

A kernel is the main program or component of an operating system, it's the thing that does all the work, and without it you have no operating system. Linux is actually nothing more than a kernel; the programs that make up the rest of the operating system are generally GNU software, so the entire operating system is usually referred to as GNU/Linux.

User mode is the normal mode of operating for programs. Web browsers, calculators, etc. will all be in user mode. They don't interact directly with the kernel, instead, they just give instructions on what needs

to be done, and the kernel takes care of the rest. Kernel mode, on the other hand, is where programs communicate directly with the kernel. A good example of this would be device drivers. A device driver must tell the kernel exactly how to interact with a piece of hardware, so it must be run in kernel mode. Because of this close interaction with the kernel, the kernel is also a lot more vulnerable to programs running in this mode, so it becomes highly crucial that drivers are properly debugged before being released to the public.

A process can run in two modes:

1.User Mode.

2.Kernel Mode.

User Mode:

A mode of the CPU when running a program. In this mode, the user process has no access to the memory locations used by the kernel. When a program is running in User Mode, it cannot directly access the kernel data structures or the kernel programs.

The kernel-mode programs run in the background, making sure everything runs smoothly - things like printer drivers, display drivers, drivers that interface with the monitor, keyboard, mouse, etc. These programs all run in such a way that you don't notice them.

When the computer boots up, Windows calls the KERNEL, the main kernel-mode program that allows all the other programs to run, even the user-mode programs.

User mode is the normal mode of operating for programs, web browsers etc. They don't interact directly with the kernel, instead, they just give instructions on what needs to be done, and the kernel takes care of the rest. Kernel mode, on the other hand, is where programs communicate directly withthe kernel.

A good example of this would be device drivers. A device driver must tell the kernel exactly how to interact with a piece of hardware, so it must be run in kernel mode. Because of this close interaction with the kernel, the kernel is also a lot more vulnerable to programs running in this mode, so it becomes highly crucial that drivers are properly debugged before being released to the public.

These are the programs that you run when you want specific programs - e.g., MS Paint, MS Word, and Calculator. These are heavily restricted, as to not crash the system. Windows uses memory-protection services offered by the processor toprevent malicious programs from interfering with the rest of the system and corrupting it.

In User mode, the executing code has no ability to directlyaccess hardware or reference memory. Code running in user mode must delegate to system APIs to access hardware or memory.

Due to the protection afforded by this sort of isolation, crashes in user mode are always recoverable. Most of the code running on your computer will execute in user mode.

Kernel Mode:

A mode of the CPU when running a program. In this mode, it is the kernel that is running on behalf of the user process and directly access the kernel data structures or the kernel programs. Once the system call returns, the CPU switches back to user mode.

The kernel-mode programs run in the background, making sure everything runs smoothly - things like printer drivers, display drivers, drivers that interface with the monitor,
keyboard, mouse, etc. These programs all run in such a way that you don't notice them.

When the computer boots up, Windows calls the KERNEL, the main kernel-mode program that allows all the other programs to run, even the user-mode programs. Kernel mode, also referred to as system mode, is one of the two distinct modes of operation of the CPU in Linux. When the CPU is in kernel mode, it is assumed to be executingtrusted software, and thus it can execute any instructionsand reference any memory addresses (i.e., locations inmemory). The kernel (which is the core of the operatingsystem and has complete control over everything that occursin the system) is trusted software, but all other programsare considered untrusted software.

In this mode executing code has complete and unrestricted access to the underlying hardware. It can execute any CPU instruction and reference any memory address. Kernel mode is generally reserved for the lowest-level, most trusted functions of the operating system. Crashes in kernel mode are catastrophic; they will halt the entire PC.

When the CPU is in kernel mode, it is assumed to be executing trusted software, and thus it can execute any instructions and reference any memory addresses (i.e., locations in
memory).

The kernel (which is the core of the operating system and has complete control over everything that occurs in the system) is trusted software, but all other programs are considered
untrusted software. Thus, all user mode software must request use of the kernel by means of a system call in order to perform privileged instructions, such as process creation or input/output operations.

Distinguish Between User Mode and Kernel Mode

- Kernal mode has higher priority while user mode has lower priority.
- Kernal mode has ability to read user mode but user mode has not priority to interface in kernal mode.
- When a process is in user space, its system calls are being intercepted by the tracing thread. When it's in the kernel, it's not under system call tracing. This is the distinction between user mode and kernel mode.
- The transition from user mode to kernel mode is done by the tracing thread. When a process executes a system call or receives a signal, the tracing thread forces the process to run in the kernel if necessary and continues it without system call tracing.

PROCESS:

Definition: A process is an executing (i.e., running) instance of a program. Process are frequently referred to as tasks.

Process (process ID, PID): All software runs within an operating system concept known as a process. Each program running on a system is assigned its own process ID (PID). Users can easily obtain a process list (using Task Manager on Windows or ps on UNIX) in order to see what is running.

Processes inside out

Multi-user and multi-tasking:

Multiuser refers to having more than 1 person able to log into the computer and each person have their own settings (bookmark, desktop, themes, etc) Multitasking is the ability of the computer to do more than 1 thing (program) at a time. There was a time when you could not surf, type in word and listen to music on your computer all at one time.

Process types:

Interactive processes

Interactive processes are initialized and controlled through a terminal session. In other words, there has to be someone connected to the system to start these processes; they are not started automatically as part of the system functions

Running a process in the background is only useful for programs that don't need user input (via the shell).Putting a job in the background is typically done when execution of a job is expected to take a long time. In the example, using graphical mode, we open an extra terminal window from the existing one:

aravind:~> **xterm &**

```
[1]          26558
aravind:~> jobs
[1]+ Running xterm &
```

Controlling processes

Command	Meaning
regular_command	Runs this command in the foreground.
command &	Run this command in the background (release the terminal)
Jobs	Show commands running in the background.
Ctrl+Z	Suspend (stop, but not quit) a process running in the foreground (suspend).
Ctrl+C	Interrupt (terminate and quit) a process running in the foreground.
Bg	Reactivate a suspended program in the background.
Fg	Puts the job back in the foreground.
Kill	End a process

Automatic processes:

Automatic or batch processes are not connected to a terminal. Rather, these are tasks that can be queued into a (spooler area) where they wait to be executed on a FIFO (first-in, first-out) basis. Such tasks can be executed using one of two criteria:

At a certain date and time: done using the **at** command.

At times when the total system load is low enough to accept extra jobs: done using the **batch** command.

Daemons

A daemon process has no controlling terminal. It cannot open /dev/tty. Most daemons tend to last a long time, be owned by root, and do something useful. A daemon is just a process that runs in the background, usually waiting for something to happen that it is capable of working with, like a printer daemon is waiting for print commands.

Daemons are server processes that run continuously. Most of the time, they are initialized at system startup and then wait in the background until their service is required.

Process attributes

A process has a series of characteristics, which can be viewed with the **ps** command:

The process ID or PID: a unique identification number used to refer to the process.

The parent process ID or PPID: the number of the process (PID) that started this process.

Nice number: the degree of friendliness of this process toward other processes (not to be confused with process priority, which is calculated based on this nice number and recent CPU usage of the process).

Terminal or TTY: terminal to which the process is connected.

User name of the real and effective user (RUID and EUID): the owner of the process.

Real and effective group owner (RGID and EGID): The real group owner of a process is the primary group of the user who started the process.

Displaying process information:

The **ps** command is one of the tools for visualizing processes. This command has several options which can be combined to display different process attributes.With no options specified, **ps** only gives information about the current shell and eventual processes:

aravind:~> **ps**


```
PID TTY TIME CMD
4245 pts/7 00:00:00 bash
5314/7 00:00:00 ps
```

Life and death of a process:

Process creation

A new process is created because an existing process makes an exact copy of itself. This child process has the same environment as its parent, only the process ID number is different. This procedure is called *forking*. After the forking process, the address space of the child process is overwritten with the new process data. This is done through an *exec* call to the system.

Ending processes

When a process ends normally (it is not killed or otherwise unexpectedly interrupted), the program returns its *exit status* to the parent. This exit status is a number returned by the program providing the results of the program's execution.

SIGNALS:

Processes end because they receive a signal. There are multiple signals that you can send to a process. Use the **kill** command to send a signal to a process. The command **kill -l** shows a list of signals. Most signals are for internal use by the system, or for programmers when they write code. As a user, you will need the following signals:

Common signals:

Signal name	Signal number	Meaning
SIGTERM	15	Terminate the process in an orderly way.
SIGINT	2	Interrupt the process. A process can ignore this signal.
SIGKILL	9	Interrupt the process. A process can not ignore this signal.
SIGHUP	1	For daemons: reread the configuration file.

Boot process, Init and shutdown:

PC Boot and Linux Init Process:

BIOS: The Basic Input/Output System is the lowest level interface between the computer and peripherals. The BIOS performs integrity checks on memory and seeks instructions on the Master Boor Record (MBR) on the floppy drive or hard drive.

The MBR points to the boot loader (GRUB or LILO: Linux boot loader).

Boot loader (GRUB or LILO) will then ask for the OS label which will identify which kernel to run and where it is located (hard drive and partition specified). The installation process requires to creation/identification of partitions and where to install the OS. GRUB/LILO are also configured during this process. The boot loader then loads the Linux operating system.

The first thing the kernel does is to execute init program. Init is the root/parent of all processes executing on Linux.

The first processes that init starts is a script /etc/rc.d/rc.sysinit.

Based on the appropriate run-level, scripts are executed to start various processes to run the system and make it functional.

One of the most powerful aspects of Linux concerns its open method of starting and stopping the operating system, where it loads specified programs using their particular configurations, permits you to change those configurations to control the boot process, and shuts down in a graceful and organized way.

Beyond the question of controlling the boot or shutdown process, the open nature of Linux makes it much easier to determine the exact source of most problems associated with starting up or shutting down your system. A basic understanding of this process is quite beneficial to everybody who uses a Linux system. A lot of Linux systems use lilo, the LInux LOader for booting operating systems. We will only discuss GRUB, however, which is easier to use and more flexible.

Init: The kernel, once it is loaded, finds **init** in sbin and executes it.When **init** starts, it becomes the parent or grandparent of all of the processes that start up automatically on your Linux system. The first thing **init** does, is reading its initialization file, /etc/inittab. This instructs **init** to read an initial configuration

script for the environment, which sets the path, starts swapping, checks the file systems, and so on. Basically, this step takes care of everything that your system needs to have done at system initialization: setting the clock, initializing serial ports and so forth.

Then **init** continues to read the /etc/inittabfile, which describes how the system should be set up in each run level and sets the default *run level*. A run level is a configuration of processes. All UNIX-like systems can be run in different process configurations, such as the single user mode, which is referred to as run level 1 or run level S (or s). In this mode, only the system administrator can connect to the system. It is used to perform maintenance tasks without risks of damaging the system or user data. Naturally, in this configuration we don't need to offer user services, so they will all be disabled. Another run level is the reboot run level, or run level 6, which shuts down all running services according to the appropriate procedures and then restarts the system.

Use the **who** to check what your current run level is:

aravind@home:~> **who -r**

```
run-level 2 2010-10-17 23:22          last=S
```

Init run levels

Available run levels are generally described in /etc/inittab, which is partially shown below:

#

inittab This file describes how the INIT process should set up

the system in a certain run-level.

Default run level. The run levels are:

0 - halt (Do NOT set initdefault to this)

1 - Single user mode

2 - Multiuser, without NFS# (The same as 3, if you do not have networking)

3 - Full multiuser mode

4 - unused

5 - X11

6 - reboot (Do NOT set initdefault to this)

#

id:5:initdefault:

<--cut-->

Scheduling:

A Linux system can have a lot to suffer from, but it usually suffers only during office hours. Whether in an office environment, a server room or at home, most Linux systems are just idling away during the

morning, the evening, the nights and weekends. Using this idle time can be a lot cheaper than buying those machines you'd absolutely need if you want everything done at the same time.

There are three types of delayed execution:

Waiting a little while and then resuming job execution, using the **sleep** command. Execution time depends on the system time at the moment of submission.

Running a command at a specified time, using the **at** command. Execution of the job(s) depends on system time, not the time of submission.

Regularly running a command on a monthly, weekly, daily or hourly basis, using the **cron** facilities.

1. **The at command:** The **at** command executes commands at a given time, using your default shell unless you tell the command.The options to **at** are rather user-friendly, which is demonstrated in the examples below:

```
aravind@home:~> at tomorrow + 2 days
warning: commands will be executed using (in order) a) $SHELL
b) login shell c) /bin/sh
at> cat reports | mail myboss@mycompany
at> <EOT>
job 1 at 2010-10-16 12:36
```

Typing **Ctrl+D** quits the **at** utility and generates the "EOT" message. User *aravind* does a strange thing here combining two commands

```
aravind@home:~> at 0237
warning: commands will be executed using (in order) a) $SHELL
b) login shell c) /bin/sh
at> cd new-programs
at> ./configure; make
at> <EOT>
job 2 at 2010-10-14 02:00
```


The -m option sends mail to the user when the job is done, or explains when a job can't be done. Thecommand **atq** lists jobs; perform this command before submitting jobs in order prevent them from starting at the same time as others. With the **atrm** command you can remove scheduled jobs if you change your mind.

The **at** command is used to schedule one or more programs for a single execution at some later time. There are actually four client commands:

1. at: Runs commands at specified time
2. atq: Lists pending commands
3. atrm: Cancels pending jobs

4. batch: Runs commands when system load permits

The Linux **at** command accepts a number of time specifications, considerably extending the POSIX.2 standard.

(2) **Cron and crontab:**

The cron system is managed by the **cron** daemon. It gets information about which programs and when they should run from the system's and users' crontab entries. Only the root user has access to the system crontabs, while each user should only have access to his own crontabs. On some systems (some) users may not have access to the cron facility.

At system startup the cron daemon searches /var/spool/cron/ for crontab entries which are named after accounts in /etc/passwd, it searches /etc/cron.d/ and it searches /etc/crontab, then uses this information every minute to check if there is something to be done. It executes commands as the user who owns the crontab file and mails any output of commands to the owner.

Syntax

crontab [-e] [-l] [-r] [filename]

-e	edit a copy of the current user's crontab file, or creates an empty file to edit if crontab does not exist.
-l	list the crontab file for the invoking user.
-r	remove a user's crontab from the crontab
Filename	The filename that contains the commands to run.

Lines that can be in the crontab file.

minute (0-59),
hour (0-23),
day of the month (1-31),
month of the year (1-12),
day of the week (0-6 with 0=Sunday).

Example:

crontab–e fields,

min	hour	dayofmonth	monthofyear	dayofweek	Command
0	12	14	2	*	Echo "hai"

Options	Explanation
*	Is treated as a wild card. Meaning any possible value.
*/5	Is treated as ever 5 minutes, hours, days, or months. Replacing the 5 with another numerical value will change this option.
2,4,6	Treated as an OR, so if placed in the hours, this could mean at 2, 4, or 6 o-clock.
9-17	Treats for any value between 9 and 17. So if placed in day of month this would be days 9 through 17. Or if put in hours it would be between 9 and 5.

3.System Calls Related to Scheduling:

We change the scheduling parameters by means of the system calls illustrated in the Table

Table 10-1: System Calls Related to Scheduling

System Call	Description
nice( )	Change the priority of a conventional process.
getpriority( )	Get the maximum priority of a group of conventional processes.
setpriority( )	Set the priority of a group of conventional processes.
sched_getscheduler( )	Get the scheduling policy of a process.
sched_setscheduler( )	Set the scheduling policy and priority of a process.
sched_getparam( )	Get the scheduling priority of a process.
sched_setparam( )	Set the priority of a process.
sched_yield( )	Relinquish the processor voluntarily without blocking.
sched_get_ priority_min( )	Get the minimum priority value for a policy.
sched_get_ priority_max( )	Get the maximum priority value for a policy.
sched_rr_get_interval( )	Get the time quantum value for the Round Robin policy.

Most system calls shown in the table apply to real-time processes, thus allowing users to develop real-time applications.

Personalities:

Linux supports different execution domains, or personalities, for each process. Among other things, execution domains tell Linux how to map signal numbers into signal actions. The execution domain system allows Linux to provide limited support for binaries compiled under other Unix-like operating systems.

This function will return the current personality() when persona equals 0xffffffff. Otherwise, it will make the execution domain referenced by persona the new execution domain of the current process.

Name

personality - set the process execution domain

Synopsis

#include <sys/personality.h>

int personality(unsigned long *persona*);

Return Value

On success, the previous persona is returned. On error, -1 is returned, and errno is set appropriately.

- **Cloning:**

Making an image copy of your system disk is a great way to create a backup. With the cost of portable USB drives at all time lows, you could keep a couple around for rotation purposes. If your main drive does crash, you could be back up and running in a matter of minutes. Now all that's left is making the actual image copy. There are lots of ways to accomplish this task, and we'll spell a few of them out to help you along.

The biggest issue with making an image copy of your system disk is that you must boot from another device to get a true copy. Options include a "live" CD or bootable USB. You probably have a copy of your favorite distribution's installation disk lying around somewhere, so that would be an obvious choice. For our purposes we'll use the Ubuntu 10.4 distro on a USB disk. The second option would be to use an actual disk duplication distro like Clonezilla. This turns out to be one of the easier ways to get the job done, especially if you're not too comfortable with the command line.

Option One: Bootable Ubuntu USB Disk:

Creating a bootable Ubuntu USB disk is a simple task if you have a running system. It's really not that hard if you don't either. The one thing you do need is the distribution ISO file.

You should be able to boot straight from the disk once you have it created. It's possible you might have to change your BIOS setting to allow the system to boot from USB. Many newer systems (like Dell machines) have an option to bring up a boot menu when the machine first powers up by pressing the F12 key. Once you get the system booted you're ready to make your backup image copy. You might want to run the Disk Utility found under the System / Administration list. This will give you the opportunity to look at the disks attached to your system and their organization.

The Disk Utility provides a number of utilities including mount / dismount and format volume. It's a good idea to go ahead and format the drive if you're reusing an old Windows disk. GParted 0.5.1 comes standard with the basic Ubuntu 10.04 boot disk. It includes an option to copy a partition. Instructions for doing the work can be found on either the GParted site or on the Ubuntu forums. There's also a GParted Live CD if you want to go that route.

Be prepared to wait a while if you choose to backup your system to an external USB drive. The total estimated time in our case was almost four hours. One really good alternative is to use a disk caddy adapter

like the Thermaltake BlackX ST0005U. It has an eSATA connector that speeds up the data transfer process tremendously. This is a must have if you're the type that frequently tears into systems or builds new ones.

Option Two: Clonezilla:

Clonezilla is a Linux distribution specifically created for the purpose of cloning disk drives. It works for virtually any file system you can think of. Clonezilla comes in two basic flavors, Live and SE. The live version works in much the same way as the Ubuntu Live USB disk. You boot your computer from the Live USB and perform the disk copy operations on any drives connected to the computer. Clonezilla uses a number of tools along with a simple menu system to help guide you through the process. The default partition copy tool is Partclone. Clonezilla SE (Server Edition) is meant to be used to clone disks over a network.

The latest release of Clonezilla is 1.2.5.17 and comes in either a Debian or Ubuntu version. You can now download an AMD64 version which includes support for 64-bit versions of all the applications and the imaging of large partitions. All the applications have been updated to the latest versions along with the version of the Linux kernel. The hardest part of using Clonezilla to image your hard drives is making sure you know which drive is the master and which drive will be your copy. Clonezilla also takes care of copying the Master Boot Record (MBR) while accomplishing the same task with the Ubuntu Live CD method requires some command line magic.

Option Three: dd:

If you're a command line wizard, you could always use the dd command. The command to image a drive with dd would be something like the following:

 # dd if=/dev/sda of=/dev/sdb

This assumes that /dev/sda is the drive you wish to copy and /dev/sdb is the target drive. You'll find this method to be about the same speed as the GParted method mentioned in Option One above. It really doesn't matter which method you choose. The important thing is that you do some kind of system backup. Computers do fail from time to time, and it seems to be just at the time when you can least afford it. Save yourself some grief down the road and go backup your system now. Go ahead, we'll wait.

Signals: A signal is an event generated by Linux in response to some condition, which may cause a process to take some action when it receives the signal. "Raise" is the term that indicates the generation of a signal. "Catch" is the term to indicate the receipt of a signal. Introduced in UNIX systems to simplify IPC. Used by the kernel to notify processes of system Events. A signal is a short message sent to a process, or group of processes, containing the number identifying the signal. POSIX-4 defines i/f for queuing & ordering RT signals w/ arguments. Linux supports 31 non-real-time signals.

Processes end because they receive a signal. There are multiple signals that you can send to a process. Use the **kill** command to send a signal to a process. The command **kill -l** shows a list of signals. Most signals are for internal use by the system or for programmers when they write code.

Sending signals:

A program can signal a different program using the kill()system call with prototype int

kill(pid_t pid, int sig);

This will send the signal with number sig to the process with process ID pid. Signal numbers are small positive integers. (For the definitions on your machine, try /usr/include/bits/signum.h. Note that these definitions depend on OS and architecture.)

A user can send a signal from the command line using the killcommand. Common uses are kill -9 N to kill the process with pid N, or kill -1 N to force process N (maybe init or inetd) to reread its configuration file.

Certain user actions will make the kernel send a signal to a process or group of processes: typing the interrupt character (probably Ctrl-C) causes SIGINT to be sent, typing the quit character (probably Ctrl-\) sends SIGQUIT, hanging up the phone (modem) sends SIGHUP, typing the stop character (probably Ctrl-Z) sends SIGSTOP.

Certain program actions will make the kernel send a signal to that process: for an illegal instruction one gets SIGILL, for accessing non existing memory one gets SIGSEGV, for writing to a pipe while nobody is listening anymore on the other side one gets SIGPIPE, for reading from the terminal while in the background one gets SIGTTIN, etc.

Receiving signals:

When a process receives a signal, a default action happens, unless the process has arranged to handle the signal. For the list of signals and the corresponding default actions, see signal(7). For example, by default SIGHUP, SIGINT, SIGKILL will kill the process; SIGQUIT will kill the process and force a core dump; SIGSTOP, SIGTTIN will stop the process; SIGCONT will continue a stopped process; SIGCHLD will be ignored.

Traditionally, one sets up a handler for the signal using the signalsystem call with prototype

typedef void (*sighandler_t)(int);

sighandler_t signal(int sig, sighandler_t handler);

This sets up the routine handler() as handler for signals with number sig. The return value is (the address of) the old handler. The special values SIG_DFL and SIG_IGN denote the default action and ignoring, respectively.

When a signal arrives, the process is interrupted, the current registers are saved, and the signal handler is invoked. When the signal handler returns, the interrupted activity is continued.

It is difficult to do interesting things in a signal handler, because the process can be interrupted in an arbitrary place, data structures can be in arbitrary state, etc. The three most common things to do in a signal handler are (i) set a flag variable and return immediately, and (ii) (messy) throw away all the program was doing, and restart at some convenient point, perhaps the main command loop or so, and (iii) clean up and exit.

Blocking signals:

Each process has a list (bitmask) of currently blocked signals. When a signal is blocked, it is not delivered (that is, no signal handling routine is called), but remains pending.

The sigprocmask () system call serves to change the list of blocked signals. The sigpending ()system call reveals what signals are (blocked and) pending.

The sigsuspend () system call suspends the calling process until a specified signal is received. **Linux Common Signals are:**

Signal name	Signal number	Meaning
SIGHUP	1	Hangup (POSIX)
SIGINT	2	Terminal interrupt (ANSI)
SIGQUIT	3	Terminal quit (POSIX)
SIGILL	4	Illegal instruction (ANSI)
SIGTRAP	5	Trace trap (POSIX)
SIGIOT	6	IOT Trap (4.2 BSD)
SIGBUS	7	BUS error (4.2 BSD)
SIGFPE	8	Floating point exception
SIGKILL	9	Kill(can't be caught or ignored) (POSIX)
SIGUSR1	10	User defined signal 1 (POSIX)