INDEX

USER SPACE

KERNEL SPACE

LINUX PROJECT

The Linux project is aimed to explore the various intricacies of a beauty called Linux. The most important aspect of this project is the fact that whenever we get stuck, we get to learn new things about this mighty operating system.

Linux Basics:

First of all, Linux is a FLOSS software and there is no corporation to package this software

DISTRIBUTION

Windows and other Commercial softwares come in packages but in case of Linux there is no central corporation to do so. So groups or communities in Linux group together the Linux kernel, an installer package, X windows system, a desktop environment and certain necessary software to make up a distribution.

LICENSING

Before we go into the depths about Linux, we would like to make a note about licensing. Licensing is essentially the process of allowing other users to use your work in a systematic way. It does not terminate your copyrights or patents; but rather makes sure that you are in some form enlisted as one of the creator or contributor of the software, in case you were about to release it directly without any license.

GNU GPL

The GNU General Public Licence (GPL) is probably one of the most commonly used licenses for open-source projects. The GPL grants and guarantees a wide range of rights to developers who work on open-source projects. Basically, it allows users to legally copy, distribute and modify software. This means you can:

- **Copy the software.**
  Copy it onto one's own servers, client's servers, personal computer, pretty much anywhere. There's no limit to the number of copies one can make.

- **Distribute the software however you want.**
  Provide a download link on one's website. Put the software on a bunch of thumb drives and give them away. Print out the source code and throw it from the rooftops (please don't, though, because that would waste a lot of paper and make a mess).

- **Charge a fee to distribute the software.**
  If one wants to charge someone to provide the software, set it up on their website or do anything else related to it, one can do so. *But*, he must give them a copy of the GNU GPL, which basically tells them that they could probably get the software elsewhere for free.

- **Make whatever modifications to the software you want.**
  If one wants to add or remove functionality, go ahead. If one wants to use a portion of the code in another project, he can. The only catch is that the other project must also be released under the GPL.

COMMON TERMINAL COMMANDS

GENERAL REMARKS

A command is executed only after Enter is pressed.

An option is a specification (usually preceded with a dash (-), as in ls -a) which changes the execution of a command. The same option character may have a different meaning for another command. GNU programs take long options, preceded by two dashes (--), like ls --all.

The argument(s) to a command are specifications for the object(s) on which one wants the command to take effect. An example is ls /etc, where the directory /etc is the argument to the ls command. This indicates that you want to see the content of that directory, instead of the default, which would be the content of the current directory, obtained by just typing ls followed by Enter. Some commands require arguments, sometimes arguments are optional.

In Linux, like in UNIX, directories are separated using forward slashes. The absolute path of a system is the path starting from the root(/) directory. It is called the absolute path because it is unambiguous. Relative path is the path relative to the directory the user is in. In this context the symbols . and .. have special meaning. '.'refers to the current directory while '..' refers to the previous directory.

KEY COMBINATIONS IN BASH

Ctrl + A	Go to the Beginning of the line.
Ctrl + C/D	End command
Ctrl + E	Go to the end of the line
Ctrl + H	Backspace
ArrowUp and ArrowDown	View previously executed Commands
Shift+PgUp and Shift+PgDn	Go one page up or down respectively
Tab	Auto complete Directory
Tab Tab	List auto complete options.

INFORMATION COMMANDS

apropos
This command searches the list of commands via the keyword entered for possible commands which can do the task the user wants to do.

whatis
Command to get a short description of the command being searched for

man
Command to open the manual pages of any command

info
Command to get the info pages of a command

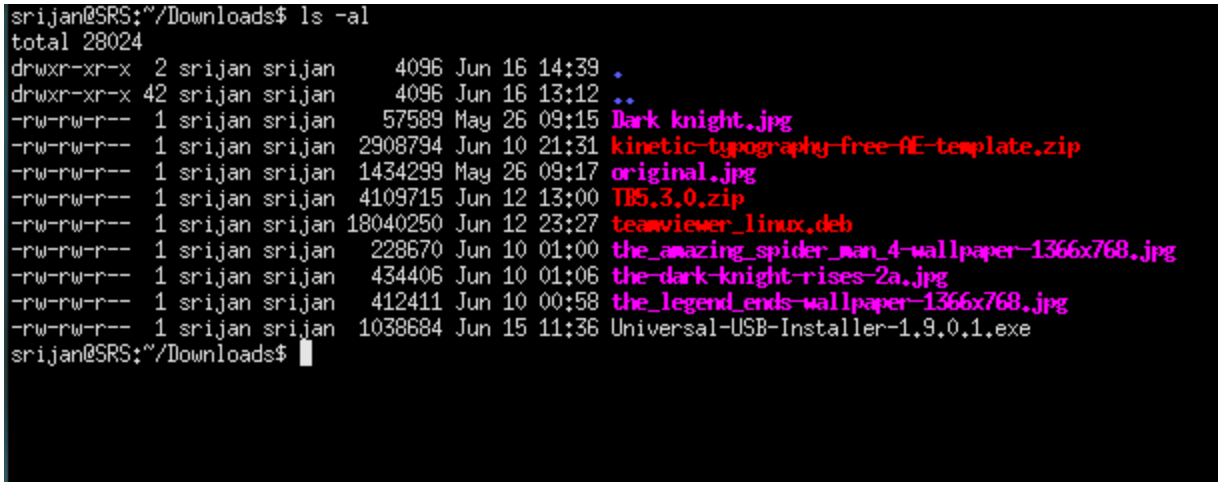
date
List the system date and time

MANIPULATING FILES

ls
The ls command lists the contents of a directory. The default is the current directory. The ls has many options which can be used like:

ls -a	Show hidden files.
ls -l	Give a long listing of files.
ls -t	Arrange files according to modification dates
ls -r	Reverse listing

Here is a sample long listing:



By default the ls option in most distributions is loaded with the --color and -F option for easy understanding of newbies. This can be checked by issuing the **alias** command which check what the commands default options are as listed in the /etc/default directory.

jamie@Ubunut:~ alias ls
alias ls='ls --color=auto'

The colour codes are:

Color	File type
Green	Executables
Blue	Directories
Red	Compressed Archives

White	Text Files
Pink	Images
Yellow	Devices
Cyan	Links
Flashing Red	Broken Links

cp
In the first form shown below, cp copies the contents of *file1* to *file2*. In the second form, it copies all files to the *dir* directory:

cp file1 file2
cp file1 ... fileN dir

Another important option is the cp -R which copies entire directories from one place to another:

cp -R dir1 dir2

mv
In the first form below, mv renames *file1* to *file2*. In the second form, it moves all files to the *dir* directory:

mv file1 file2
mv file1 ... fileN dir

rm
To delete (remove) a file, use rm. After you remove a file, it's gone. Do not expect to be able to "undelete" anything.

rm file

mkdir
The mkdir command creates a new directory, *dir*:

mkdir dir

The mkdir command will not create subdirectories automatically. To do the same we have to use the -p option.

richard:~/archive> mkdir 2001/reports/Restaurants-Michelin/
mkdir: cannot create directory `2001/reports/Restaurants-Michelin/':
No such file or directory
richard:~/archive> mkdir -p 2001/reports/Restaurants-Michelin/

rmdir
The rmdir command removes the directory *dir*:

rmdir dir

If *dir* isn't empty, this command fails. However, one can use rm -rf *dir* to delete a directory and its contents, but be careful. This is one of the few commands that can do serious damage, especially if you run it as the superuser. The -r option specifies recursive delete, and -f forces the delete operation.

cd
The cd command changes the shell's current working directory to *dir*:

cd dir

pwd
This program's name stands for "print working directory," and the command outputs the current working directory. That's all that pwd does, but it's useful.

echo
The echo command prints its arguments to the standard output:

echo Hello there.

cat
This command is used to view the contents of a file in one go:

cat file
cat file1 file2 file3

more and less
The more command is used in place of the cat command, it displays one page at a time. The user can move to the next page by pressing space, exit the file by pressing q and going back one page pressing b. (Enter can be used to go one line ahead)

more filename

less is the GNU version of more and has extra features allowing highlighting of search strings, scrolling back etc.

less filename

head and tails
These commands are used to view the first n and last n lines of a particular file respectively.

diff
To see the differences between two text files, use diff:

diff file1 file2

There are several options that can control the format of the output, such as -c, but the default output format is often the most comprehensible .

FINDING FILES

Shell auto-complete
GNU provides the excellent feature of auto complete where in the user can type half the file and press Tab. The Tab button finds all the files in the current directory and searches for a match. If there are no ambiguities, it directly lists the file otherwise nothing happens; in such a case pressing Tab twice gives a list of all the possible matches.

whereis
This is another search utility which can be used to search the PATH of the user

which
The which command searches the PATH of the user for any matching files. The PATH of the user is a shell environmental variable which contains all the possible paths in which any executable file can be found. To display the PATH one can use the command.

echo $PATH

The PATH of any user can also be changed if the user requires by the following command.

export PATH=$PATH: dir

The above command appends a new directory to the existing path. A way of completely changing the path from scratch is:

PATH= dir1: dir2:/
dir3:dir4
export PATH

find
The find command is one of the most useful commands to find files everywhere across the system. It is a very flexible command which has many options.

find <dir> -name <searchstring>	Search files with matching name
find <dir> -size <searchstring>	Search files with matching size

locate
The find command can take an awful lot of time at times, this is where the locate command comes in. The locate finds a file from a database of files created everyday by the system. Although this may not always give the right result, it is a very fast tool.

These days the locate command is a symbolic link to the slocate, viz the secure locate command. The slocate command does not show the files to which the user has not permission to access like important system configuration files and all.

grep
grep prints the lines from a file or input stream that match an expression. For example, if one wants to print the lines in the /etc/passwd file that contain the text root, use this command:

grep root /etc/passwd

The grep command is extraordinarily handy when operating on multiple files at once, because it prints the filename in addition to the matching line when in this multiple-file mode. For example, if one wants to check on every file in /etc that contains root, one could use this command:

*grep root /etc/**

Two of the most important grep options are -i (for case-insensitive matches) and -v (which inverts the search; that is, it prints all lines that *don't* match).

grep -w search_string

Matches the entire search string while searching.

Shell History search
A list of all the commands executed by the user is stored in the file .bahs_history. The user can view this file anytime he wants by either pressing Ctrl+R in the shell prompt and the typing in the search string or by using the grep command.

Shell Wildcards
The shell is capable of matching simple patterns with files in the current working directory. The simplest of these is the star character (*), which means match any number of arbitrary characters. For example, the following command prints a list of files in the current directory:

*echo **

After matching files to wildcards, the shell substitutes the filenames for the wildcard in the command line and then runs the revised command line. Here are some more wildcard examples: at* matches all files starting with at; *at matches files that end with at; and *at* matches any files that contains at. If no files match a wildcard, the shell does no substitution, and the command runs with literal characters such as * .

Another shell wildcard character is the question mark (?), instructing the shell to match exactly one arbitrary character. For example, b?at matches boat and brat.

Shell Special Characters
Characters that have a special meaning to the shell have to be escaped. This means that these characters have to be preceded by some other character to use them in their normal usage. The escape character in Linux is backslash. The most common special characters are /, ., ? and *. A full list can be found in the Info pages and documentation of the shell.

For instance, say that you want to display the file "*" instead of all the files in a directory, you would have to use

less *

The same goes for filenames containing a space:

cat This\ File

ADVANCED COMMANDS

sudo
Unlike the other flavors of Linux, Ubuntu usually keeps the root user password locked. But there is a possible workaround for this. We can prep-end sudo to any command we want to execute as root; the general arguments and uses of sudo are given below, one can also refer to the documentation available at https://help.ubuntu.com/community/RootSudo.

sudo command	to run a command as root
sudo -i	executing all commands as root
sudo -s	Run command in user environment with root
sudo sh -c “command”	makes the shell run the command as root
sudo -i -u username	run commands as the username.
gksudo	sudo for graphical actions.

sudo !! To perform the previous command as root

file
If you see a file and are unsure of its format, try using file to see if the system can guess, based on a large set of rules, file tries to make an educated guess about the format of a file. But in the end, the result of file is a guess, so file can surely be tricked.

file file

Some examples include:

mike:~> file Documents/
Documents/: directory

mike:~> file high-tech-stats.pdf
high-tech-stats.pdf: PDF document, version 1.2

mike:~> file Nari-288.rm
Nari-288.rm: RealMedia file

mike:~> file bijlage10.sdw
bijlage10.sdw: Microsoft Office Document

mike:~> file logo.xcf
logo.xcf: GIMP XCF image data, version 0, 150 x 38, RGB Color

mike:~> file cv.txt
cv.txt: ISO-8859 text

mike:~> file image.png
image.png: PNG image data, 616 x 862, 8-bit grayscale, non-interlaced

mike:~> file figure
figure: ASCII text
Introduction to Linux

mike:~> file me+tux.jpg
me+tux.jpg: JPEG image data, JFIF standard 1.01, resolution (DPI),
"28 Jun 1999", 144 x 144

mike:~> file 42.zip.gz
42.zip.gz: gzip compressed data, deflated, original filename,
`42.zip', last modified: Thu Nov 1 23:45:39 2001, os: Unix

mike:~> file vi.gif
vi.gif: GIF image data, version 89a, 88 x 31

mike:~> file slide1
slide1: HTML document text

mike:~> file template.xls
template.xls: Microsoft Office Document

mike:~> file abook.ps
abook.ps: PostScript document text conforming at level 2.0

mike:~> file /dev/log
/dev/log: socket

mike:~> file /dev/hda
/dev/hda: block special (3/0)

ps

It is used to view all the running processes.

ps -e	View all processes
ps -el	View all processes in detail
ps -ef	View all processes with file listing.
ps -L	View all thread
ps -forest	View hierarchy
ps -p	View process ID
ps -u	View User ID.

COMMANDS DEALING WITH USER

Linux has a primary system console which is a combination of the keyboard - the text entry or standard input- and the monitor - the standard output or display - can be used to switch between virtual consoles/ virtual terminals which are a conceptual combination of the console, that is, they are not associated with the console until and unless they are active.Virtual consoles are used to execute processes independently by different users simultaneously. (An important point here is that the terminal which we run when in X is a terminal emulator and not a Virtual Terminal).

One can shift between such virtual consoles - when in the X Windows System - by pressing Ctrl+Alt+F1; seven virtual consoles exists with the seventh being the X Windows System. Some commands dealing with such consoles are:

id
This command gives the id of the user currently the user is logged in as. It not only includes the User ID (UID) but also his primary group ID (GID) but also a list of other groups of which the user is a part.
tty
Prints the device path of the current virtual teminal/ terminal emulator.

who
Lists all the user logged in right now with a listing of their terminals as well. **ps -ef | grep bash** can be also used to view who all is logged in, the advantage of using this command is that in addition to listing users, it also makes a difference between virtual terminals and emulators. The bash terminals preceded by hyphen '-' are the virtual consoles.

last
Gives a listing of the some previous logins by users.

exit
Logouts the current terminal session and gives the prompt to the login daemon

EPILOGUE

Some general points to remember:

1. A program which takes one file can take multiple files by placing space between the files.

I/O REDIRECTION

STANDARD INPUT AND OUTPUT

Many Linux commands take input and produce some output. The default device or file from which the command or the program takes in input is called the standard input (which by default is the keyboard) and the place where the program prints its output is called the standard output (which by default is the console).
Being the very flexible system Linux is, it allows the user to change from where the program/ command takes in input and prints output. This is done through I/O redirection.

BASIC I/O REDIRECTION

In basic I/O redirection, the first thing that can be tried is to redirect the output of a command or program to a file for inspecting it later on. This is achieved by the greater than sign '>'.

```
nancy:~> cat test1
some words

nancy:~> cat test2
some other words

nancy:~> cat test1 test2 > test3

nancy:~> cat test3
some words
```

Redirecting "nothing" to an existing file is equal to emptying the file:

```
nancy:~> ls -l list
-rw-rw-r--    1    nancy  nancy  117    Apr 2   18:09   list

nancy:~> > list
nancy:~> ls -l list
-rw-rw-r--    1    nancy          0      Apr 4   12:01   list
```

This process is called truncating. The same redirection to an nonexistent file will create a new empty file with the given name:

```
nancy:~> ls -l newlist
ls: newlist: No such file or directory

nancy:~> > newlist

nancy:~> ls -l newlist
-rw-rw-r--    1    nancy          nancy  0      Apr 4   12:05   newlist
```

An important point to remember is that the '>' operator will overwrite any existing file, which can be potentially very dangerous. In order to append in a file rather than overwrite it, we use the '>>' operator.

```
mike:~> cat wishlist
more money
less work

mike:~> date >> wishlist

mike:~> cat wishlist
more money
less work
```

Thu Feb 28 20:23:07 CET 2002

The date command is used to display the system date and time.

Passing a file may be useful but another very important redirection is the passing of the output of one command to another command which is achieved by the pipe '|' operator. Some examples using piping of commands:

To find a word within some text, display all lines matching "pattern1", and exclude lines also matching "pattern2" from being displayed:

grep pattern1 file | grep -v pattern2

To display output of a directory listing one page at a time:

ls -la | less

To find a file in a directory:

ls -l | grep part_of_file_name

This was about redirecting the output, another feature is redirecting the input of a program/ command which is achieved by the less than command '<'. The example includes

andy:~> mail mike@somewhere.org < to_do

COMBINING REDIRECTIONS

Input and output redirections can be combined in a single command to perform complex operations in one step. Examples:

The file text.txt is first checked for spelling mistakes, and the output is redirected to an error log file:

spell < text.txt > error.log

The following command lists all commands that you can issue to examine another file when using less:

mike:~> less --help | grep -i examine

If you want to save output of this command for future reference, redirect the output to a file:

mike:~> less --help | grep -i examine > examine-files-in-less
mike:~> cat examine-files-in-less

Output of one command can be piped into another command virtually as many times as you want, just as long as these commands would normally read input from standard input and write output to the standard output. Sometimes they don't, but then there may be special options that instruct these commands to behave according to the standard definitions.

FILE DESCRIPTORS

Whenever a Linux Program has to work with any file, it has to open the file. When a file is opened, the program associates a number with that file. Any Linux program by default opens the following files when it starts up.

- standard input: 0
- standard output: 1

- standard error: 2

Some practical examples will make this more clear:

ls > file 2>&1

This command instructs that the standard output of ls should be redirected to the file and 2>&1 instructs the shell to send messages headed to stderr (2) to the same place messages to stdout (1) are sent. In this example, that place is file. Also note > is equivalent to 1>, and < is short for <0. A point to note is that the redirections persist.

ls 2>&1 > dirlist

will only direct standard output to dirlist and the standard error to standard output. This can be a useful option for programmers.

Note: Here, the ampersand merely serves as an indication that the number that follows is not a file name, but rather a location that the data stream is pointed to. Also note that the bigger-than sign should not be separated by spaces from the number of the file descriptor. If it would be separated, we would be pointing the output to a file again.

If a process generates a lot of errors, this is a way to thoroughly examine them:

command 2>&1 | less

This is often used when creating new software using the make command, such as in:

andy:~/newsoft> make all 2>&1 | less
--output ommitted--

Constructs like these are often used by programmers, so that output is displayed in one terminal window, and errors in another. Find out which pseudo terminal you are using issuing the tty command first:

andy:~/newsoft> make all 2> /dev/pts/7

The descriptors 3–9 can be connected to normal files, and are mainly used in shell scripts

WRITING OUTPUT AND FILES SIMULTANEOUSLY

The tee command can be used to do so. Using the -a option to tee results in appending input to the file(s). This command is useful if one wants to both see and save output. The > and >> operators do not allow to perform both actions simultaneously.

This tool is usually called on through a pipe (|), as demonstrated in the example below:

mireille ~/test> date | tee file1 file2
Thu Jun 10 11:10:34 CEST 2004

mireille ~/test> cat file1
Thu Jun 10 11:10:34 CEST 2004

mireille ~/test> cat file2
Thu Jun 10 11:10:34 CEST 2004

mireille ~/test> uptime | tee -a file2 load average: 0.04, 0.16, 0.26
11:10:51 up 21 days, 21:21, 57 users,

mireille ~/test> cat file2 load average: 0.04, 0.16, 0.26
Thu Jun 10 11:10:34 CEST 2004

11:10:51 up 21 days, 21:21, 57 users,

FILTERS

Filters are program which take some input and produce some output, that is they are mostly used to format the output of a given program. Using a filter with the output of a given program increases the readability of the output.

grep
Grep scans the output line per line, searching for matching patterns. All lines containing the pattern will be printed to standard output. This behavior can be reversed using the -v option. Some examples: suppose we want to know which files in a certain directory have been modified in February:

jenny:~> ls -la | grep Feb

The -i option is used to make grep case insensitive. A lot of GNU extensions are available as well, such as --colour, which is helpful to highlight search terms in long lines. A recursive grep that searches all sub-directories of encountered directories can be issued using the -r option. As usual, options can be combined.

sort
The command sort arranges lines in alphabetical order by default:

thomas:~> cat people-I-like | sort

Auntie Emmy
Boyfriend
Dad
Grandma
Mum
My boss

Sort can perform more functions as well. With this command, directory content is sorted smallest files first, biggest files last:

ls -la | sort -h

The sort command is also used in combination with the uniq program (or sort -u) to sort output and filter out double entries:

thomas:~> cat itemlist

1
4
2
5
34
567
432
567
34
555

thomas:~> sort -u itemlist
1
2
34

4
432
5
555
567

FILESYSTEMS

INTRODUCTION

Everything in Linux is a file; if it is not a file then it is a process.

Most files are just files, called regular files; they contain normal data, for example text files, executable files or programs, input for or output from a program and so on.

The various types of files - apart from the normal regular executable or text files - are:

1. Directories: files that are lists of other files.
2. Special files: the mechanism used for input and output. Most special files are in /dev.
3. Links: a system to make a file or directory visible in multiple parts of the system's file tree.
4. (Domain) sockets: a special file type, similar to TCP/IP sockets, providing inter-process networking protected by the file system's access control.
5. Named pipes: act more or less like sockets and form a way for processes to communicate with each other, without using network socket semantics.

The -l option to ls displays the file type, using the first character of each input line:

jaime:~/Documents> ls -l
total 80
-rw-rw-r-- 1 jaime jaime 31744 Feb 21 17:56 intro Linux.doc
-rw-rw-r-- 1 jaime jaime 41472 Feb 21 17:56 Linux.doc
drwxrwxr-x 2 jaime jaime 4096 Feb 25 11:50 course

The meaning of the various symbols in the ls -l listing are:

-: Regular file
d: Directory
l: Link
c: Special file
s: Socket
p: Named pipe
b: Block device

Another important point to remember is that files preceded by the period are hidden files in Linux and can be viewed using the ls -a command.

PARTITIONS

A partition is a part of a hard disk which can essentially function as a separate hard disk. There are three types of partitions of any disk:

1. Primary
2. Extended
3. Logical

Due to the limitations of the master boot record, which is referred to while booting, there can be a maximum of only 4 primary partitions. To overcome this problem only one of the

four partitions can be made into a extended partition which can be further sub-divided into any number of logical partition. A logical partition is no different from a primary partition except for the fact that some OSes can only boot from a primary partition. (Another important distinguishing feature is that all the logical partitions are contiguous, that is, one logical partition contains the adress of the next by the help of a pointer.)

In Linux the following naming scheme is used for naming the partitions or hard drives.

Any IDE/SCSI drive is given a name of sdx (where x is an alphabet) and its partitions are given the names sdxy (where y is a digit starting from 1). The digits 1 to 4 are reserved for only primary partitions so the numbering of logical drives starts from 5 irrespective of the number of primary partitions. The information about the various partitions is stored in the / dev directory with each partition having a separate file. The name of the file represents the name given to the partition.

Linux necessary needs only two partitions which are the root and the swap partition respectively. (They can be either logical or primary). A swap partition is essentially a part of the Hard disk which can be used as extra RAM if required.

Every information of the partitions of a system is stored in the Partition table in the Master Boot Record.

PARTITIONS IN LINUX

Linux requires two basic types of partitions which are:

1. Data Partition: This type of partition is used to store the normal types of files, rather this is the other type of partition for normal data files.
2. Swap partition: This is the swap space for the system which is used in case the system runs out of memory.

Linux usually needs a root partition, one or two more data partitions and one or two swap spaces. The root partition is the main partition which contains all the system essential command binaries, configuration files, shared library files, modules etc. It may also contain the boot images of the kernel which may alternatively be kept in a separate filesystem named /boot.

The swap space is space reserved to be used when the system runs out of memory, this is the reason that Linux rarely crashes. It may be argued that the read/write time on a disk is way slower than on the RAM, yet this safety measure is very useful.

The partitions are attached to the system in Linux via the mount points; the mount point is a place where the filesystem of a partition is loaded onto to the directory tree of Linux so that it becomes a part of the single unified directory tree unlike Windows.

The different partitions are mounted onto the system using the mount command (covered up later). Also there is a list of partitions which are automatically mounted. This list is maintained at the /etc/fstab file. To know more about the partitions in a Linux system, one uses the df command with the h option (human readable)

freddy:~> df -h

Filesystem	*Size*	*Used*	*Avail*	*Use%*	*Mounted on*
/dev/hda8	*496M*	*183M*	*288M*	*39%*	*/*
/dev/hda1	*124M*	*8.4M*	*109M*	*8%*	*/boot*
/dev/hda5	*19G 1*	*5G*	*2.7G*	*85%*	*/opt*
/dev/hda6	*7.0G*	*5.4G*	*1.2G*	*81%*	*/usr*
/dev/hda7	*3.7G*	*2.7G*	*867M*	*77%*	*/var*
fs1:/home	*8.9G*	*3.7G*	*4.7G*	*44%*	*/.automount/fs1/root/hom*

Apart from these, Linux can also have the following data partitions depending on the system.

1. a partition for usr
2. a partition for var
3. a partition for home
4. a partition for opt

FILESYSTEM

A filesystem is an interface which provides the options for properly updating, storing and retrieving data which has to be permanently stored in the computer. They are also responsible for maintaining a proper hierarchy in the directory tree so that the files can be properly maintained and backed up. It also decides which programs get access to the data stored in it. Filesystems are generally associated with flash drives, hard drives etc. The filesystem is usually tuned with the device it is associated with and the various partitions are labelled according to the filesystems they are meant to be used with.

Unlike Windows which has different filesystem types in different drives; Linux has a single unified directory tree. This does not mean that different filesystems cannot be used in Linux; in fact Linux provides the option of mounting the filesystem in any block storage device onto this directory tree at certain mount-points like /media or /mnt.

As stated in the IBM developers guide “Usually, the kernel starts this mount process by mounting the filesystem on some hard drive partition as /. You may mount other hard drive partitions as /boot, /tmp, or /home. You may mount the filesystem on a floppy drive as /mnt/floppy, and the filesystem on a CD-ROM as /media/cdrom1, for example. You may also mount files from other systems using a networked filesystem such as NFS. There are other types of file mounts, but this gives you an idea of the process. While the mount process actually mounts the *filesystem* on some device, it is common to simply say that you "mount the device," which is understood to mean "mount the filesystem on the device."

Now, suppose you have just mounted the root file system (/) and you want to mount a CD-ROM, /dev/sr0, at the mount point /media/cdrom. The mount point must exist before you mount the CD-ROM over it. When you mount the CD-ROM, the files and sub-directories on the CD-ROM become the files and sub-directories in and below /media/cdrom. Any files or sub-directories that were already in /media/cdrom are no longer visible, although they still exist on the block device that contained the mount point /media/cdrom. If the CD-ROM is unmounted, then the original files and sub-directories become visible again. You should avoid this problem by not placing other files in a directory intended for use as a mount point.”

FILESYSTEM TYPES IN LINUX

A filesystem type usually is an implementation of the standard functionality of the filesystem definition associated with a particular partition. So partitions and filesystems are used synonymously. The content stored in a filesystem is the crux of the filesystem and it is the deciding factor which is used to decide which filesystem needs to be used. So, referring to a filesystem may synonymously refer to the content stored in it.

Some of the filesystem types that are supported by Linux are listed below:

minix
is the file system used in the Minix operating system, the first to run under Linux. It has a number of shortcomings: a 64MB partition size limit, short filenames, a single timestamp, etc. It remains useful for floppies and RAM disks.

ext

is an elaborate extension of the **minix** file system. It has been completely superseded by the second version of the extended file system (**ext2**) and has been removed from the kernel (in 2.1.21).

ext2
is the high performance disk file system used by Linux for fixed disks as well as removable media. The second extended file system was designed as an extension of the extended file system (**ext**). **ext2** offers the best performance (in terms of speed and CPU usage) of the file systems supported under Linux.

ext3
is a journaling version of the ext2 file system. It is easy to switch back and forth between ext2 and ext3.

ext4
is a set of upgrades to ext3 including substantial performance and reliability enhancements, plus large increases in volume, file, and directory size limits.

Reiserfs
is a journaling file system, designed by Hans Reiser, that was integrated into Linux in kernel 2.4.1.

XFS
is a journaling file system, developed by SGI, that was integrated into Linux in kernel 2.4.20.

JFS
is a journaling file system, developed by IBM, that was integrated into Linux in kernel 2.4.24.

xiafs
was designed and implemented to be a stable, safe file system by extending the Minix file system code. It provides the basic most requested features without undue complexity. The **xia** file system is no longer actively developed or maintained. It was removed from the kernel in 2.1.21.

msdos
is the file system used by DOS, Windows, and some OS/2 computers. **msdos** filenames can be no longer than 8 characters, followed by an optional period and 3 character extension.

umsdos
is an extended DOS file system used by Linux. It adds capability for long filenames, UID/ GID, POSIX permissions, and special files (devices, named pipes, etc.) under the DOS file system, without sacrificing compatibility with DOS.

vfat
is an extended DOS file system used by Microsoft Windows95 and Windows NT. VFAT adds the capability to use long filenames under the MSDOS file system.

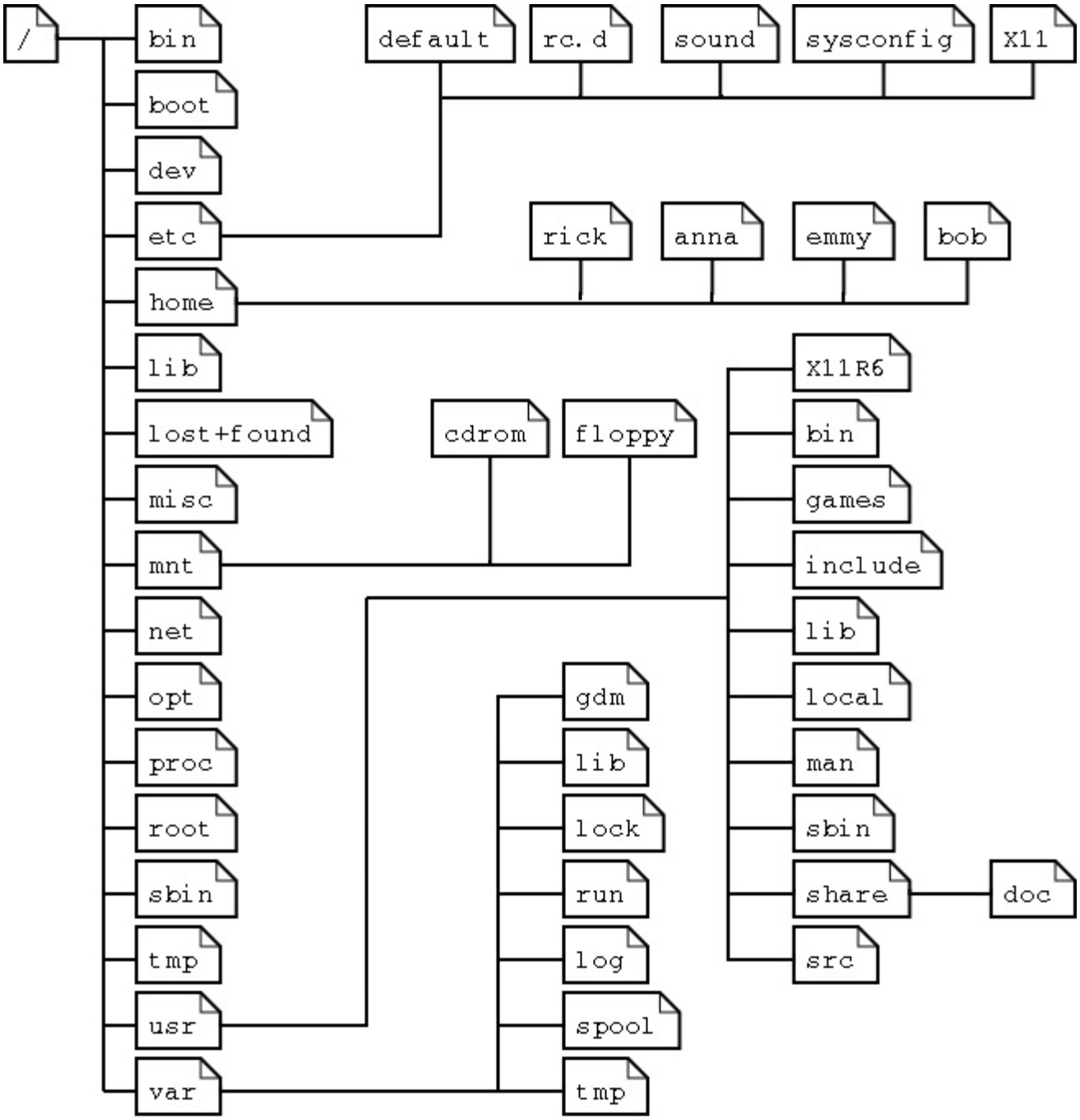
ntfs
replaces Microsoft Window's FAT file systems (VFAT, FAT32). It has reliability, performance, and space-utilization enhancements plus features like ACLs, journaling, encryption, and so on.

proc
is a pseudo file system which is used as an interface to kernel data structures rather than reading and interpreting */dev/kmem*. In particular, its files do not take disk space. See**proc**(5).

iso9660
is a CD-ROM file system type conforming to the ISO 9660 standard.

LINUX DIRECTORY TREE

Before going into filesystems, we went through an overview of the Linux directory tree which is laid out by the Filesystem Hierarchy Standard (FHS). This is done so that programs can be easily ported from one system to another and also so that the user knows where to find the pertinent files.



Linus has four major filesystems. But this list isn't exhaustive, one can always group filesystem in to one if the machine is essentially run by a single user or the filesystem can be increased if the machine is run by multiple users. The four major filesystems are:

1) Root filesystem

2) User Filesystem

3) Var filesystem

4) Home filesystem

root filesystem

This is a very small less mutated filesystem. It major contains the boot files and certain commands which are necessary during the boot process or by the root in case of emergencies. The major components of this filesystem are:

/bin	Common programs for users and system administrators.
/boot	Contains the boot images and GRUB configuration files
/dev	Device files
/etc	System configuration files.
/home	Personal files of the users
	/lib Contians the necessary library files
	/lib/modules Contains the dynamically loadable modules (driver modules)
	/lost+found Present in every partition, contains files which were unsaved during system crash.
	/mnt Standard mount mount point for media devices.
	/net Standard mount point for network devices.
	/proc Illusionary filesystem created by kernel to keep information about processes and the system resources.
	/root Home directory of the root user.
/sbin	Programs for use by the sysadmin and the system.
/tmp	Directory for temporary files.
/usr	Unchanging files like commands, documentation, man pages.
/var	Variable files like cache, spool, logs etc

Some of the directories in the root filesystem are very important and further information about them has been given below:

/etc

The /etc directory contains a listing of very important configuration files, some of which have been listed below:

/etc/rc?.d	Scripts or directories of scripts to run at startup or when changing the run level.
/etc/passwd	The user database, with fields giving the username, real name, home directory, encrypted password, and other information about each user. The format is documented in the passwd manual page. The encrypted passwords are much more commonly found in the /etc/shadow these days. This means that almost everything about the user except the password is stored in the passwd file. History and convention make a name change undesirable.
/etc/fstab	Lists the filesystems mounted automatically at startup by the mount −a command (in /etc/rc or equivalent startup file). Under Linux, also contains information about swap areas used automatically by swapon −a.
/etc/group	Similar to /etc/passwd, but describes groups instead of users.
/etc/issue	Output by getty before the login prompt. Usually contains a short description or welcoming message to the system. The contents are up to the system administrator.
/etc/magic	The configuration file for file command.
/etc/motd	Message of the day.
/etc/mtab	List of currently mounted filesystems. Initially set up by the bootup scripts, and updated automatically by the mount command. Used when a list of mounted filesystems is needed, e.g., by the **df** command.
/etc/shadow	Shadow password file on systems with shadow password software installed. Shadow passwords move the encrypted password from /etc/passwd into /etc/shadow; the latter is not readable by anyone except root. This makes it harder to crack passwords. If your distribution gives you a choice (many do) of whether or not to use shadow passwords then you are highly recommended to do so.
/etc/login.defs	Configuration file for the login command.
/etc/profile, /etc/csh.login, /etc/csh.cshrc	Files executed at login or startup time by the Bourne or C shells. These allow the system

	administrator to set global defaults for all users.
/etc/securetty	Identifies secure terminals, i.e., the terminals from which root is allowed to log in. Typically only the virtual consoles are listed, so that it becomes impossible (or at least harder) to gain superuser privileges by breaking into a system over a modem or a network. Do not allow root logins over a network. Prefer to log in as an unprivileged user and use su or sudo to gain root privileges.
/etc/shells	Lists trusted shells. The chsh command allows users to change their login shell only to shells listed in this file. ftpd, the server process that provides FTP services for a machine, will check that the user's shell is listed in /etc/shells and will not let people log in unless the shell is listed there.
/etc/apache	Configuration files for the Apache web server.
/etc/bashrc	The system-wide configuration file for the Bourne Again SHell. Defines functions and aliases for all users. Other shells may have their own system-wide config files, like cshrc.
*/etc/crontab and the /etc/cron.*directories*	Configuration of tasks that need to be executed periodically - backups, updates of the system databases, cleaning of the system, rotating logs etc.
/etc/default	Default options for certain commands, such as useradd.
/etc/filesystems	Known file systems: ext3, vfat, iso9660 etc.
/etc/fstab	Lists partitions and their mount points.
/etc/ftp*	Configuration of the ftp-server: who can connect, what parts of the system are accessible etc.
/etc/hosts	A list of machines that can be contacted using the network, but without the need for a domain name service. This has nothing to do with the system's network configuration, which is done in /etc/sysconfig.
/etc/inittab	Information for booting: mode, number of text consoles etc.
/etc/issue	Information about the distribution (release version and/or kernel info).
/etc/ld.so.conf	Locations of library files.

*/etc/logrotate.**	Rotation of the logs, a system preventing the collection of huge amounts of log files.
/etc/mail	Directory containing instructions for the behavior of the mail server.
/etc/modules.conf	Configuration of modules that enable special features (drivers).
/etc/motd	Message Of The Day: Shown to everyone who connects to the system (in text mode), may be used by the system admin to announce system services/maintenance etc.
/etc/mtab	Currently mounted file systems. It is advised to never edit this file.
/etc/nsswitch.conf	Order in which to contact the name resolvers when a process demands resolving of a host name.
/etc/pam.d	Configuration of authentication modules.
/etc/printcap	Outdated but still frequently used printer configuration file. Don't edit this manually unless you really know what you are doing.
/etc/profile	System wide configuration of the shell environment: variables, default properties of new files, limitation of resources etc.
*/etc/rc**	Directories defining active services for each run level.
/etc/resolv.conf	Order in which to contact DNS servers (Domain Name Servers only).
/etc/sendmail.cf	Main config file for the Sendmail server.
/etc/services	Connections accepted by this machine (open ports).
/etc/sound	Configuration of the sound card and sound events.
/etc/ssh	Directory containing the config files for secure shell client and server.
/etc/sysconfig	Directory containing the system configuration files: mouse, keyboard,network, desktop, system clock, power management etc.
/etc/X11	Settings for the graphical server, X. Also contains the general directions for the window managers available on the system, for example gdm, fvwm, twm, etc.
/etc/xinetd. or /etc/inetd.conf*	Configuration files for Internet services that

	are run from the system's (extended) Internet services daemon (servers that don't run an independent daemon).

/dev
The /dev directory contains a list of device files - all the computer peripherals excepting the CPU - which are automatically mounted by the bootprocess or the MAKEDEV script as instructed by the system administrator. The device files are essentially files which form an interface between the kernel and the device drivers.

A list of the common devices is:

/dev/cdrom	CD drive
/dev/console	Special entry for the currently used console.
*/etc/cua**	Serial ports
*/dev/ir**	Infrared devices
*/dev/js**	Joystick(s)
*/dev/lp**	Printers
/dev/mem	Memory
/dev/Modem	Modem
/dev/mouse	All kinds of mouses
/dev/null	Bottomless garbage can
*/dev/par**	Entries for parallel port support
*/det/pty**	Pseudo terminals
*/dev/radio**	For Radio Amateurs (HAMs).
*/dev/ram**	boot device
*/dev/sd**	SCSI and IDE disks with their partitions
*/dev/tty**	Virtual consoles simulating vt100 terminals.
*/dev/usb**	USB card and scanner
*/dev/video**	For use with a graphics card supporting video.

usr filesystem

The user filesystem is the filesystem which contains immutable files, that is the files which can be kept to be read only by the system administrator. This could as well be considered a secondary hierarchy similar to the root directory's hierarchy. It can be a locked filesystem common for all the users and kept in a network drive as well. It contains mainly the files installed by the distribution and some programs installed by the admin in the /usr/local directory. This way the admin knows that the program installed by him are not mutated by the distribution.

*/usr/X11**	This is the directory which contains the files for the X Windows system which implements

	the GUI in Linux. By default it is not incorporated in the kernel for flexibility and ease of development.
/usr/bin	Contains all the necessary common user commands.
/usr/sbin	Contains the system admin commands not listed in /sbin
/usr/local	Contains all the files installed by the system admin.
/usr/share	Contains man pages, GNU Info pages, and other Documentation.
/usr/include	Contains C header files.
/usr/lib	Contains the library files needed by the normal programs.

var filesystem

This is the filesystem which gets changed during the normal functioning of the system. It contains various directories like.

As a security measure, all the contents of the /var filesystem should be placed on a separate partition because it even contains contents that are accessible by anonymous users on the internet. This prevents the harm form the /var drive to get propagated to the other system critical partitions.

/var/cache	Contains the cache of various programs like apt, etc
/var/local	Contains the mutable files of the locally installed programs.
/var/lock	Programs create a lock file and store it here. The lock is continuously used by the program.
/var/tmp	Temporary files which are large are stored here rather than the root filesystem. Different from the /tmp directory as the /tmp directory can be directly modified by the system. (This may happen to this directory as well but is rare)
/var/spool	The queues are stored over here.
/var/mail	The mail box of the user is stored over here.
/var/run	The information about the current system is stored here.
/var/games	Files used by games is store over here.

proc Filesystem

This is an illusionary filesystem created by the kernel in the memory and not on the hard disk

and gives information about the systems health.

/proc/x A directory with information about process number x. Each process has a directory below /proc with the name being its process identification number.

*/proc/cpuinfo*Information about the processor, such as its type, make, model, and performance.

/proc/devices List of device drivers configured into the currently running kernel.

/proc/dma Shows which DMA channels are being used at the moment.

/proc/filesystems Filesystems configured into the kernel.

/proc/interrupts Shows which interrupts are in use, and how many of each there have been.

/proc/ioports Which I/O ports are in use at the moment.

/proc/kmsg Messages output by the kernel. These are also routed to syslog.

/proc/ksyms Symbol table for the kernel.

/proc/loadavg The `load average' of the system; three meaningless indicators of how much work the system has to do at the moment.

/proc/meminfo Information about memory usage, both physical and swap.

/proc/modules Which kernel modules are loaded at the moment.

/proc/net Status information about network protocols.

/proc/self A symbolic link to the process directory of the program that is looking at /proc. When two processes look at /proc, they get different links. This is mainly a convenience to make it easier for programs to get at their process directory.

/proc/stat Various statistics about the system, such as the number of page faults since the system was booted.

/proc/uptime The time the system has been

up.

*/proc/version*The kernel version.

To know which data partition, a particular filesystem belongs to; one issues the following command in that particular filesystem:

df -h .
df -h dir

SYSTEM IMPLEMENTATION OF A FILESYSTEM

Even though the Linux system can be viewed as partitions mounted onto a single directory tree, this is not the way the system implements the filesystems.

The partition associates an inode which is a serial associated with every file in this system - note that this inode is unique only in the partition and not elsewhere. This inode then describes a data structure which contains the following information for a standard file:

- Owner and group owner of the file.
- File type (regular, directory, ...)
- Permissions on the file Section 3.4.1
- Date and time of creation, last read and change.
- Date and time this information has been changed in the inode.
- Number of links to this file (see later in this chapter).
- File size
- An address defining the actual location of the file data.

An inode contains all this but not the name of the file or the directory to which the file belongs to. This is because the directory - which is a directory file - is a file which contains information in inode and filename pairs. Whenever we issue a ls command, the system reads the information about the files and the inodes from the directory file and lists it.

When a hard disk is initialized to accept data storage, usually during the initial system installation process or when adding extra disks to an existing system, a fixed number of inodes per partition is created according to the size of the partition. This number will be the maximum amount of files, of all types (including directories, special files, links etc.) that can exist at the same time on the partition.

There exists a block called the super-block which contains all the information about the number of allotted and free inodes. All the inodes are stored exactly next to the super-block. An inode exists in the memory as long as there are links to it in data structure, as soon as the number of links of the inode goes to zero, it is erased out of the memory.

So, all in all the inode is at the core of the filesystem because it tells the partition what is exactly stored in it. The information about the inode about a file can be obtained by issuing ls -i.

LINK FILES

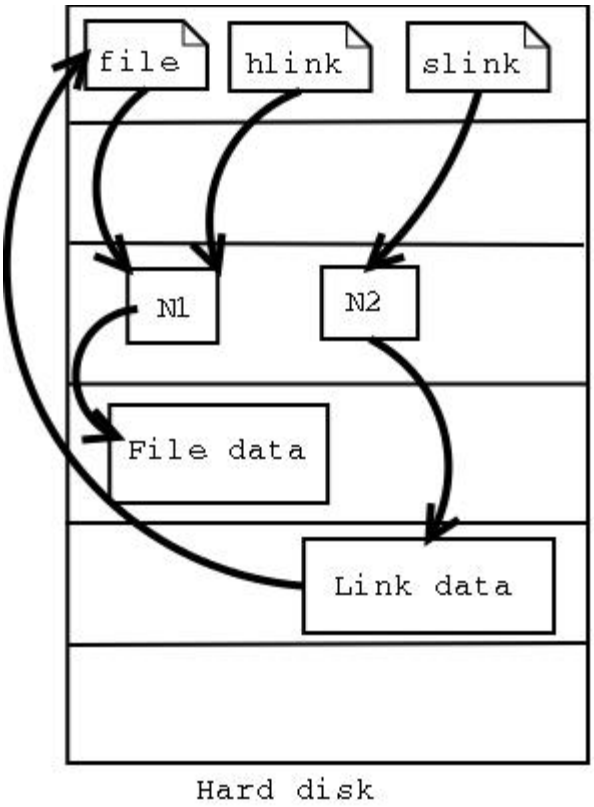
A link is nothing more than a way of matching two or more file names to the same set of file data. There are essentially three types of link files:

1. Hard Link: Associates two or more file names with the same inode. Hard links share the same data blocks on the hard disk, while they continue to behave as independent files. There is an immediate disadvantage: hard links can't span partitions, because inode numbers are only unique within a given partition.Each regular file is in principle

a hardlink

2. Soft link or symbolic link (or for short: symlink): a small file that is a pointer to another file. A symbolic link contains the path to the target file instead of a physical location on the hard disk. Since inodes are not used in this system, soft links can span across partitions. Removing the target link of a symlink renders it useless.

The two link types behave similar, but are not the same, as illustrated in the scheme below:



.

We can also say that theer is a third type of link called the user-space link, which is similar to a shortcut in MS Windows. These are files containing meta-data which can only be interpreted by the graphical file manager. To the kernel and the shell these are just normal files.

The syntax to create a symbolic link is very easy:

ln -s targetfile directory name

When we list the contents of a directory by using ls -l the output gives us the number of hard links the file has. The info about the hard links of a file is also stored in the inode of the file. A file is deleted or removed from the hard disk when the number of hard disks to that file becomes zero. It is then that the space associated with that file is freed for the used by the system.

PROCESSES

After files, the most important component in Linux, are undoubtedly processes. Linux being based on UNIX, has multiple users logged into the system and using the system resources simultaneously. So, measures have been taken so that the CPU can manages theses processes efficiently and there is enough flexibility that the user can control process handling as well. In some cases, processes will have to continue to run even when the user who

started them logs out.

PROCESS TYPES

Interactive processes

Interactive processes are initialized by the user manually and controlled by him as well through a terminal session. In other words, there has to be someone connected to the system to start these processes; they aren't auto started. These process have the ability to run in the foreground - where the user has to maintain a constant vigilance on the process and provide it with the information it requires. Or they may be run in the background - this allows the user to do some other work while the program/command runs. Only processes that do not require user intervention should be run in the background.

A process which runs in the foreground occupies the terminal and there after the terminal can only understand those commands which the running process can understand. While a process running in the background allows the user to perform other action while the process is being carried out.

The shell offers a feature called job control which allows easy handling of multiple processes. This mechanism switches processes between the foreground and the background. Putting a job in the background is typically done when execution of a job is expected to take a long time.

Using this system, programs can also be started in the background immediately. In order to free the issuing terminal after entering the command, a trailing ampersand is added.

billy:~> xterm &
[1] 26558

billy:~> jobs
[1]+ Running
xterm &

Some important commands in this line of thought are:

Command	Meaning
command_name	starts the command in the foreground
command_name &	starts command in the background
jobs	shows a list of 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.
%n	Every process running in the background gets a number assigned to it. By using the % expression a job can be referred to using its number, for instance fg %2.
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 the terminal; they are of two types on the basis of their execution:

- Executed at a particular date or time or scheduled to repeat at specific intervals. The former is accomplised using the at command while the latter is done using the cron command.
- Executed when the system load is low enough to accept extra jobs: done using the **batch** command. By default, tasks are put in a queue where they wait to be executed until the system load is lower than 0.8. In large environments, the system administrator may prefer batch processing when large amounts of data have to be processed or when tasks demanding a lot of system resources have to be executed on an already loaded system. Batch processing is also used for optimizing system performance.

Automatic processes can be queued into a spooler area, where they wait to be executed on a FIFO (first-in, first-out) basis.

Daemons
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.
- **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. The real owner is the user issuing the command, the effective user is the one determining access to system resources. RUID and EUID are usually the same, and the process has the same access rights the issuing user would have.
- **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. The effective group owner is usually the same, except when SGID access mode has been applied to a file.

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.

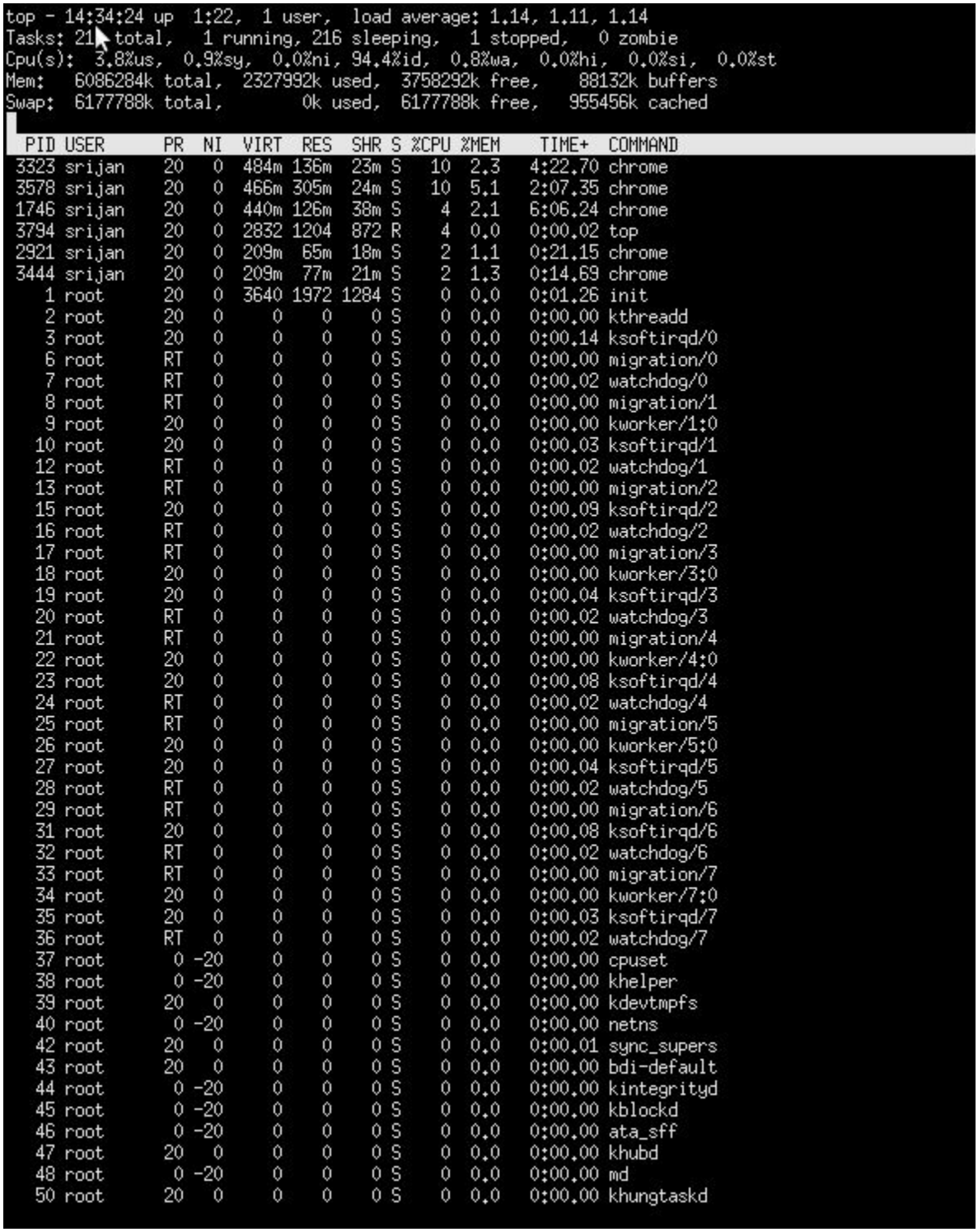
With no options specified, ps only gives information about the current shell and the processes started by it.

theo:~> ps

PID	*TTY*	*TIME*	*CMD*
4245	*pts/*	*700:00:00*	*bash*
5314	*pts/*	*700:00:00*	*ps*

This usually oes not give the required information and sice the normal ps -e command lists hundreds of processes which are running in the system, usually the grep command in a pipe, is used to select processes.

More info can be found the usual way: ps --help or man ps. GNU ps supports different styles of option formats; note that ps only gives a momentary state of the active processes, it is a one-time recording. The top program displays a more precise view by updating the results given by ps (with a bunch of options) once every five seconds, generating a new list of the processes causing the heaviest load periodically, meanwhile integrating more information about the swap space in use and the state of the CPU, from the proc file system:



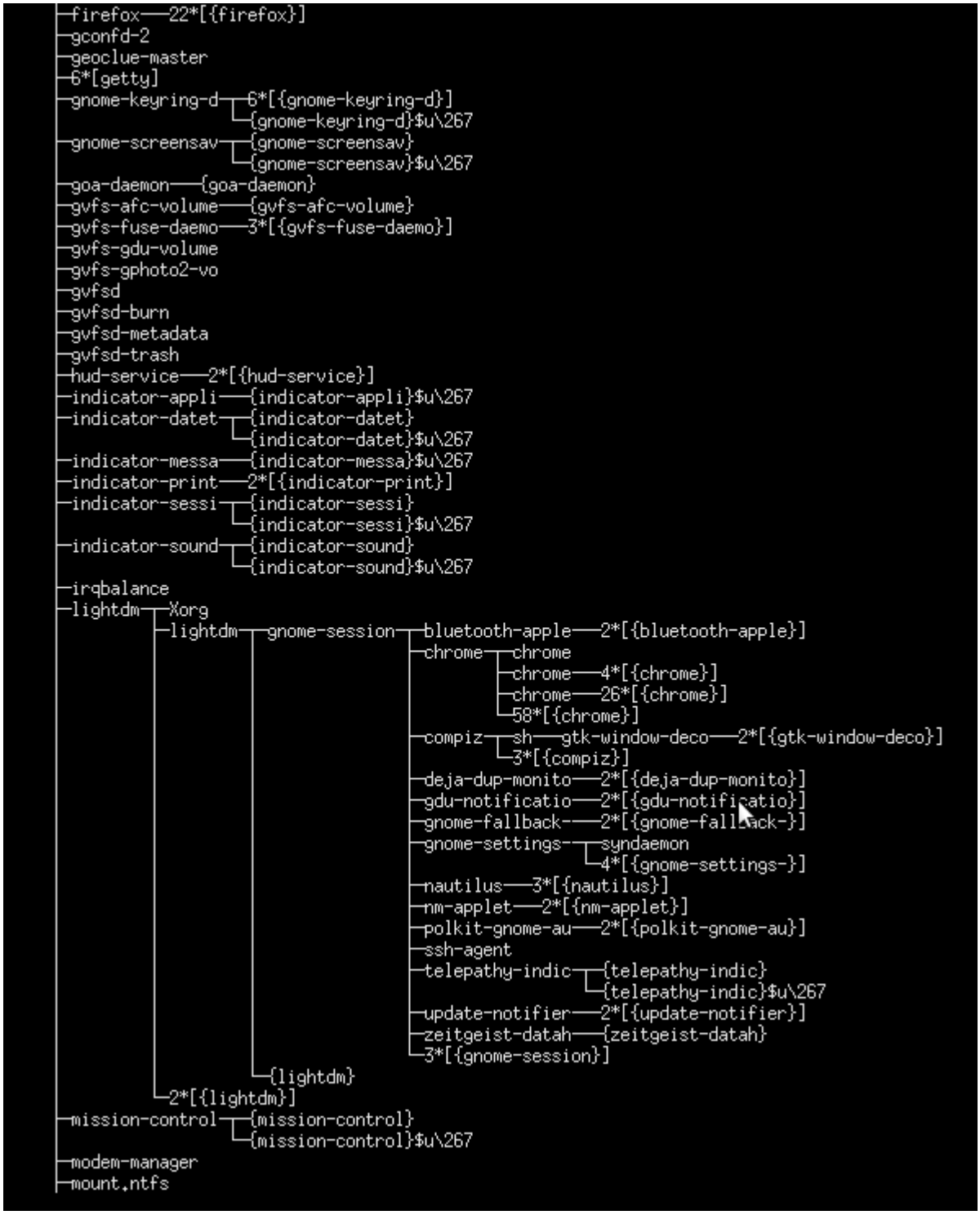
The first line of top contains the same information displayed by the uptime command:

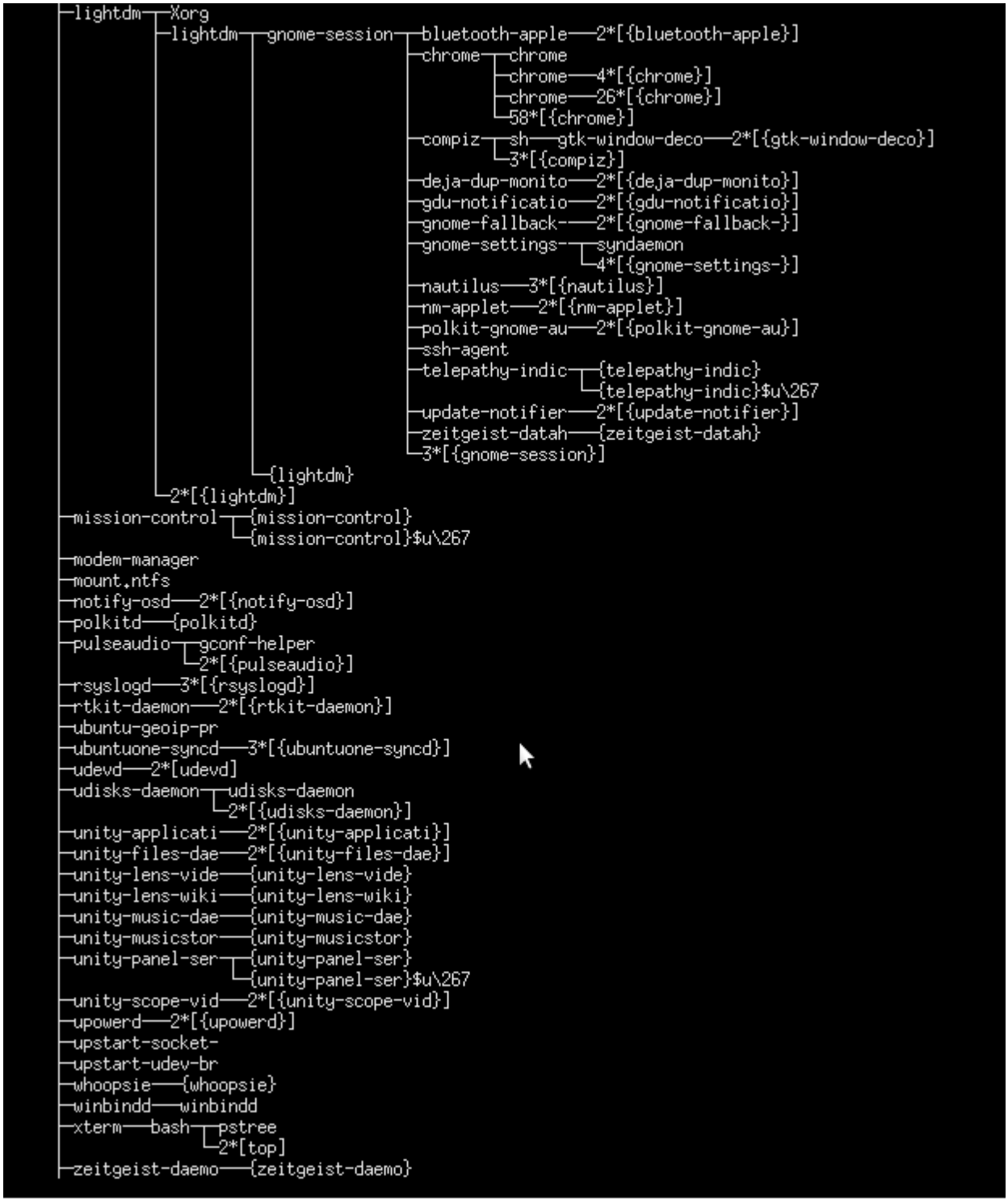
jeff:~> uptime
3:30pm, up 12 days, 23:29, 6 users, load average: 0.01, 0.02, 0.00

The data for these programs is stored among others in /var/run/utmp (information about currently connected users) and in the virtual file system /proc, for example /proc/loadavg (average load information). There are all sorts of graphical applications to view this data, such as the Gnome System Monitor and lavaps. Over at FreshMeat and SourceForge you will find tens of applications that centralize this information along with other server data and logs from multiple servers on one (web) server, allowing monitoring of the entire IT infrastructure from one workstation.

The relations between processes can be visualized using the pstree command:


init─┬─NetworkManager─┬─dnsmasq
└─2*[{NetworkManager}]
├─accounts-daemon───{accounts-daemon}
├─acpid
├─at-spi-bus-laun──2*[{at-spi-bus-laun}]
├─atd
├─avahi-daemon──avahi-daemon
├─bamfdaemon──2*[{bamfdaemon}]
├─bluetoothd
├─chrome-sandbox──chrome─┬─22*[chrome──3*[{chrome}]]
├─2*[chrome──4*[{chrome}]]
├─chrome──2*[{chrome}]
├─chrome──7*[{chrome}]
└─nacl_helper_boo
├─colord──2*[{colord}]
├─console-kit-dae─┬─63*[{console-kit-dae}]
└─{console-kit-dae}$u\267
├─cron
├─cupsd
├─2*[dbus-daemon]
├─dbus-launch
├─dconf-service──2*[{dconf-service}]
├─duckduckgo-lens──{duckduckgo-lens}
├─evince──3*[{evince}]
├─evinced──{evinced}
├─firefox──22*[{firefox}]
├─gconfd-2
├─geoclue-master
├─6*[getty]
├─gnome-keyring-d─┬─6*[{gnome-keyring-d}]
└─{gnome-keyring-d}$u\267
├─gnome-screensav─┬─{gnome-screensav}
└─{gnome-screensav}$u\267
├─goa-daemon──{goa-daemon}
├─gvfs-afc-volume──{gvfs-afc-volume}
├─gvfs-fuse-daemo──3*[{gvfs-fuse-daemo}]
├─gvfs-gdu-volume
├─gvfs-gphoto2-vo
├─gvfsd
├─gvfsd-burn
├─gvfsd-metadata
├─gvfsd-trash
├─hud-service──2*[{hud-service}]
├─indicator-appli──{indicator-appli}$u\267
├─indicator-datet─┬─{indicator-datet}
└─{indicator-datet}$u\267
├─indicator-messa──{indicator-messa}$u\267
├─indicator-print──2*[{indicator-print}]
├─indicator-sessi─┬─{indicator-sessi}
└─{indicator-sessi}$u\267
├─indicator-sound─┬─{indicator-sound}
└─{indicator-sound}$u\267
├─irqbalance
├─lightdm─┬─Xorg
├─lightdm─┬─gnome-session─┬─bluetooth-apple──2*[{bluetooth-apple}]
├─chrome─┬─chrome


├─firefox──22*[{firefox}]
├─gconfd-2
├─geoclue-master
├─6*[getty]
├─gnome-keyring-d─┬─6*[{gnome-keyring-d}]
│ └─{gnome-keyring-d}$u\267
├─gnome-screensav─┬─{gnome-screensav}
│ └─{gnome-screensav}$u\267
├─goa-daemon──{goa-daemon}
├─gvfs-afc-volume──{gvfs-afc-volume}
├─gvfs-fuse-daemo──3*[{gvfs-fuse-daemo}]
├─gvfs-gdu-volume
├─gvfs-gphoto2-vo
├─gvfsd
├─gvfsd-burn
├─gvfsd-metadata
├─gvfsd-trash
├─hud-service──2*[{hud-service}]
├─indicator-appli──{indicator-appli}$u\267
├─indicator-datet─┬─{indicator-datet}
│ └─{indicator-datet}$u\267
├─indicator-messa──{indicator-messa}$u\267
├─indicator-print──2*[{indicator-print}]
├─indicator-sessi─┬─{indicator-sessi}
│ └─{indicator-sessi}$u\267
├─indicator-sound─┬─{indicator-sound}
│ └─{indicator-sound}$u\267
├─irqbalance
├─lightdm─┬─Xorg
│ ├─lightdm─┬─gnome-session─┬─bluetooth-apple──2*[{bluetooth-apple}]
│ │ ├─chrome─┬─chrome
│ │ │ ├─chrome──4*[{chrome}]
│ │ │ ├─chrome──26*[{chrome}]
│ │ │ └─58*[{chrome}]
│ │ ├─compiz─┬─sh──gtk-window-deco──2*[{gtk-window-deco}]
│ │ │ └─3*[{compiz}]
│ │ ├─deja-dup-monito──2*[{deja-dup-monito}]
│ │ ├─gdu-notificatio──2*[{gdu-notificatio}]
│ │ ├─gnome-fallback-──2*[{gnome-fallback-}]
│ │ ├─gnome-settings-─┬─syndaemon
│ │ │ └─4*[{gnome-settings-}]
│ │ ├─nautilus──3*[{nautilus}]
│ │ ├─nm-applet──2*[{nm-applet}]
│ │ ├─polkit-gnome-au──2*[{polkit-gnome-au}]
│ │ ├─ssh-agent
│ │ ├─telepathy-indic─┬─{telepathy-indic}
│ │ │ └─{telepathy-indic}$u\267
│ │ ├─update-notifier──2*[{update-notifier}]
│ │ ├─zeitgeist-datah──{zeitgeist-datah}
│ │ └─3*[{gnome-session}]
│ └─{lightdm}
│ └─2*[{lightdm}]
├─mission-control─┬─{mission-control}
│ └─{mission-control}$u\267
├─modem-manager
├─mount.ntfs



The -u and -a options give additional information.

LIFE AND DEATH OF PROCESSES

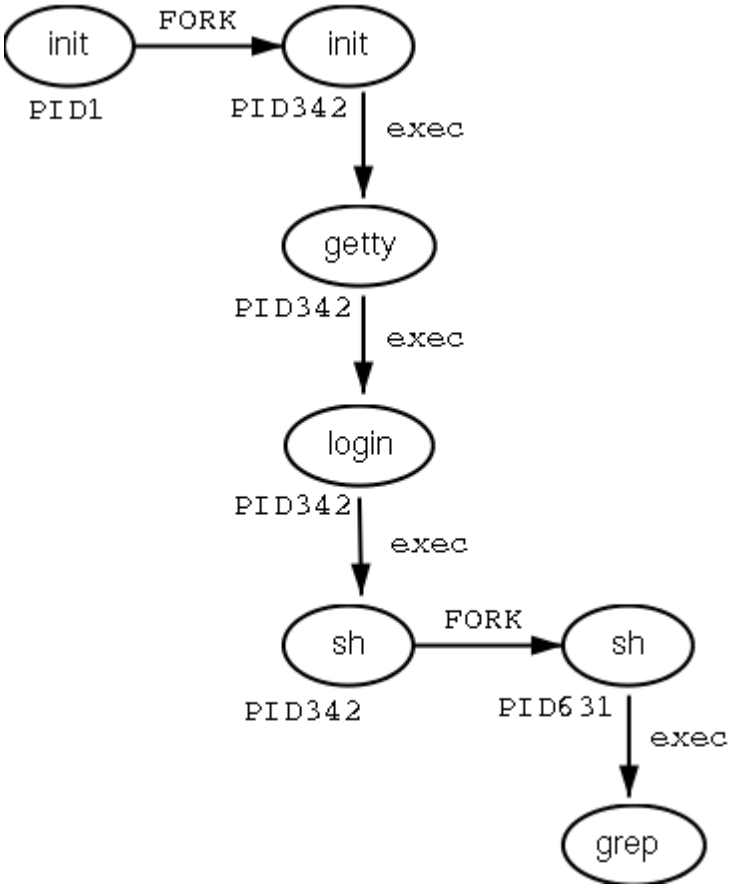
Process creation

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.

The fork-and-exec mechanism thus switches an old command with a new, while

the environment in which the new program is executed remains the same, including configuration of input and output devices, environment variables and priority. This mechanism is used to create all UNIX processes, so it also applies to the Linux operating system. Even the first process, init, with process ID 1, is forked during the boot procedure in the so-called bootstrapping procedure.

This scheme illustrates the fork-and-exec mechanism. The process ID changes after the fork procedure:



There are a couple of cases in which init becomes the parent of a process, while the process was not started by init. Many programs, for instance, daemonize their child processes, so they can keep on running when the parent stops or is being stopped. A window manager is a typical example; it starts an xterm process that generates a shell that accepts commands. The window manager then denies any further responsibility and passes the child process to init. Using this mechanism, it is possible to change window managers without interrupting running applications.

Every now and then things go wrong, even in good families. In an exceptional case, a process might finish while the parent does not wait for the completion of this process. Such an unburied process is called a zombie process.

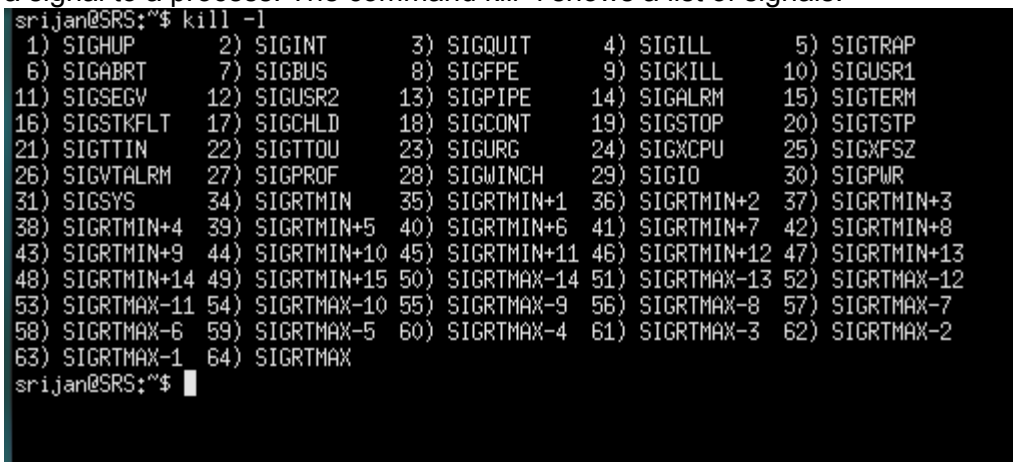
Ending processes
When a process ends normally (it is not killed or otherwise unexpectedly interrupted), the program returns its exit status to the parent. This exit status is a number returned by the program providing the results of the program's execution. The system of returning information upon executing a job has its origin in the C programming language in which UNIX has been written.

The return codes can then be interpreted by the parent, or in scripts. The values of the return

codes are program-specific. This information can usually be found in the man pages of the specified program, for example the grep command returns -1 if no matches are found, upon which a message on the lines of "No files found" can be printed. Another example is the Bash built-in command true, which does nothing except return an exit status of 0, meaning success.

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:

Signal name	Signal number	Meaning
SIGTERM	15	Terminate the process in an orderly fashion.
SIGINT	2	Interrupt the process. Overidable.
SIGKILL	9	Interuppt the process. Not overidable
SIG	1	For daemons: reread the configuration file.

FILE SECURITY

Linux derives it file security measures from UNIX (which are pretty robust). In fact the file security measures of Linux are even more robust than that of UNIX.

Every file in Linux has file permissions for the user who owns the file, the user group that owns the file and the other users. These permissions can be see by issuing the ls -l command in the directory where the file exists or by reading the inode of the file.

marise:~> ls -l To_Do
-rw-rw-r-- 1 marise users 5 Jan 15 12:39 To_Do

marise:~> ls -l /bin/ls
*-rwxr-xr-x 1 root root 45948 Aug 9 15:01 /bin/ls**

The first character in this listing tells us about the type of file as has been already discussed while the next nine characters talk about the file permissions. The first three characters are

for the user who owns the file, the next three for the owner group and the rest three for other users. All permissions are listed as read, write and execute.

For easy use with commands, both access rights or modes and user groups have a code. See the tables below.

Code	Meaning
0 or -	The access right that is supposed to be on this place is not granted.
4 or r	read access is granted to the user category defined in this place
2 or w	write permission is granted to the user category defined in this place
1 or x	execute permission is granted to the user category defined in this place

Access mode codes

The codes for the user, group and others are:

Code	Meaning
u	User permissions
g	Group permissions
o	Permissions for others

User group codes

To know about your user name, user group and all, one can issue the id command:

tilly:~> id
uid=504(tilly) gid=504(tilly) groups=504(tilly),100(users),2051(org)

The chmod command
The chmod command is one that is most certainly used a lot by normal users. It has two modes to change the permissions of a a file. Th alphanumeric method and the numeric method. The example below uses alphanumeric options in order to solve a problem that commonly occurs with new users.

asim:~> ./hello
bash: ./hello: bad interpreter: Permission denied

asim:~> cat hello
#!/bin/bash
echo "Hello, World"

asim:~> ls -l hello
-rw-rw-r-- 1 asim asim 32 Jan 15 16:29 hello

asim:~> chmod u+x hello

asim:~> ./hello
Hello, World

asim:~> ls -l hello
*-rwxrw-r-- 1 asim asim 32 Jan 15 16:29 hello**

The + and - operators are used to grant or deny a given right to a given group. The permissions of users, groups and other can be changed simultaneously by using commas; plus common permissions can be set by using a. Here's another one, which makes the file from the previous example a private file to user asim:

asim:~> chmod u+rwx,go-rwx hello
asim:~> ls -l hello
*-rwx------ 1 asim asim 32 Jan 15 16:29 hello**

Problems with an error message which says permission not denied is usually associated with file permissions; also, comments like, "It worked yesterday," and "When I run this as root it works," are most likely caused by the wrong file permissions.

When using chmod in th numeric mode, we use the following command:

chmod xyz filename

Here xyz is a three digit number where x signifies the permission of the owner, y the permission of the group and z or everyone else.(There is actually a fourth digit on Linux systems, that precedes the first three and sets special access modes, which is discussed later). The number x/y/z is obatained by adding individual permissions for example, 7 is read, write and execute permission and so on. Below is a list of common permissions:

Command	**Meaning**
chmod 400 file	To protect a file against accidental overwriting.
chmod 500 directory	To protect yourself from accidentally removing, renaming or moving files from this directory.
chmod 660 file	Users belonging to your group can change this file, others don't have any access to it at all.
chmod 700 file	Protects a file against any access from other users, while the issuing user still has full access.
chmod 644 files	A publicly readable file that can only be changed by the issuing user.
chmod 600 file	A private file only changeable by the user who entered this command.

chmod 755 directory	For files that should be readable and executable by others, but only changeable by the issuing user.
chmod 775 file	Standard file sharing mode for a group.
chmod 777 file	Everybody can do everything to this file.

Logging on to another group
We use the id command to view our UID, primary GID and a list of all other groups to which we belong to. Some Linux version implement a scheme where the user has only one active primary group, which is the group to which he is logged into. By default, this active or primary group is the one that you get assigned from the /etc/passwd file. The fourth field of this file holds users' primary group ID, which is looked up in the /etc/group file. An example:

asim:~> id
uid=501(asim) gid=501(asim) groups=100(users),501(asim),3400(web)

asim:~> grep asim /etc/passwd
asim:x:501:501:Asim El Baraka:/home/asim:/bin/bash

asim:~> grep 501 /etc/group
asim:x:501:

The fourth field in the line from /etc/passwd contains the value "501", which represents the group asim in the above example. From /etc/group we can get the name matching this group ID. When initially connecting to the system, this is the group that asim will belong to.

Linux these days uses a User Private group scheme where every user is alloted a private user group with the same name as his username and containing only the user as the member of this group. This provides additional security to the content of the user as by default, the group which can acess his data is his own private user group.

Apart from his own private group, user asim can also be in the groups users and web. Because these are secondary groups to this user, he will need to use the newgrp to log into any of these groups (use gpasswd for setting the group password first). In the example, asim needs to create files that are owned by the group web.

asim:/var/www/html> newgrp web

asim:/var/www/html> id
uid=501(asim) gid=3400(web) groups=100(users),501(asim),3400(web)

When asim creates new files now, they will be in group ownership of the group web instead of being owned by the group asim:

asim:/var/www/html> touch test

asim:/var/www/html> ls -l test
-rw-rw-r-- 1 asim web 0 Jun 10 15:38 test

The file mask
Linux uses a system function to create new files (that includes downloading files from the internet or copying files). This function creates both files and directories ans assigns them

with default permission; which are full read,write and execute permissions for directories and full read and write permissions for files.

But whenever this new file is saved, a mask is applied to it, that is a fixed set of permissions are taken away from this file, this is how the standard security procedure is apllied. Files without permissions don't exist on Linux. The value of this mask can be displayed using the umask command:

bert:~> umask
0002

In the example above, however, we see 4 values displayed, yet there are only 3 permission categories: user, group and other. The first zero is part of the special file attributes settings. It might just as well be that this first zero is not displayed on your system when entering the umask command, and that you only see 3 numbers representing the default file creation mask.

The umask value is subtracted from these default permissions after the function has created the new file or directory. Thus, a directory will have permissions of 775 by default, a file 664, if the mask value is (0)002.

This is demonstrated in the example below:

bert:~> mkdir newdir
bert:~> ls -ld newdir
drwxrwxr-x 2 bert bert 4096 Feb 28 13:45 newdir/

bert:~> touch newfile
bert:~> ls -l newfile
-rw-rw-r-- 1 bert bert 4096 Feb 28 13:45 newdir/

A directory always gets more permissions, it always has the execute permission because without that, we will not be able to do stuff like change directory etc.

If you log in to another group using the newgrp command, the mask remains unchanged. Thus, if it is set to 002, files and directories that you create while being in the new group will also be accessible to the other members of that group; you don't have to use chmod.

The root user usually has stricter default file creation permissions:

[root@estoban root]# umask
022

These defaults are set system-wide in the shell resource configuration files, for instance /etc/ bashrc or /etc/profile.

Changing user and group ownership
The chgrp and chown commands can be used to change the owners of files and directories. The chown command is more flexible and allows both the group and the owner to be changed while the chgrp only changes the owner. But an important point ot keep in mind is that to run these commands, the user has to have adequate permissions.

In order to only change the user ownership of a file, use this syntax:

chown newuser file

If the name of the user if followed by a colon, then the group of the file is also changed and is set to the primary group of the user.

```
jacky:~> id
uid=1304(jacky) gid=(1304) groups=1304(jacky),2034(pproject)

jacky:~> ls -l my_report
-rw-rw-r-- 1     jacky    project          29387          Jan 15          09:34          my_report

jacky:~> chown jacky: my_report
jacky:~> chmod o-r my_report
jacky:~> ls -l my_report
-rw-rw---- 1 jacky       jacky            29387          Jan 15          09:34          my_report
```

If jacky would like to share this file, without having to give everybody permission to write it, he can use the chgrp command:

```
jacky:~> ls -l report-20020115.xls
-rw-rw---- 1 jacky       jacky   45635          Jan 15          09:35   report-20020115.xls

jacky:~> chgrp project report-20020115.xls
jacky:~> chmod o= report-20020115.xls
jacky:~> ls -l report-20020115.xls
-rw-rw---- 1 jacky       project 45635          Jan 15          09:35   report-20020115.xls
```

Using the -r option, the permission changes made by chgrp and chown are carried forth recursively to all the underlying sub directories and files of a directory.

Special modes
For the system admin to not be bothered solving permission problems all the time, special access rights can be given to entire directories, or to separate programs. There are three special modes:

- **Sticky bit mode:** When applied to an entire directory, however, the sticky bit has a different meaning. In that case, a user can only change files in this directory when she is the user owner of the file or when the file has appropriate permissions. This feature is used on directories like /var/tmp, that have to be accessible for everyone, but where it is not appropriate for users to change or delete each other's data. The sticky bit is indicated by a t at the end of the file permission field:
  mark:~> ls -ld /var/tmp
  drwxrwxrwt 19 root root 8192 Jan 16 10:37 /var/tmp/

  The sticky bit is set using the command chmod o+t directory. The historic origin of the "t" is in UNIX' save Text access feature.

- **SUID (set user ID) and SGID (set group ID):** represented by the character s in the user or group permission field. When this mode is set on an executable file, it will run with the user and group permissions on the file instead of with those of the user issuing the command, thus giving access to system resources.
- **SGID (set group ID) on a directory:** in this special case every file created in the directory will have the same group owner as the directory itself (while normal behavior would be that new files are owned by the users who create them). This way, users don't need to worry about file ownership when sharing directories:
  mimi:~> ls -ld /opt/docs
  drwxrws--- 4 root users 4096 Jul 25 2001 docs/

  mimi:~> ls -l /opt/docs
  -rw-rw---- 1 mimi users 345672 Aug 30 2001- Council.doc

  This is the standard way of sharing files in UNIX.

Existing files are left unchanged! Files that are being moved to a SGID directory but were created elsewhere keep their original user and group owner. This may be confusing.

SHELL

INTRODUCTION

A shell is the interface between the user and the kernel. It uses a terminal like interface to accept the commands from the user and then conveys them to the kernel. In a sense the kernel is like a language. While one may use the GUI language to interact with the compiler: this type of interaction is limited because the user has to choose from what the computer provides. While using a shell the user has flexibility and so the communication is more interactive because the user can do whatever he wants bringing in freedom to he user and making the shell very powerful.

TYPES OF SHELLS

The most common types of shells are:

1. Bourne Shell: This was and is the standard shell of many UNIX environments.
2. Bourne Again Shell: The bash shell is the standard shell for GNU. It is a very convenient shell for beginners and at the same time powerful enough for power users. It is a superset of the Bourne shell with plug-ins and addons.
3. C Shell: This is a shell which is based on the C programming language.
4. Turbo C Shell: A super set of the C shell adding in more user friendliness and other options.

The standard shell of a user can be changed by directly writing the name of the shell and pressing enter because after all a shell is an executable file. To know one's shell, one can write the following command in the terminal:

echo $SHELL

This command prints the environment variable SHELL which stores the name of the default shell.

NETWORKING

SETTING SYSTEM PROXY FOR APT

(This method is only applicable for apt) Without setting the system proxy in Linux, nothing can be done in Linux atleast in IITK. After setting the system proxy, APT and other such tools which require an internet access can be easily run.

To set the network proxy in a Linux system - which is a must for the average IITian - , we must edit the apt.conf file which is located in the /etc/apt/ directory. After opening this file using gksudo one has to add the following lines:

Acquire::http::proxy "http://user:pass@host:port/"
Acquire::https::proxy "http://user:pass@host:port/"

One can add similar lines for the various services used. This method not only adds a proxy but also provides the method for authentication.

After this step, a file should made in the /etc/apt/apt.conf/d directory named 80proxy with the

entire content of apt.conf copied.

BASH RC METHOD

This method involves the editing of the .bashrc file in $HOME directory. This method is useful for applying a http-proxy to applications like apt-get and other applications for instance wget.

gedit ~/.bashrc

Append the following lines to the *~/.bashrc* file (substitute details for proxyaddress and proxyport)

http_proxy=http://username:password@proxyaddress:proxyport
export http_proxy

Save the file. Close your terminal window and then open another terminal window or source the *~/.bashrc* file:

source ~/.bashrc

UBUNTU

This section of the documentation is mainly concerned with Ubutu, one of the most popular Linux Distributions.

REPOSITORIES

Repositories are online sites which contain softwares as binaries as well as source code of packages which can be downloaded directly using apt or similar package managers. Repositories provide an easy way - because they can be downloaded directly- as well as provide secure software because the software are thoroughly tested. According to the licenses and the support given to different software, there are the following repository components:

1. Main - Officially supported and free.
2. Restricted - Official supported but not completely free.
3. Universe - Community maintained but not officially supported.
4. Multiverse - Software that are not free

As we are focusing mainly on using the terminal, we will be using the apt to install, remove, upgrade and update the installed programs.

ADDING REPOSITORIES

Usually the system contains a list of repositories in the /etc/apt/sources.list file which can be edited to add extra repositories as per the requirement of the user. The tutorial for the same is available at https://help.ubuntu.com/community/Repositories/CommandLine

Before making any changes to the sources.list file, it is advisable to back it up with the command

sudo cp /etc/apt/sources.list /etc/apt/sources.list.backup

A standard sources.list looks like

 # sources.list
 # deb cdrom:[Ubuntu 8.04.1 _Hardy Heron_ - Release amd64 (20080701)]/ hardy main restricted

#deb cdrom:[Ubuntu 8.04.1 _Hardy Heron_ - Release amd64 (20080701)]/ hardy main restricted
 # See http://help.ubuntu.com/community/UpgradeNotes for how to upgrade to
 # newer versions of the distribution.

deb http://us.archive.ubuntu.com/ubuntu/ hardy main restricted
deb-src http://us.archive.ubuntu.com/ubuntu/ hardy main restricted

Major bug fix updates produced after the final release of the
distribution.
deb http://us.archive.ubuntu.com/ubuntu/ hardy-updates main restricted
deb-src http://us.archive.ubuntu.com/ubuntu/ hardy-updates main restricted

The lines starting with # and ## are comments. For the rest, the following should be known:

1. **deb** - means that this repository contains binaries and precompiled packages.
2. **deb-src** - means that this repository contains source code packages.
3. **http://us.archive.ubuntu.com/ubuntu/** - URL of the download location.
4. **main** - repository component
5. **hardy** - release name

The Universe and Multiverse can be enabled by uncommenting the corresponding apt line (i.e. delete the '#' at the beginning of the line). In this example, we would uncomment the following lines for the Universe:

deb http://us.archive.ubuntu.com/ubuntu/ hardy universe
deb-src http://us.archive.ubuntu.com/ubuntu/ hardy universe
deb http://us.archive.ubuntu.com/ubuntu/ hardy-updates universe
deb-src http://us.archive.ubuntu.com/ubuntu/ hardy-updates universe

For the Multiverse:

deb http://us.archive.ubuntu.com/ubuntu/ hardy multiverse
deb-src http://us.archive.ubuntu.com/ubuntu/ hardy multiverse
deb http://us.archive.ubuntu.com/ubuntu/ hardy-updates multiverse
deb-src http://us.archive.ubuntu.com/ubuntu/ hardy-updates multiverse

Hardy-updates is necessary for updates. After this we have to update apt.

sudo apt-get update

The partner repositories can be added by uncommenting the following lines in your /etc/apt/ sources.list file:

deb http://archive.canonical.com/ubuntu hardy partner
deb-src http://archive.canonical.com/ubuntu hardy partner

Then update as before:

sudo apt-get update

PPA

Ubuntu has a normal release cycle of 6 months, which means that a major release is made in every six months with changes in all system files and programs. While this ensures that all the new programs are thoroughly testes, yet this also means that the users cannot get hold

of new software which has been release by the developers. To tackle this, canonical allows the developers to host PPAs or Personal Package Achieves which are like repositories but hold one or more softwares. Including PPAs in the sources.list of Ubuntu, allows the user to get an always up to date changes in his favorite software. To add a ppa we use the standard command

sudo add-apt-repository ppa:ppaname

APT

Apt or advanced package tool is the command line tool used to install, update, upgrade, remove and clean the programs or packages installed in a Linux system. The tutorial for using apt can be seen from https://help.ubuntu.com/community/AptGet/Howto?action=show&redirect=AptGetHowto

Some excerpts from the tutorial are:Installation commands:

All of these commands must be run as root or with superuser privileges. Replace <package_name> with the name of the package you are attempting to install.

apt-get install <package_name> This command installs a new package.

apt-get build-dep <package_name> This command searches the repositories and installs the build dependencies for <package_name>. If the package is not in the repositories it will return an error.auto-apt

apt-get update Run this command after changing */etc/apt/sources.list* or */etc/apt/preferences* .. Run this command periodically to make sure your source list is up-to-date.

apt-get upgrade This command upgrades all installed packages. This is the equivalent of "Mark all upgrades" in Synaptic.

apt-get dist-upgrade It tells APT to use "smart" conflict resolution system, and it will attempt to upgrade the most important packages at the expense of less important ones if necessary.apt-get check

apt-get check This command is a diagnostic tool. It does an update of the package lists and checks for broken dependencies.

apt-get -f install Fixes broken packages.

apt-get autoclean This command removes .deb files for packages that are no longer installed on your system. Depending on your installation habits, removing these files from */var/cache/apt/archives* may regain a significant amount of diskspace.

apt-get clean The same as above, except it removes ***all*** packages from the package cache.

dpkg-reconfigure <package_name>
Reconfigure the named package. With many packages, you'll be prompted with some configuration questions you may not have known were there.

apt-get remove <package_name> This command removes an installed package, leaving configuration files intact.

apt-get purge <package_name> This command completely removes a package and the associated configuration files. Configuration files residing in ~ are not usually affected by this command.

apt-get autoremove This command removes packages that were installed by other packages and are no longer needed.

apt-get autoremove <package_name>
This command removes an installed package and dependencies.

apt-cache search <search_term> This command will find packages that include <search_term>.

apt-cache show <package_name> This command shows the description of package <package_name> and other relevant information including version, size, dependencies and conflicts.

apt-file search <package_name> It answers the question, "what package provides this file?".

apt-file needs to be updated regularly like apt-get. Use the command: apt-file update

apt-cache pkgnames This command provides a listing of every package in the system

METAPACKAGES

Metapackages are essentially, a file which contains a list of files or programs to be downloaded. A meta-package is installed in the normal way using the apt-get command but in this process it triggers a series of more apt-get commands which install all the listed files in the meta-package.

TASKSEL

Tasksel is a utility like apt which is used to install programs. Rather than installing programs, it starts a list of tasks; these tasks are essentially like metapackages. A task refers to installing a list of packages which constitute a bigger package like the GNOME Desktop, LAMP server, KDE environment etc.

COMPILING FROM SOURCE

While the normal user using Linux may never need to compile sources available on the internet, it is a very handy tool. The tutorial for learning how to compile from sources is available at: https://help.ubuntu.com/community/CompilingEasyHowTo.

The whole process is very simple and basically can be completed in three steps:

1. **./config:** This command is a pre compilation command where in the program checks whether it has all the things it needs to compile plus custom options by the user are also set into the compilation.
2. **make:** This is the command which compiles the source code and after this the program is ready to be installed into the system. An important point to note is that uptil now, no changes have been made into the system, all the changes have been local. So no root permission was required uptill now. The next step though makes changes into the system so requires root permissions.
3. **make install**: This command installs the compiled program into the system and after this the program is ready to be used. An optional step after this is executing the make clean command which cleans all the temporary files.

The problem with compiling from source is that one can go through dependency hell; which is when one has no clue what dependencies the program needs. Dependencies are programs/commands on which the program to be compiled depends; finding all such dependencies is pretty hectic and two tools particularly useful in this context are auto-apt and apt-file which can search in which package a particular dependency lies in.

ENVIRONMENTAL VARIABLES

PATH

http_proxy

HOME

USER

SHELL

APPENDIX

POSIX, an acronym for "**P**ortable **O**perating **S**ystem **I**nterface", is a family of standards specified by the IEEE for maintaining compatibility between operating systems. POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems.

A **patch** is a piece of software designed to fix problems with, or update a computer program or its supporting data. This includes fixing security vulnerabilities and other bugs, and improving the usability or performance. Though meant to fix problems, poorly designed patches can sometimes introduce new problems (see software regressions). In some special cases updates may knowingly break the functionality, for instance, by removing components for that the update provider is no longer licensed or disabling the jailbreaked device.

Standardization is the process of developing and implementing technical standards.

A core file or core dump is sometimes generated when things go wrong with a program

during its execution. The core file contains a copy of the system's memory, as it was at the time that the error occured.

EXPLORING LINUX OPERATING SYSTEM (KERNEL)

OS VS KERNEL

Generally people think that OS and kernel are the same thing but kernel is only a part of OS. Besides kernel, OS also consist boot loader , command shell or other user interface and basic file and system utilities.

Kernel is the innermost core of the system which acts as a link between user's application and hardware. It provides services like managing hardware,distributing system resources etc.

KERNEL: Typical components of kernel are interrupt handlers , device drivers, schedulers, memory management systems, and system services such as networking, and inter process communication.

A device driver is an interface between the actual device and the kernel. It acts as an abstraction class for a various number of hardware devices. They are used owing to the fact that there are many devices that essentially do the same task but belong to different companies or do it in a different way. Having device drivers makes the job of the kernel much easier.

GETTING KERNEL SOURCE CODE

GIT

By the very virtue of this project, we had to work on the Linux Kernel source code and edit it. We needed to maintain a version history so that if we got stuck at any stage, we could revert back to the previous working state. We decided to use git for this purpose, which is a Version Control System (VCS). To host our git repository, we used github which is a free website which allows us to store our repository online.

To learn about git, you can refer to the online study manual http://git-scm.com/book; setting up git on a linux system is also pretty easy, for the instructions to the same refer to http://help.github.com/linux-set-up-git/.

For the next step the team forked the Linux kernel repository from Torvalds's and made a clone on each of the systems we used for working. The instructions to doing so can be obtained from http://help.github.com/fork-a-repo/.

You can also make your own repository on git for which you can refer to this link http://help.github.com/create-a-repo/ . We did make our own test repository and practised basic functions made available by git like commiting, adding, pushing, pulling etc. Doing this really helps to get familiar with the git.

OTHER METHODS

Another method for getting the source code is to directly download it from the site kernel.org which is a site which maintains a list of all the released kernels. It is the official site of storing all the versions of the Linux kernel.

BUILDING- CONFIGURING AND COMPILING THE KERNEL

Building the kernel starts with configuring it. There are many ways to the kernel. The ones which we tried are mentioned here.

$ make config

This is a very long way of configuring but it is most compatible. This utility goes through each option one by one and asks user to interactively select yes, no or (for tristates) module. It requires some hours to configure by this method.

$ make defconfig

This will apply default settings but they may not be compatible with your hardware. So, although it is a good start as it creates the .config file and we can simply make changes there.

$ make oldconfig

It reads the existing .config file and prompts the user for options in the current kernel source that are not found in the file. This is useful when taking an existing configuration and moving it to a new kernel.

Once we have configured the kernel now we have to build it this can be done with a single command :

$ make

But, make takes a lot of time so we can assign the multiple jobs which will run in parallel and therefore reduces the time. The command for this is :

$ make -jn

Here “n” is the number of jobs you want to assign for the make.

After make we have the bzImage present in arch/x86/boot/bzImage which we now need to copy to /boot/Kernel-version . But here it is important to note that the path specified in above line depends on architecture and boot loader dependent it is given for our architechture.

After this we have to edit /boot/grub/grub.cfg file by adding a new entry for a new kernel.****

After this we have to install all the compiled modules for which the command is:

$ make modules_install.

GRUB TUTORIAL

Grub is the standard bootloader of Linux, the most recent version of it being Grub2. A bootloader is a software which resides in the volatile memory of the system(RAM) and is the first thing to be run when the system starts. The bootloader comprises a code and certain data which helps it to load the code of the OS/Kernel onto the RAM for normal functioning of the System.

There are various intricacies associated with the GRUB bootloader which have been

mentioned quite elegantly in the following article https://help.ubuntu.com/community/Grub2#A.2BAC8-etc.2BAC8-default.2BAC8-grub_.28file.29.

GRUB2 is a completely revamped bootloader with its configuration files residing in etc/default/grub, the scripts reside in etc/grub.d and the main configuration file lies in boot/grub/grub.cfg. When the update-grub command is run, the program runs the scripts in etc/grub.d which use the data in etc/default/grub to create the final grub.cfg file.

SYSTEM CALLS

INTRODUCTION

System Calls are the programs that run to provide services such as hardware and other resources to the user. They act as the interface between the user space and the operating system. They are the basic programs(around 320) in linux kernel 3.0 that performs task in the kernel space. The advantage of having system calls is that if a program is given full independence to access hardware and computer resources then it would lead to a total mess up.

We go through a number of system calls and made some programs using those.These system calls are:

1) fork():
This system call creates a child process. As this function is called from lets say "y" program then in parallel to the existing process a child process is also started which executes the program from below that "fork()" command.Now there are two process running in parallel one the parent one and other the child one. One may wonder what if, let there are some variables defined earlier in the program and are used after the fork command then how will the child process will implement that variable....answer to this question is that when the child process is called it's memory address remains the same as parent but separately spaced.

Eg: executing a simple code

```
#include <stdio.h>
int main()
{
        int i;
    i= fork();
        if(i==0) printf("%d\tCHILD\n",i);
        else printf("%d\tPARENT\n",i);
        return 0;
}
```

upon running this it generates the output:

```
2387 PARENT
0       CHILD
```

The fork() command returns the pid of child to parent and 0 to child.

2) access():
This system call tells the permission status of a file for the process calling this system call. Whether file exists and if exists then status about read,write and execute permissions.If return value is 0 then positive result else fail.

```
access(path,F_OK);  file existence
```

access(path,R_OK); readable
access(path,W_OK); writing
access(path,X_OK); executable

3) fcntl():
This system call serves the purpose for locking files. They may be read locked or write locked.Read lock can be implemented by several process but write lock is only implemented by a single process. Suppose, a file is been read by many users/process but a change can be made only by one at a time,if this is not so then it may create a havoc.

Set the l_type field of the structure “flock” to F_RDLCK for a read lock or F_WRLCK for a write lock.Then call fcntl, passing a file descriptor to the file, the F_SETLCKW operation code, and a pointer to the struct flock variable. If another process holds a lock that prevents a new lock from being acquired, fcntl blocks until that lock is released.

4) wait(&status):
This system call waits for the child process to terminate and then execute parent process.When we simply use fork() the thing is processor decides itself which process to execute first but wait(&status) call execute child process first.

5) execve(file name,argv[],envp):
Using this system call we can tell the file pointed by file path to execute using the argv[] as their variable entry.

6) fsync(fd):
Whenever we write something to a file what actually happen is first it is written to a disk buffer and then to the file. Therefore if somehow our system shut down suddenly(unexpectedly) our written data may not save to the file, therefore to remedy this fsync(fd) may be used because this system call directly saves it to hard drive.

7) getrlimit,setrlimit:
These system calls allows one to tell and set the amount of resource that can be used by a process ie. executable time,memory and other.

8) readdir,writedir:
“readdir” system calls is used to get a DIR type pointer to the next entry in a directory specified and NULL if no next entry exist. “writedir” system call returns a pointer to the specified directory.

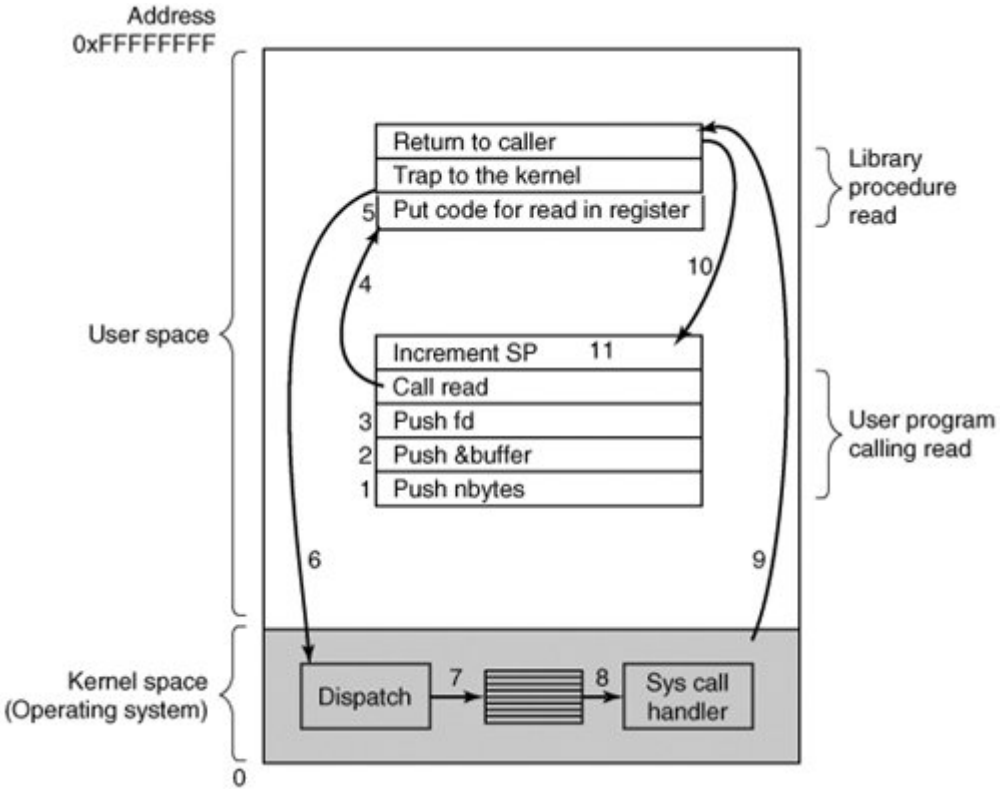
9)stat():
These functions return information about a file.

We explored many other system calls and implemented them.

Order Of Process Happening For A System Call

Let’s take the system call read(fd,buf*,count) for example. When our program calls a system call the arguments passed are stored in a stack in reverse order. After this the library procedure is called which gives the system call number and stores it in a register.Then it executes a TRAP instruction to switch from the user space to kernel space and start execution at a fixed address in kernel space.The kernel code that starts examines the system call number and then dispatches to the correct system call handler, usually via a

table of pointers to system call handlers indexed on system call number.At that point system call handler runs. After compliting it's work , control may be returned back to the user space using a similar TRAP instruction.To finish the job, the user has to clean up the stack, as it does after any procedure call.



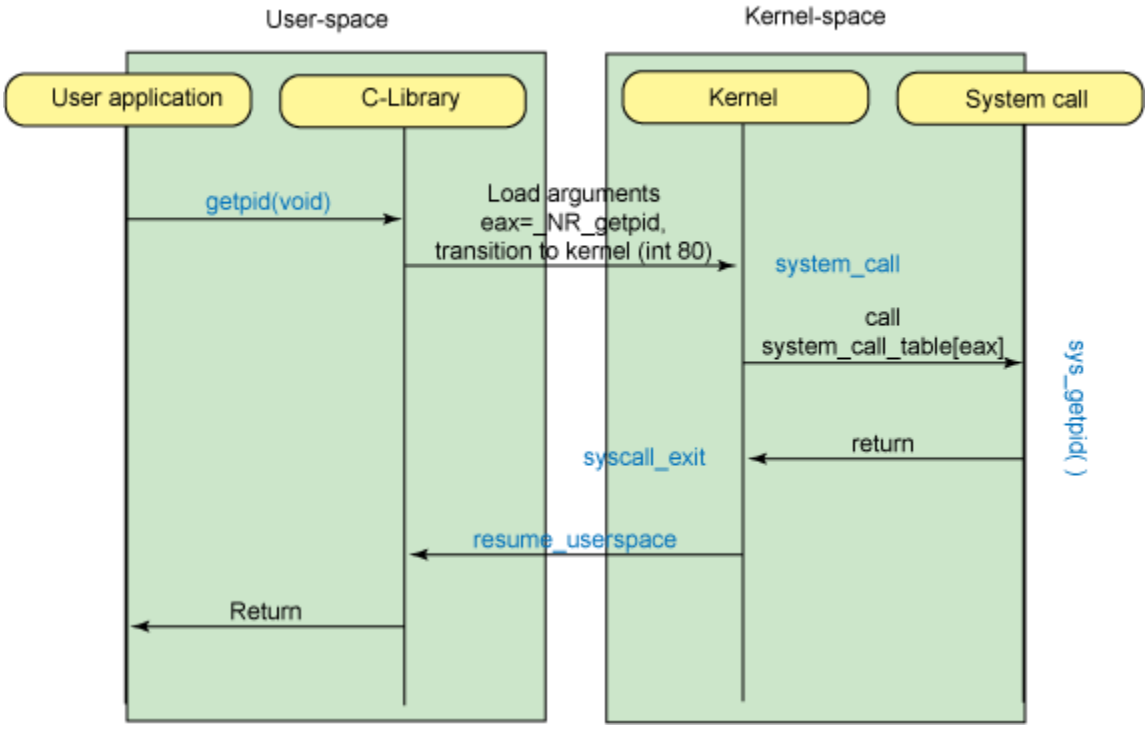
WRITING A SYSTEM CALL

Now, after reading through these system calls and some other we practiced them by implementing them we moved on to making our own system calls.

The steps to do so are:

1. You need to save the full kernel tree, it will not work on the existing kernel you are working on(the one you get by default when you install ubuntu for eg) as the new tree will only have the source code.
2. System calls often serves unique purpose.
3. They often interact with hardware.
4. Now coming to how to write it, we write a system call in c language.
5. After writing it we have to place it inside kernel with the name xyz.c
6. Now we have to manually write it's entry in each architecture ie arch/x1y1z1/include/asm/unistd.h (:x1y1z1 represents folder in arch)
7. We have to also define it in entry.S file

The location of file may vary with version of LINUX KERNEL you are working with.



DETAILS

A system call cannot be called directly from a user process. Instead, they are called indirectly via an *interrupt* and looked up in an interrupt table. Thus, to define a new system call we have to insert a new entry in this table. We do this by editing the file linux/arch/x86/kernel/entry64.S. Inside, we see lines like:

```
.data
ENTRY(sys_call_table)
  .long SYMBOL_NAME(sys_ni_call)          /* 0 */
  .long SYMBOL_NAME(sys_exit)
  .long SYMBOL_NAME(sys_fork)
  ...
  .long SYMBOL_NAME(sys_vfork)            /* 190 */
```

After the "sys_vfork" line, we have to add our entries for our new system calls, with the words "sys_" prepended. For example, our new system call "hello":

```
  .long SYMBOL_NAME(sys_hello)        /* 191 */
```

We also need to generate a system call so that an ordinary user program can invoke our system call. To do this we have to edit the file linux/include/asm/unistd.h where one will find lines like:

```
/*
  * This file contains the system call numbers.
```

```
 */

#define __NR_exit              1
#define __NR_fork              2
...
#define __NR_vfork           195
```

You should add #defines for your new system calls at the end, with the prefix "__NR_" in front of it. For example, you might add the line:

```
#define __NR_hello         196
```

NOTE: numbering may change.

After putting the entry the next task is to write the system call. But the question is where to write it. It can be written in any file that already contains the definition of some other system call already or you can create your own .c file an put it in linux/include/linux folder.

if you create your own .c file then you have to modify the Makefile in the directory you placed your .c file so that your code gets compiled and linked in properly. Modify the Makefile line to have a .o of your source code. For example, adding hello.o:

```
O_OBJS += ... hello.o
```

The rest of the Makefiles can remain untouched (Makefile changes will be similar if you choose to add your code elsewhere).

In order to be linked properly, your system calls need the word "asmlinkage" prepended to their function header and "sys_" prepended to the name. For example, we would have:

```
asmlinkage int sys_hello(ivoid) {
  /* implementation of hello */
}
```

Also, we will have to have #include <linux/linkage.h> so the compiler will recognize the word "asmlinkage".

There are some macros defined for this in <linux/unistd.h>. The format is "_syscallN(return type, function name, arg1 type, arg1 name ...)" where "N" is the number of parameters. For example, you might have the line:

```
_syscall0(int, hello);
```

Note, that this call should be put in the header file (hello.h). Also, we need to #include <linux/unistd.h> in hello.h to make this work. A user program could then just call hello() as they do other system calls.

Here is an example hello.h and hello.c, assuming hello.h is under linux/include/linux/

```
_____________________________________________________________________________
_
        //hello.h
#include <linux/unistd.h>
#include <linux/linkage.h>

_syscall2(int, hello);

_____________________________________________________________________________
_
#include <linux/myservice.h>                    //hello.c

asmlinkage int sys_hello(void) {
   printk("HELLO/n");
}

_____________________________________________________________________________
__
```

Basic Kernel Library Functions

Here is a few basic kernel library functions that might be useful. Refer to "man" pages for a detailed information. Note that you should use care to use cli() and sti(), since misuse of them can freeze your kernel with no error messages.

printk()	NAME printk - print messages to console log SYNOPSIS #include <linux/kernel.h> int printk(const char*fmt, ...)
kmalloc(), kfree()	NAME kmalloc, kfree - allocate and free pieces of memory SYNOPSIS #include <linux/malloc.h> void * kmalloc (size_t size, int priority); void kfree (void * __ptr);

cli(), sti()	NAME cli, sti - disable/enable interrupts SYNOPSIS #include <asm/system.h> extern void cli() extern void sti()

get_user(),
 	
put_user(),
 	
copy_from_user(),
 	
copy_to_user()
	NAME
	get_user, put_user, copy_from_user, copy_to_user - copy data between kernel space and user space
	
	SYNOPSIS
	#include <asm/uaccess.h>
	
	err = get_user (x, addr);
	err = put_user (x, addr);
	
	bytes_left = copy_from_user(void*to, const void *from, unsigned long n);
	bytes_left = copy_to_user(void*to, const void *from, unsigned long n);

DEVICE DRIVER

Each device is treated as a file in linux. But, simply to use read and write functions on them we have to do some operations on them.

Hardware(device) is treated as a special file called device file or device node. The system, therefore, needs a mechanism whereby it can distinguish between the various types of devices and behave accordingly.

To access a device accordingly, the operating system must be told what to do. Obviously, the manner in which the kernel accesses a hard disk will be different from the way it accesses a terminal. Both can be read from and written to, but that's about where the similarities end. To access each of these totally different kinds of devices, the kernel needs to know that they are, in fact, different.

Inside the kernel are functions for each of the devices the kernel is going to access. All the routines for a specific device are jointly referred to as the device driver. Each device on the

system has its own device driver. Within each device driver are the functions that are used to access the device. For devices such as a hard disk or terminal, the system needs to be able to (among other things) open the device, write to the device, read from the device, and close the device. Therefore, the respective drivers will contain the routines needed to open, write to, read from, and close (among other things) those devices.

-Types of device drivers

There are three types of device drivers :

1) Character Device Drivers: These devices work on a come and go basis meaning reading and writing on the device is done character by character like any normal file. Examples for such a device driver are terminal,printer,mouse.

2) Block Device Drivers: These act on devices that are capable of holding a filesystem themselves. In these devices, device manager can read or write data as a set (block) of defined size which is 512 bytes or any power of 2 inmost unix systems.In Linux they can also be read as char wise but the internal representation of block devices is entirely different from char devices.

3) Networking Devices Drivers: The network transactions are done through a device to communicate with other hosts. This device can be hardware as well as software. The network driver handles the packets. A network interface i.e. our device is incharge of sending and receiving data packets driven by the network subsystem of the kernel.

-writing modules

One of the good features of Linux is the ability to extend at runtime the set of features offered by the kernel. This means that you can add functionality to the kernel (and remove functionality as well) while the system is up and running.

Each piece of code that can be added to the kernel at runtime is called a module. The Linux kernel offers support for quite a few different types (or classes) of modules, including, but not limited to, device drivers. Each module is made up of object code (not linked into a complete executable) that can be dynamically linked to the running kernel by the insmod program and can be unlinked by the rmmod program.

To have a look at currently loaded modules on your system write the “lsmod” command in terminal. Now to unload any module type “rmmod modulename” and to load any module type “insmod modulename”. If loading of a module require some other non loaded module to be loaded then use “modprobe modulename”.

Every module has to include init.h and module.h file. Depending on your utility module contains many functions.

Hello_World Module

When loaded this module just print hello on the console and bye on exit.

```
#include<init.h>

#include<module.h>

#include <linux/module.h>

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init()

{

        printk(KERN_ALERT "hello\n");

        return 0;

}

static void hello_exit() printk(KERN_ALERT "bye\n");

module_init(hello_init);

module_exit(hello_exit);
```

Now, in this module helo_init() function is called when module is loaded and prints hello.hello_exit is being called when module is unloaded.

Macros module_init and module_exit are the macros that are being looked by kernel when the module is loaded and unloaded and these macros only execute the corresponding function.

-building modules

Once you have made a module now its time for building it so that it can be loaded by the kernel. Before building you need to create a makefile for your module. This is to be done because whenever we build kernel it looks for these makefiles to build these modules.After building a “modulename.ko” file is formed the “modulename.o” object file.

Structure of makefile

Makefile tells the kernel to create the .ko file out of .o . In fact, for the “hello” example shown earlier, a single line will suffice:

```
obj-m := hello.o
```

If, instead, you have a module called module.ko that is generated from two source files (called, say, file1.c and file2.c), the correct incantation would be:

```
obj-m := module.o
```

module-objs := file1.o file2.o

After the makefile it's time to build it. For this write the following command in the terminal:

$make -C ~/kernel-'uname -r' M=`pwd` modules

This command starts by changing its directory to the one provided with the -C option (that is, your kernel source directory). There it finds the kernel's top-level makefile. The M= option causes that makefile to move back into your module source directory before trying to build the modules target. This target, in turn, refers to the list of modules found in the obj-m variable, which we've set to module.o in our case.

-kernel symbol table

The "insmod" resolves undefined symbols against the table of public kernel symbols. The table contains the addresses of global kernel items—functions and variables—that are needed to implement modularized drivers. When a module is loaded, any symbol exported by the module becomes part of the kernel symbol table.In the usual case, a module implements its own functionality without the need to export any symbols at all. You need to export symbols, however, whenever other modules may benefit from using them.

-module parameters

If we need to define some values at runtime ie while calling insmod or modprobe we can do this.Kernel provides methods to do so. This is often used as certain specifications differ fron system to system.

However, before insmod can change module parameters, the module must make them available. Parameters are declared with the module_param macro, which is defined in moduleparam.h (include<linux/stat.h> module_param takes three parameters: the name of the variable, its type, and a permissions mask to be used for an accompanying sysfs entry. The macro should be placed outside of any function.

eg:

int c=1;
module_param(c);

Or if you want to take an array of values use the following function:

module_param_array(name,type,num,perm);

Where name is the name of your array (and of the parameter), type is the type of the array elements, num is an integer variable, and perm is the usual permissions value. If the array parameter is set at load time, num is set to the number of values supplied. The module loader refuses to accept more values than will fit in the array.

-Major and Minor Number

Now, finally after understanding about modules(writing,compiling and building) it's time to come to device drivers. Now, Each device driver is given two numbers called major and

minor number.Major number identifies the device driver for a similar group of devices and minor number specifies the device uniquely.

Now, we will discuss about how ro get major and minor number,hoe to alloate one for your device both dynamically and before hand.

dev_t type is used to store majar and minor number. To obtain a major or minor parts of a dev_t type use:

MAJOR(dev_t dev);

MINOR(dev_t dev);

If, instead, you have the major and minor numbers and need to turn them into a dev_t, use:

MKDEV(int major, int minor);

One of the first things your driver will need to do when setting up a char device is to obtain one or more device numbers to work with. The necessary function for this task is(definrd in <linux/fs.h>:

int register_chrdev_region(dev_t first, unsigned int count, char *name);

Here, first is the beginning device number of the range you would like to allocate. The minor number portion of first is often 0, but there is no requirement to that effect. count is the total number of contiguous device numbers you are requesting.

As with most kernel functions, the return value from register_chrdev_region will be 0 if the allocation was successfully performed. In case of error, a negative error code will be returned, and you will not have access to the requested region.

register_chrdev_region works well if you know ahead of time exactly which device numbers you want. Often, however, you will not know which major numbers your device will use.

Therefore kernel provides the following function to allocate device number dynamically:

int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);

With this function, dev is an output-only parameter that will, on successful completion, hold the first number in your allocated range. firstminor should be the requested first minor number to use; it is usually 0. The count and name parameters work like those given to request_chrdev_region.

Regardless of how you allocate your device numbers, you should free them when they are no longer in use. Device numbers are freed with:

void unregister_chrdev_region(dev_t first, unsigned int count);

The usual place to call unregister_chrdev_region would be in your module’s cleanup function.

-Different structures for char devices

As we have discussed there are three types of device driver so we now discuss about character devices. Kernel provides predefined structures to play with character devices. These structures are defined in <linux/fs.h>. Details of these structures are given below:

From a data-flow perspective, char drivers own the following key data structures:

1. *A per-device structure*: This is the information repository around which the driver revolves. This structure is the basic structure that represents the device. It has elements which represents information about devices. The following example is the structure that we wrote for our device.

```
struct mydevice{
        struct dataset set;             // To store data
        unsigned long size;             // To store the memory occupied
        struct cdev cdev;               //structure which represent file in kernel internel
        int quanta;                     // We have distributed memory into samll chunks using
                                         quanta to make our data handling simple.
        int qset;                       // current array size. Stores number of current quanta.
        struct semaphore sem;           // barrier lock to check compilor optimization.
  };
```

---

2. Inode structure : The inode structure contains a great deal of information about the file. As a general rule, only two fields of this structure are of interest for writing driver code:

dev_t i_rdev : For inodes that represent device files, this field contains the actual device number.

struct cdev *i_cdev : struct cdev is the kernel's internal structure that represents char devices; this field contains a pointer to that structure when the inode refers to a char device file. A kernel abstraction for character drivers. This structure is usually embedded inside the perdevice structure referred previously.

3. struct file_operations: which contains the addresses of all driver entry points. To use this structure you should include linux/fs.h . Each of the file is operated through struct file_operations which contains function pointer of different functions like open, read, lseek, write and many more.

(If you have heard first time about function pointers than you can just consider a file like object and the function pointer as similar to methods in object-oriented programming languages. A function pointer is similar to the data pointers but instead here we also pass function as an argument. Here is an examle :)

The list of the function pointers that are present in structure are listed in file linux/fs.h you can have a look at it or the list of some of them is as follows :

struct module *owner The first file_operations field is not an operation at all; it is a pointer to the module that “owns” the structure. This field is used to prevent the module from being unloaded while its operations are in use. Almost all the time, it is simply initialized to THIS_MODULE, a macro defined in <linux/module.h>.

loff_t (*llseek) (struct file *, loff_t, int); The llseek method is used to change the current read/write position in a file, and the new position is returned as a (positive) return value. The loff_t parameter is a “long offset” and is at least 64 bits wide even on 32-bit platforms. Errors are signaled by a negative return value. If this function pointer is NULL, seek calls will modify the position counter in the file structure in potentially unpredictable ways.

ssize_t (*read) (struct file *, char __user *, size_t, loff_t *); Used to retrieve data from the device. A null pointer in this position causes the read system call to fail with -EINVAL (“Invalid argument”). A nonnegative return value represents the number of bytes successfully read (the return value is a “signed size” type, usually the native integer type for the target platform).

ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *); Sends data to the device. If NULL, -EINVAL is returned to the program calling the write system call. The return value, if nonnegative, represents the number of bytes successfully written.

int (*readdir) (struct file *, void *, filldir_t); This field should be NULL for device files; it is used for reading directories and is useful only for filesystems.
Here is the code for our file operations: ----------

// This structure is predefined in fs.h which we can use to perform operations on our memory.

```
struct file_operation mydiv_fops={
        .open=myopen,
        .owner=THIS_MODULE,
        .read=myread,
        .write=mywrite,
        .release=myrelease,
        .llseek=myposptr,
  };
```

---

4. struct file: which contains information about the associated /dev node. It is also present in linux/fs.h . This is not the FILE that we normally use in a c program. The file structure represents an open file. (It is not specific to device drivers; every open file in the system has

an associated struct file in kernel space.) It is created by the kernel on open and is passed to any function that operates on the file, until the last close. After all instances of the file are closed, the kernel releases the data structure. The pointer to this structure in kernel is called filp.

CODE OF OUR DEVICE DRIVER :

```
/* This device driver make use of memeory as a device and performs read and write
functions on it. This is a char driver that acts on a memory area as though it were a device.
This device which we are using is hardware independent it just acts on the piece of memory
allocated by the kernel. In this device driver we have used functions "myread" and "mywrite".
In this device driver we are using memory in the form of quanta each quanta can hold 4000
bytes of the memory and we are using qset to represent the    number of quantums.

*/

#include<linux/kdev.h>           //including MKDEV macro;

#include<linux/fs.h>             //including file and other structures

#include<linux/module.h>

#include<linux/init.h>

#include<asm/uaccess.h>       //function to move data

#include<linux/kernel.h>        //container_of() fnctn

#include<linux/cdev.h>           //registry fnctn

#include<linux/slab.h>           //to use the functions kmalloc and kfree


#define major 900

#define minor 0

// This structure is predefined in fs.h which we can use to perform operations on our
memory.

struct file_operation mydiv_fops={

            .open=myopen,

            .owner=THIS_MODULE,

            .read=myread,

           .write=mywrite,

            .release=myrelease,

            .llseek=myposptr,
```

```
        };
// This structure is used to point at the data.
struct dataset{
        void** data;
        struct dataset* next;
       };
// This is our main device structure.
struct mydevice{
        struct dataset set;      // To store data
        unsigned long size;    // To store the memory occupied
        struct cdev cdev;       //structure which represent file in kernel internel
        int quanta;              // We have distributed memory into samll chunks using quanta
                                  to make our data handling simple.
        int qset;                // current array size. Stores number of current quanta.
        struct semaphore sem;  // barrier lock to check compilor optimization.
        };
//Prototypes of the functions
static void my_device_setup_cdev(struct mydevice* , int );     //This function intializes and
                                                                adds the device
to the                                                          system for usage.
int myopen(struct inode * , struct file * );                    // To open the device.
int myrelease(struct inode * , struct file * );                 // To release the device.
int mytrim(struct mydevice *);                                   // This function trims the memory
                                                                 when required. For example
                                                                 when we over-write with a  short
                                                                 file then it trims rest of the
                                                                 memory.
int myread(struct file *, char __user *, size_t , loff_t * );
int mywrite(struct file *, char __user *, size_t , loff_t * );
//SEPERATE FUNCTION DEFINATIONS
int mytrim(struct mydevice *dev)
```

```
{
    struct dataset *next, *dptr;
    int qset = dev->qset;           /* "dev" is not-null */
    int i;
    for (dptr = dev->data; dptr; dptr = next)
    { /* all the list items */
            if (dptr->data)
            {
                    for (i = 0; i < qset; i++) kfree(dptr->data[i]);
                    kfree(dptr->data);
                    dptr->data = NULL;
            }
            next = dptr->next;
            kfree(dptr);
    }
    dev->size = 0;
    dev->quantum = 4000;
    dev->qset =1000;
    dev->data = NULL;
    return 0;
}
int myopen(struct inode *inode,struct file * filp)
{
    struct mydev *dev;              // device information
    dev = container_of(inode->i_cdev, struct scull_dev, cdev);
    filp->private_data = dev;     // for other methods
    // now trim to 0 the length of the device if open was write-only
```

```
    if ( (filp->f_flags & O_ACCMODE) = = O_WRONLY) mytrim(dev);     //ignore errors


    /* O_ACCMODE and O_WRONGLY are macros defined in fcntl.h O_WRONLY stands for
write only access. O_ACCMODE when bitwise ANDed with file status flag value it produces
a value representing the file access mode.*/

 return 0;                    // success

}

static void my_device_setup_cdev(struct mydevice *dev , int index)

{

    int err, devno = MKDEV(major, minor + index);

    cdev_init(&dev->cdev, &scull_fops);          //initializing device

    dev->cdev.owner = THIS_MODULE;

    dev->cdev.ops = &scull_fops;

    err = cdev_add (&dev->cdev, devno, 1);    //adding device so that it's entry get defined in
                                                proc/devices

    //if fail

    if (err) printk(KERN_NOTICE "Error %d adding scull%d", err, index);

}

int myread(struct file *filp , char __user *buf , size_t size, loff_t *f_pos)

{

    mydevice *dev;

    dev=filp->private_data;

    int quant=dev->quanta,qset=dev->qset;

    long int setsize=quant*qset,dsize=dev->size;

    loff_t pos=*f_pos;

    if(pos>=dsize) retorn 0;

    if(pos+size>dsize) size=dsize-pos;

    /* Now we have to reachto the pos to read data from there. As data is being written in the
form of qset and quantum we have to reachto the proper quantum.    */

    int setpos=pos/setsize;
```

```
    int rest=pos%setsize;
    int qpos=rest/quant;
    int qbit=rest%quant;
    int temp=0;
    struct dataset dset=dev->set;
    //reach to setpos
    while(setpos--) dset=dset.next;
    //now we have the dataset that contains our pos, we just now want to reach exactly to that position.
    char *data=dset[qpos]+qbit;
    if(size > quant-qbit) size=quant-qbit;
    if(copy_to_user(buf,data,size))
    {
            return 0;
    }
    *f_pos+=size;
    return size;
}
int mywrite(struct file *filp , char __user *buf , size_t size, loff_t *f_pos)
{
    mydevice *dev;
    dev=filp->private_data;
    int quant=dev->quanta,qset=dev->qset;
    long int setsize=quant*qset,dsize=dev->size;
    loff_t pos=*f_pos;
    if(pos>=dsize) retorn 0;
    if(pos+size>dsize) size=dsize-pos;
    /* Now we have to reach to the pos to write data from there.    */
    int setpos=pos/setsize;
```

```
    int rest=pos%setsize;
    int qpos=rest/quant;
    int qbit=rest%quant;
    int temp=0;
    struct dataset dset=dev->set;
    //reach to setpos
    while(setpos--) dset=dset.next;
   //now we have the dataset that contains our pos, we just now want to reach exactly to that position and also we have to assign memory.
    if(!dset->data)
    {
        dset->data = kmalloc(qset*sizeof(char*),GFP_KERNEL);
        memset(dset->data, 0, qset * sizeof(char *));          //Initialises to NULL
    }
    if(!dset->data[qpos]) dset->data[qpos]=kmalloc(quant,GFP_KERNEL);
    if(size > quant-qbit) size=quant-qbit;
    if(copy_from_user(dset->data[qpos]+qbit,buf,size))
    {
        return 0;
    }
    *f_pos+=size;
    if(*f_pos > dev->size) dev->size+=(long) *f_pos;
    return size;
}
```

==========

Debugging by the printk :

This is the most simplest way of debugging. Printk lets you classify messages according to their severity by associating different loglevels, or priorities, with the messages. You usually indicate the loglevel with a macro. For example, KERN_INFO, which we saw prepended to some of the earlier print statements, is one of the possible loglevels of the message. The

loglevel macro expands to a string, which is concatenated to the message text at compile time.

There are eight possible loglevel strings, defined in the header <linux/kernel.h>; we list them in order of decreasing severity:

KERN_EMERG Used for emergency messages, usually those that precede a crash.

KERN_ALERT A situation requiring immediate action.

KERN_CRIT Critical conditions, often related to serious hardware or software failures.

KERN_ERR Used to report error conditions; device drivers often use KERN_ERR to report hardware difficulties.

KERN_WARNING Warnings about problematic situations that do not, in themselves, create serious problems with the system.

KERN_NOTICE Situations that are normal, but still worthy of note. A number of security-related conditions are reported at this level.

KERN_INFO Informational messages. Many drivers print information about the hardware they find at startup time at this level.

KERN_DEBUG Used for debugging messages.

Recovering system hangs:

If the code enters an endless loop, the kernel stops scheduling,* and the system doesn't respond to any action, including the magic Ctrl-Alt-Del combination. You have two choices for dealing with system hangs—either prevent them beforehand or be able to debug them after the fact. You can prevent an endless loop by inserting schedule invocations at strategic points. The schedule call (as you might guess) invokes the scheduler and, therefore, allows other processes to steal CPU time from the current process. If a process is looping in kernel space due to a bug in your driver, the schedule calls enable you to kill the process after tracing what is happening.

Concurrency and its solution : Suppose for a moment that two processes (we'll call them "A" and "B") are independently attempting to write to the same offset within the same scull device. Each process reaches the if test in the first line of the fragment above at the same time. If the pointer in question is NULL, each process will decide to allocate memory, and each will assign the resulting pointer to dptr->data[s_pos]. Since both processes are assigning to the same location, clearly only one of the assignments will prevail***.***

The usual technique for the access management is called locking or the mutual exclusion - making sure that only one thread of execution can manipulate a shared resource any time.

SEMAPHORE :

Our goal is to make our operations on the mydevice data structure atomic, meaning that the entire operation happens at once as far as other threads of execution are concerned.

For our present needs, however, the mechanism that fits best is a semaphore. At its core, a semaphore is a single integer value combined with a pair of functions that are typically called P and V. A process wishing to enter a critical section will call P on the relevant semaphore; if the semaphore's value is greater than zero, that value is decremented by one and the process continues. If, instead, the semaphore's value is 0 (or less), the process must wait until somebody else releases the semaphore. Unlocking a semaphore is accomplished by calling V; this function increments the value of the semaphore and, if necessary, wakes up processes that are waiting.

The P function is called down—or some variation of that name. Here, "down" refers to the fact that the function decrements the value of the semaphore and, perhaps after putting the caller to sleep for a while to wait for the semaphore to become available, grants access to the protected resources. There are three versions of down:

void down(struct semaphore *sem);

int down_interruptible(struct semaphore *sem);

int down_trylock(struct semaphore *sem);

down decrements the value of the semaphore and waits as long as need be. down_interruptible does the same, but the operation is interruptible. The interruptible version is almost always the one you will want; it allows a user-space process that is waiting on a semaphore to be interrupted by the user.

The Linux equivalent to V is up:

void up(struct semaphore *sem);

Once up has been called, the caller no longer holds the semaphore.

REFERENCES

-Modern Operating Systems 2nd Ed By Tanenbaum (With Pdf Index)

-Essential Linux Device Drivers[2008,851p]

-intro-linux.pdf

-http://www.linuxmanpages.com/

-google search engine