Lab 1: Basic Linux Familiarity

Objectives:

- To get familiar with Linux and its command systems

Submission:

- A report with the answers of all questions which will be supplied separately.

Introduction:

This motivation of this lab is to prepare for you for the subsequent labs in which we will be using the Ubuntu Linux environment. There will be 25 questions in this lab which will be supplied separately during the lab. You will also need to submit a report at the end of the lab with the answers of the questions. This lab document starts with a little bit of history of Unix and Linux. Then, it introduces different other aspects of the Linux operating system.

UNIX:

UNIX is a powerful operating system designed to be a multiuser and multitasking system. The original UNIX was created by Ken Thompson in 1969 at Bell Labs. Today, the term "UNIX" does not refer to a single operating system sold by a single company. Instead, it refers to any operating system that meets certain standards.

Most large-scale computers and some desktop personal computers use an UNIX operating system, which could be a generic system or one written by the computer manufacturer. Some of the more popular UNIX operating systems include Linux, Ultrix (DEC), Irix (Silicon Graphics), and Solaris (Sun Microsystems). Mac OSX is built on BSD Unix.

Since each UNIX operating system must meet the same standards, they function similarly. Thus, after you are familiar with the use of one UNIX operating system, you will easily adapt to a different one. The major differences are usually in the administration of the system—meaning, unless you are the administrator of the system, you never have to worry about that aspect.

Linux:

The Linux (Lynn-ucks) operating system was created by Linus Torvalds in 1991 while he was a graduate student at the University of Helsinki (Finland). Torvalds created Linux as an alternative to Microsoft Windows and to provide a UNIX operating system for use on the PC. Linux is and has always been an open- source project that allows other programmers to view and modify the source code. Today, hundreds of programmers work on Linux—mostly in their spare time—under the direction of Torvalds.

The Linux operating system is very popular today due in part to its availability and open source status. There are a number of Linux distributions available from different companies and groups such as Red Hat, Fedora, Ubuntu, Slackware, SuSe, and Corel. All of these use the same Linux operating system.

The major differences between the distributions are the services provided and the various applications included with the Linux distribution. Linux is very powerful and is easy to learn

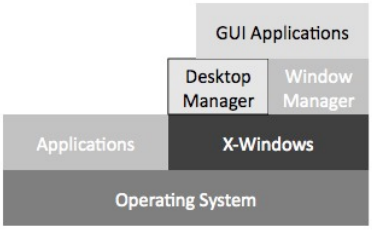
and use. All major distributions provide a graphical user interface frontend that will be familiar to Microsoft and Macintosh users.

In our lab we will be using Ubuntu Linux Distribution.

User Interface:

Users commonly interact with a UNIX system via a text-based command-line interface. In a terminal, commands are entered at a prompt and results are displayed. Numerous commands are provided for file and directory manipulation, program execution, and file processing. With a text-based interface, users can work with the system directly on the physical machine or connect to a remote system via the Internet. One of the biggest wins with the text-based interface is how easy it is to automate complicated processes.

In addition to the text-based interface, most UNIX systems also provide a graphical interface similar to those of the Macintosh and Windows platforms. Unlike those systems, however, the graphical interface under UNIX consists of several layers, as illustrated below.



The actual graphical environment is provided by a standalone program called X-Windows (tm). This program is executed on top of the text-based interface. X-Windows is responsible for managing the monitor, keyboard, and mouse in addition to providing the “windowing capability”.

The look and feel of the graphical environment is provided by a window manager. The window manager is a standalone program that works in conjunction with X-Windows to provide decorations for windows, buttons, and other components in addition to controlling the behaviour and action of the various components. There are a number of window managers from which to choose.

Linux systems also provide a desktop manager. The desktop manager is responsible for the icons on the desktop, the panels that provide menus and application launchers, and virtual workspaces. The two most popular desktop managers are GNOME (default) and KDE, however, Unity is the default desktop manager of recent Ubuntu releases.

The Terminal:

Before you can begin working with UNIX commands, you need a terminal window, which runs a program called a shell. The shell provides an interface between you and the operating system. The terminal can be launched using the Menu, or typing Terminal into the application area or using the Ctrl + Alt + T shortcut.

The terminal window on your desktop should contain some characters that look something like:

```
[username@lisp ~]$
```

This is called the prompt. The prompt gives you information about the account and machine being used and the current directory you're in. For example, the prompt above is for someone whose username is username and who is using the computer named lisp and that the user is in his home directory, as indicated by the ~, the shortcut for the home directory.

The prompt indicates that UNIX is waiting for (or "prompting") you to type something. Whenever you type something after a UNIX prompt, UNIX tries to understand it as a command. If you type a command that UNIX understands, UNIX carries out the command. Otherwise UNIX displays a message indicating that the command was not recognizable.

Note: Commands and filenames in UNIX are case sensitive.
Spaces are required between commands, arguments, and options.
The basic template for a UNIX command is

```
command-name [options] [argument1] [argument2] ...
```

To enter a UNIX command, the terminal window must be the active window. To make a window active, simply move the mouse so that the mouse pointer is located within the limits of the window and click on the window. When a window is active, its border changes colour.

- Make the terminal window the active window.
- At the prompt, type in your last name and press Enter.

You should see the error message that UNIX displays when it cannot interpret what you type as one of its commands (unless your parents happened to name you "ls" or "cp" or some such thing).

Linux Manual

Unix documentation is traditionally in the form of a *Unix Manual*, which is comprised of a set of *manual pages*, or simply *man pages*, organized into nine sections. Section one of the manual is for user commands, section two for system calls, section three for higher-level API calls and so on. Sometimes you will see commands (and API functions) written as *name*(*n*). This notation specifies a name and a manual section. For example, tty(1) refers to the user command t t y, whereas tty(4) refers to the device driver named *tty*.

You read man pages using the man command. The man command itself has a man page, which you read by issuing the command man man.

- Type man man in your terminal and try to understand what is written

Before moving on to more advanced tasks, you have to become comfortable reading man pages, and referring to the man pages must become second nature. Any time you wonder how a command works, read the man pages. If you need to know what format a file has, read the man pages. If you don't have anything else to do, read a man page; you might just learn something.

Man pages are divided into named sections such as “SYNOPSIS”, “DESCRIPTION”, “EXAMPLES” and “FILES”. If you are familiar with the more common sections of man pages you can find information a lot faster than by trying to read the whole thing from beginning to end. The man page for man itself lists some of the common sections and conventions.

Linux File Systems

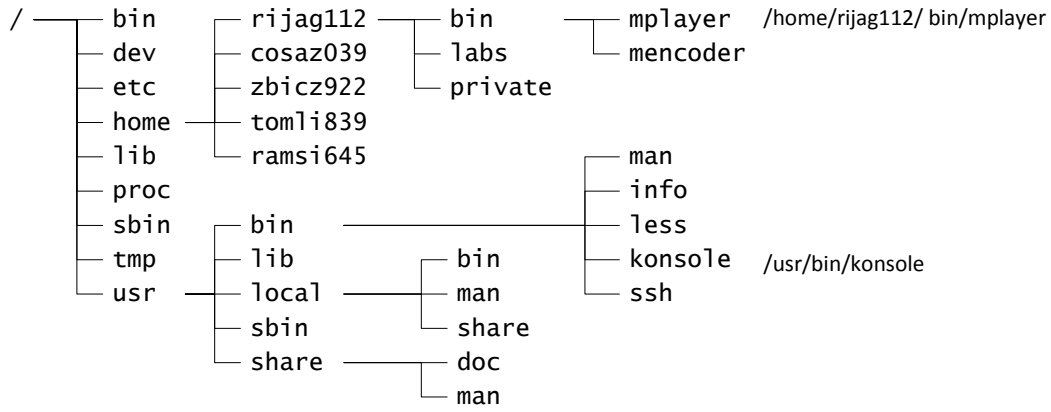
Understanding how files and directories are organized and can be manipulated is vital when using or managing a Linux system. All files and directories in Linux are organized in a single tree, regardless of what physical devices are involved (unlike Microsoft Windows, where individual devices typically form separate trees).

The basic unit of storage in UNIX is a file. A file may contain many kinds of information, including a Python script, an HTML document, a research paper, an image, or an executable program. Files are organized in a hierarchical system of directories. A directory may contain files and other directories. The directory at the “top” of the file system, in which all other directories are located, is called the root directory.

The root of the tree is called /, and is known as the root directory or simply the root. The root contains a number of directories, most of which are standard on Linux systems. The following top-level directories are particularly important:

Directory	Purpose
bin	Commands (binaries) needed at startup. Every Unix command is a separate executable binary file. Commands that are fundamental to operation, and may be needed while the system is starting, are stored in this directory. Other commands go in the /usr directory.
dev	Interfaces to hardware and logical devices. Hardware and logical devices are represented by device nodes: special files that are stored in this directory.
etc	Configuration files. The /etc directory holds most of the configuration of a system. In many Linux systems, /etc has a subdirectory for each installed software package.
home	Home directories. User’s home directories are subdirectories of /home.
sbin	Administrative commands. The commands in /sbin typically require administrative privileges or are of interest only to system administrators. Commands that are needed when the system is starting go in /sbin. Others go in /usr/sbin.
tmp	Temporary (non-persistent) files. The /tmp directory is typically implemented in main memory. Data stored here is lost when the system reboots. Many applications use /tmp for storing temporary files (others use /var).
usr	The bulk of the system, including commands and data not needed at startup. The usr subdirectory should only contain files that can be shared between a number of different computers, so it should contain no configuration data that is unique to a particular system.

The figure below shows part of a Unix system.



File and path names

There are two ways to reference a file in Unix: using a relative path name or using an absolute path name. An absolute path name always begins with a / and names every directory on the path from the root to the file in question. For example, in the figure above, the konsole file has the absolute path name /usr/bin/konsole. A relative path names a path relative the *current working directory*. The current working directory is set using the **cd** command. For example, if the current working directory is /usr, then the konsole file could be referenced with the name bin/konsole. Note that there is no leading /. If the current working directory were /usr/share, then konsole could be referenced with ../bin/konsole. The special name ".." is used to reference the directory above the current working directory.

File system permissions

Like most operating systems Linux has permissions on files and directories that grant individual users or groups of users rights on the files and folders.

In Linux, permissions are divided into three groups: "user", "group" and "other". User permissions apply to the owner of a file or directory; group permissions to the members of the file's (or directory's) group; other permissions apply to everyone else.

Every group contains three main permissions: read, write and execute, and each is represented as one bit in an integer. The read (r) bit grants permission to read the contents of a file or directory; the write (w) bit grants permission to write to the file or create files in a directory, and the execute (x) bit grants permission to execute a file as a program. On directories the execute bit grants permission to traverse the directory (i.e. set it as the working directory).

There are other permission bits as well. The most important of these are the setuid and setgid bits (in the user and group permission groups, respectively). When a program with the setuid bit set is run, it is run as the owner of the file, not the user who started the program. The setgid bit works the same, but for groups.

To list the permissions of a file or directory, use the ls command with the -l option (to enable long file listing; see the man page for ls). You can see something like this:

```
%ls -l foobar
-rwxr-xr-- 1 john users 64 May 26 09:55 foobar
```

Each group of permissions is represented by three characters in the leftmost column of the listing. The very first character indicates the type of the file, and is not related to permissions. The next three characters (in this case rwx) represent user permissions. The following three (in this case r-x) represent group permissions and the final three represent permissions for others (in this case r--).

The owner and group of the file are given by the third and fourth column, respectively (user john and group users in this example).

In this example the owner, "john", is allowed to read, write and execute the file (rwx). Users belonging to the group "users" are allowed to read and execute the file (r-x), but cannot write to it. All other users are allowed to read foobar (r--), but not write or execute it.

File types

The first character, the type field, indicates the file type. In the example above the file type is "-", which indicates a regular file. Other file types include: d for directory, l (lower case ell) for symbolic link, s for Unix domain socket, p for named pipe, c for character device file and b for block device file.

Manipulating access rights

The chmod and chown commands are used to manipulating permissions.

chmod is used to manipulate permissions. Permissions can be specified using either "long" format or a numeric mode (all permission bits together are called the file's mode). The long format takes a string of permission values (r, w or x) together with a plus or minus sign. For example, to prevent any user from changing the file foobar we would do as follows to disable write permission, then verify that the change has taken place:

```
% chmod -w foobar
% ls –l foobar
-r-xr-xr-x 1 john users 81 May 26 10:43 foobar
```

In numeric mode, each permission is treated as a single bit value. The read permission has value 4, write value 2 and execute value 1. The mode is a three character octal string where the first digit contains the sum of the user permissions, the second the sum of the group permissions and the third the sum of the others permissions. For example, to set the permission string "-rwxrw-r--" (user may do anything, group may read or write, but not execute and all others may read) for a file, you would calculate the mode as follows:

- User: 4 + 2 + 1 = 7 (rwx)
- Group: 4 + 2 = 6 (rw-)
- Others: 4 = 4 (r--)

Together with chmod the string "764" can then be used to set the file permissions:

```
% chmod 764 foobar
% ls -l foobar
-rwxrw-r-- 1 john users 81 May 26 10:43 foobar
```

Numeric combinations are generally quicker to work with once you learn them, especially when making more complicated changes to files and directories. Therefore, you are encouraged to use them. It is useful to learn a few common modes by heart:

755	Full rights to user, execute and read rights to others. Typically used for executables.
644	Read and write rights to user, read to others. Typically used for regular files.
777	Read, write and execute rights to *everybody*. Rarely used.

The chown is used to change the owner and group for a file. To change the user from “john” to “mike” and the group from “users” to “wheel” issue:

```
% chown mike:wheel foobar
```

Note that some Unix systems do not support changing the group with chown. On these systems, use chgrp to change tile file’s group. Changing owner of a file can only be done by privileged users such as root. Unprivileged users can change the group of a file to any group they are a member of. Privileged users can alter the group arbitrarily.

Symbolic links

In Unix, it is possible to create a special file called a symbolic link that points to another file, the target file, allowing the target file to be accessed through the name of the special file. Similar functions exist in other operating systems, under different names (e.g. “shortcut” or “alias”).

For example, to make it possible to access /etc/init.d/myservice as /etc/rc2.d/S98myservice, you would issue the following command:

```
% ln –s /etc/init.d/myservice /etc/rc2.d/S98myservice
```

Symbolic links can point to any type of file or directory, and are mostly transparent to applications.

File and Directory Manipulation Commands

Many Unix commands are concerned with manipulating files and directories. The following lists some of the most common commands in their most common forms. The man page for each command contains full details, and reading the man pages will be necessary to complete following the exercises.

Command	Purpose
touch *filename*	Change the creation date of *filename* (creating it if necessary).
pwd	Displays the current working directory.
cd *directory*	Changes the current working directory to *directory*.
ls	Lists the contents of directory. If directory is omitted, lists the contents of the current working directory. With arguments, can display information about each file (see the manual page).
cat *filename*	Display the contents of *filename*
less *filename*	Displays the contents of *filename* page-by-page (less is a so-called pager). Press the space bar to advance one page; b to go back one page; q to quit; and h for help on all commands in less.
rm *filename*	Removes the file *filename* from the file system.
mv *oldname newname*	Renames (moves) the file *oldname* to *newname*. If *newname* is an existing directory, moves *oldname* into the directory *newname*.
mkdir *dirname*	Creates a new directory named *dirname*.
rmdir *dirname*	Removes the directory *dirname*. The directory must be empty for rmdir to work.
cp *filename newname*	Creates a copy of *filename* named *newname*. If *newname* is a directory, creates a copy named filename in the directory *newname*.
chmod *modes filename*	Change permissions on *filename* according to *modes*.
chgrp *group filename*	Change the group of *filename* to *group*.
chown *user filename*	Change the owner of *filename* to *user*.
ln –s *oldname newname*	Creates a symbolic link, so that *oldname* can also be accessed as *newname*.

The Command shell

In Unix, the shell is the program that is responsible for interpreting commands from the user. The canonical shell is the *bourne shell*, sh, which has evolved into the POSIX shell. This shell has limited functionality, but is often used for shell scripts (programs written in the shell command language). On Linux, the most common shell is bash (*bourne again shell*). Bash is a POSIX-compatible shell that adds a number of useful functions. For interactive use, its line editing and command history are particularly important. There are a number of other shells available. The *Korn shell* (ksh), is standard on many systems, as is the *C shell* (csh) and the *TC shell* (t csh).

Each shell uses its own syntax for internal functions (such as setting variables, redirecting I/O and so forth), but there are two main variants in widespread use. Shells that trace their roots to the bourne shell use one syntax (which is POSIX-compatible), and shells that are based on the C shell use another. In addition, there are a number of shells which owe little to either of these traditions, and they may use a completely different (and occasionally quite bizarre) syntax.

When the shell starts, it reads one or more files, depending on how it is started. These are called rc or init files. For example, the bourne shell reads the file .profile in your home directory, while t csh reads .login and .tcshrc (if started as a login shell). These files may contain sequences of shell commands to run automatically. Typically, they are used to set up the shell and environment to suit the user's preferences.

Using the shell efficiently

Learning to use the shell efficiently is a very worthwhile investment. New users should at the very least learn how to use the command history (repeating previous commands), command line editing (editing the current or previous commands) and tab completion (saving time by letting the computer figure out what you mean).

The following text assumes that you are using bash or zsh with bash-like key bindings. Other shells will behave differently; the manual for the shell will explain how.

Command history

All (at least many) of the commands you type are kept in the command history. You can browse the history by using the up and down arrows (or crtl+P and crtl+N). When you find a command you want to use, you can edit it just as if you had typed it on the command line. You should also be aware of esc+< and esc+>, which move to the beginning and the end of the command history, respectively. You can also search the command history by typing crtl+R and then the word you want to search for.

Tab completion

Completion is one of the most useful features of a good shell. The idea behind completion is that often the prefix of something (a command, file name or even command-line option) uniquely identifies it, or at least uniquely identifies *part* of it. For example, if there are two files in a directory, READFIRST and READSECOND, when a user types R where the shell expects a file name, the shell can deduce that the next three characters will be EAD, and when the user has typed READS, the shell can deduce that the user means READSECOND.

Rather than type out annoyingly long file names, learn to use tab completion.

Environment and shell variables

Unix, and many other operating systems, including Windows NT/2000/XP/2003/Vista have the concept of *environment variables*. These are name-to-value mappings that are available to each running program, and constitute the program's environment. In Unix, environment variables are widely used for simple configuration of programs. Unix shells typically support *shell variables* in addition to environment variables. These are variables that are available to the shell, but are not exported to other processes.

Environment and shell variables are altered using shell syntax:

NAME=*VALUE*

POSIX (and `bash`) syntax. Sets the variable *NAME* to *VALUE*. Does not necessarily set the environment variable (shell dependent).

`export` *NAME*

POSIX (and `bash`) syntax. Makes *NAME* and its value part of the environment, so its value is available to any program that is started from the shell after the export command was given (programs started from other shells are not affected).

`set env` *NAME VALUE*

C shell syntax. Sets the environment variable *NAME* to *VALUE*. Use `set` instead of `set env` to set a shell variable.

All (useful) Unix shells support *parameter expansion*. This process replaces part of a command line with the contents of an environment or shell variable. In most shells, the syntax is "${ NAME} " to expand the environment variable NAME. The echo command can be combined with variable expansion to output the value of a particular variable. For example, "echo ${ HOME} ", when HOME is set to "/home/user ", will output "/home/user ". Note that the shell is responsible for expanding the variable; the echo command will receive the contents of the variable as its sole argument. The man page for your shell will list various ways of performing parameter expansion.

Redirecting I/O & Pipeline

Unix provides several ways of redirecting the output of commands to files or other commands and several ways of directing data to the input of commands. The basic mechanisms are redirections and pipes. The precise mechanisms depend on the shell you are using; these instructions assume the bash shell (see "The Command shell" above for more information about shells).

In Unix, I/O is performed from *file descriptors*. These are simply numbered input or output streams that point to sources or destinations of data (e.g. files, terminals, network connections). By convention, file descriptor 0 is called *standard input* or *stdin*, and is the default source for input; file descriptor 1 is called *standard out* or *stdout*, and is where output is sent by default; file descriptor 2 is called *standard error* or *stderr*, and is usually used for printing error messages.

I/O redirection simply is a matter of changing what the file descriptors point to. You can redirect output from a command to a file using the > or >> operators.

command > filename

The output of *command* is written to *filename*. The file will be created if it doesn't exist, and any previous contents will be erased. In some shells there is a *noclobber* option. If this is set, you may have to use the >! operator to overwrite an existing file.

In technical terms, this opens *filename* for writing, then changes file descriptor 1 (stdout) to point to the open file.

command >> filename

The output of *command* is appended to *filename*. The file will be created if it doesn't already exist.

In technical terms, this opens *filename* for writing, seeks to the end of the file, then changes file descriptor 1 (stdout) to point to the open file.

These basic redirection commands only redirect standard output; they do not redirect standard error. If you want to redirect all output, you have to redirect file descriptor two as well. The exact syntax for redirecting errors (and other file descriptors) is very shell-dependent.

command 2> *filename*

The output of *command* to standard error (usually error messages) written to *filename*. The file will be created if it does not already exist, and any previous contents will be overwritten.

In technical terms, this is the same as >, but it changes file descriptor 2 (stderr) instead of file descriptor one.

command 2>> *filename*

The output of *command* to standard error (usually error messages) is appended to *filename*. The file will be created if it does not already exist.

In technical terms, this is the same as >>, but it changes file descriptor 2 (stderr) instead of file descriptor one.

command 2>&1

Output from *command* to standard error is sent to whatever standard out points to at the moment (it does *not* link standard error and standard out, so if standard out is redirected later, that redirection will *not* affect standard error). The most common use of this is to redirect standard out and standard error to the same file.

Technically, file descriptor 2 becomes a copy of file descriptor 1 so that they point to the same thing. The two file descriptors remain independent of each other. This means that the order in which you perform redirections matters when using 2>&1.

In addition to redirecting output to files, it is possible to redirect output to other commands. The mechanism that makes this possible is called *pipe*. The Unix philosophy of command design is that each command should perform one small function well, and that complex functions are performed by combining simple commands with pipes and redirection. It actually works quite well.

command1 | *command2*

The output (standard out) from *command1* is used as the input (standard in) to *command2*. Note that this connection is made *before* any redirection takes place.

From a technical point of view, file descriptor 1 (stdout) of *command1* becomes linked to file descriptor 0 (stdin) of *command2*.

command1 2>&1 | *command2*

Both standard out and standard error from *command1* will be used as input (standard in) to *command2*.

From a technical point of view, both file descriptor 2 (stderr) and file descriptor 1 (stdin) will be linked to file descriptor 0 (stdin) of *command2*.This works because pipes are always connected before redirection.

Processes and jobs

Linux is a multi-tasking, multi-user operating system. Several users can use the computer at once, and each user can run several programs at the same time. Every program that is executed results in at least one *process*. Each process has a process identifier and has its own memory area not shared with other processes. A *job* is a processes that is under the control

of a command shell. Since jobs are connected to command shells, they are slightly easier to manipulate than other processes.

Processes are very important in Unix, so you should be very familiar with the terminology and commands associated with Unix processes.

Processes and terminals

A *terminal* is an I/O device, which basically represents a text-based terminal device. Terminals (or *ttys*) play a special role in Unix, as they are the main method of interaction between a user and text-based programs. Traditionally terminals were physical devices; today we tend to use windowing systems with terminal emulators; in Unix terms, these are implemented using *pseudo-terminals* (or *ptys*), which behave like physical terminals from the program's point of view, but are really only implemented in software.

A process in Unix may have a *controlling terminal*. The controlling terminal is inherited when a new process is created, so all processes with a common ancestry share the same controlling terminal. For example, when you log in, a command shell is started with a controlling terminal representing the terminal or window you logged in on; processes created by the shell inherit the same controlling terminal. When you log out, all processes with the same controlling terminal as the process you terminated by logout are sent the HUP signal (see below).

A process with a controlling terminal can be controlled from the keyboard. The default settings in Unix are that crtl+Z suspends a process, crtl+C terminates it and crtl+\ aborts it (terminates with extreme prejudice). This is actually implemented by having the terminal driver intercept the key presses and sending predefined signals to the process.

Foreground, background and suspended processes

The distinction between foreground and background processes is mostly related to how the process interacts with the terminal driver. There may be at most one foreground process at a time, and this is the process which receives input and signals from the terminal driver. Background processes may send output to the terminal, but do not receive input or signals. If a background process attempts to read from the terminal it is automatically suspended. It is shown like this in the terminal:

A process that is suspended is not executing. It is essentially frozen in time waiting to be woken. Processes are suspended by sending them the TSTP or STOP signals. The TSTP signal can be sent by typing crtl+Z when the process is in the foreground (assuming standard shell and terminal settings). The STOP signal can be sent using the kill command. A process which is suspended can be resumed by sending it the CONT signal (e.g. using fg, bg or kill).

Sometimes it is desirable to run a process in the background, detached from its parent and from its controlling terminal. This ensures that the process will not be affected by its parent terminating or a terminal closing. Processes which run in the background like this are called *daemons*, and the logic that detaches them is in the program code itself. Some shells (e.g. zsh) have a feature that allows the user to turn any process into a daemon.

Signals

The simplest form of inter-process communication in Unix are signals. These are content-free messages sent between processes, or from the operating system to a process, used to signal

exceptional conditions. For example, if a program attempts to violate memory access rules, the operating system sends it a SEGV signal (known as a segmentation fault).

There is a wide range of signals available, and each has a predefined meaning (there are two user- defined signals, USR1 and USR2 as well) and default reaction. By default, some signals are ignored (e.g. WINCH, which is signalled when a terminal window changes its size), while others terminate the receiving program (e.g. HUP, which is signalled when the terminal a process is attached to is closed), and others result in a core dump (dump of the process memory; e.g. SEGV, which is sent when a program violates memory access rules).

Programs may redefine the response to most, but not all, signals. For example, a program may ignore HUP signals, but it can never ignore KILL (kill process) ABRT (process aborted) or STOP (suspend process). A few process related commands are given in the next page.

Command	Purpose
ps aux	List all running processes.
kill -*signal pid*	Send signal number *signal* to process with ID *pid*. Omit *signal* to just terminate the process. If *pid* has the form %*n*, then send signal to job *n*.
kill -9 *pid*	Send signal number 9 (SIGKILL) to process with ID *pid*. This is a last-resort method to terminate a process.
pkill *pattern*	Kill all processes that match *pattern*. By default, only the command name is searched for *pattern*.
jobs	Display running jobs.
^C	Interrupts (terminates) the process currently in the foreground.
^Z	Suspends the process currently running in the foreground.
CONTROL S	Stops output in the active terminal (this is not strictly process control, but output control).
CONTROL Q	Resumes output in the active terminal.

command &	Runs *command* in the background.
bg	Resumes a suspended process in the background. If the process needs to read from the terminal, it will be suspended again.
fg	Brings a process in the background to the foreground. This will resume the process if it is currently suspended.

Editing and viewing files

There are lots of text editors available for Linux. Regardless of which text editor your prefer, it is useful to have a working knowledge of vi , since it is shipped with almost every Unix variant that exists. You should learn vi to the point where you can edit text files, but there is no point in becoming an expert – you only need to know enough so you can get a system to the point where you can install emacs.

Inexperienced Unix users tend to load text files into editors to view them. The problem with opening text files in an editor is that you might accidentally *change* them. In this course, please use the appropriate commands to view files rather than opening them in editors.

To display a short file, use the cat command. Simply typing cat *filename* will display the file named *filename*.

Practically all Unix systems come with a so-called pager. A pager is a program that displays text files one page at a time. The default pager on most Unix systems is named ***more***. To display a text file (named *filename*) one page at a time, simply type:

```
more filename
```

You can use more to display the output of any program one page at a time. For example, to list all files that end in ".h" on the system, one page at a time, type:

```
find / -name '*.h' –print | more
```

If you try this you may notice that you can only move forward in the output – more will not let you move back and forth. You may also notice that more exits when the last line of output has been displayed.

The preferred alternative to more is called less. It is not installed by default, but it is worthwhile installing it as soon as you can on a new system. less has several advantages over more, chief of which is that it allows paging forwards and backwards in any file, even if it is piped into less. It also has better search facilities. Learn about less by reading the man page. Typing 'h' in less will display a list of keyboard commands.

Sometimes it is convenient to edit a file without using an interactive editor. This is often the case when editing files from shell scripts, or when making a large number of systematic changes to a file. Unix includes a number of utilities that can be used to non-interactively edit a file. Read the man pages for sed, awk, cut and past e for detailed information about some of the more useful commands. Here are some common examples:

```
sed –e 's/REGEX/REPLACEMENT/g' < INFILE > OUTFILE
```

Replace all occurrences of *REGEX* in *INFILE* with *REPLACEMENT*, and write the output to *OUTFILE*. This is probably the most common use of sed.

```
awk –e '{ print $2 }' < INFILE
```

Print the second column of *INFILE* to standard output. The column separator can be changed by setting the FS variable. See the awk manual for details.

```
cut –d: -f1 < /etc/passwd
```

Print all user names in /etc/passwd (really, print the first column in /etc/passwd, assuming that columns are separated by colons).

Shell scrips

Writing shell scripts is one of the most widely used tasks that might be required for any security administrator. There are lots of mundane tasks that will required to execute on day to day basis. These can be automated using shell scripts. In the following we will learn the basic of writing shell scripts.

The whole purpose of this script is nothing else but print "Hello World" using **echo** command to the terminal output. Using any text editor to create a new file named **hello.sh** containing the below code:

```
#!/bin/bash

echo "Hello World"
```

Once ready, make your script executable with the `chmod` command and execute it using relative path `./hello.sh` :

```
$ chmod +x hello-world.sh
$ linuxconfig.org:~$ ./hello-world.sh
Hello World
$
```

Examples of Some other programming are given below

```
#!/bin/bash

greeting="Welcome"
user=$(whoami)
day=$(date +%A)

echo "$greeting back $user! Today is $day, which is the best day of the entire week!"
echo "Your Bash shell version is: $BASH_VERSION. Enjoy!"
```

By now you should possess all required skills needed to create a new script, making it executable and running it on the command line. After running the above `welcome.sh` script, you will see an output similar to the one below:

```
$ ./welcome.sh
Welcome back linuxconfig! Today is Wednesday, which is the best day of the entire we
Your Bash shell version is: 4.4.12(1)-release. Enjoy!
```

:

Bash Shell Numeric and String Comparisons

Description	Numeric Comparison	String Comparison
less than	-lt	<
greater than	-gt	>
equal	-eq	=
not equal	-ne	!=
less or equal	-le	N/A
greater or equal	-ge	N/A
Shell comparison example:	*[100 -eq 50]; echo $?*	*["GNU" = "UNIX"]; echo $?*

```bash
#!/bin/bash

# This bash script is used to backup a user's home directory to /tmp/.

user=$(whoami)
input=/home/$user
output=/tmp/${user}_home_$(date +%Y-%m-%d_%H%M%S).tar.gz

# The function total_files reports a total number of files for a given directory.
function total_files {
        find $1 -type f | wc -l
}

# The function total_directories reports a total number of directories
# for a given directory.
function total_directories {
        find $1 -type d | wc -l
}

tar -czf $output $input 2> /dev/null

echo -n "Files to be included:"
total_files $input
echo -n "Directories to be included:"
total_directories $input

echo "Backup of $input completed!"

echo "Details about the output backup file:"
ls -l $output
```

```bash
#!/bin/bash

string_a="UNIX"
string_b="GNU"

echo "Are $string_a and $string_b strings equal?"
[ $string_a = $string_b ]
echo $?

num_a=100
num_b=100

echo "Is $num_a equal to $num_b ?"
[ $num_a -eq $num_b ]
echo $?
```

```bash
#!/bin/bash

num_a=100
num_b=200

if [ $num_a -lt $num_b ]; then
    echo "$num_a is less than $num_b!"
fi
```

```bash
#!/bin/bash

num_a=400
num_b=200

if [ $num_a -lt $num_b ]; then
    echo "$num_a is less than $num_b!"
else
    echo "$num_a is greater than $num_b!"
fi
```