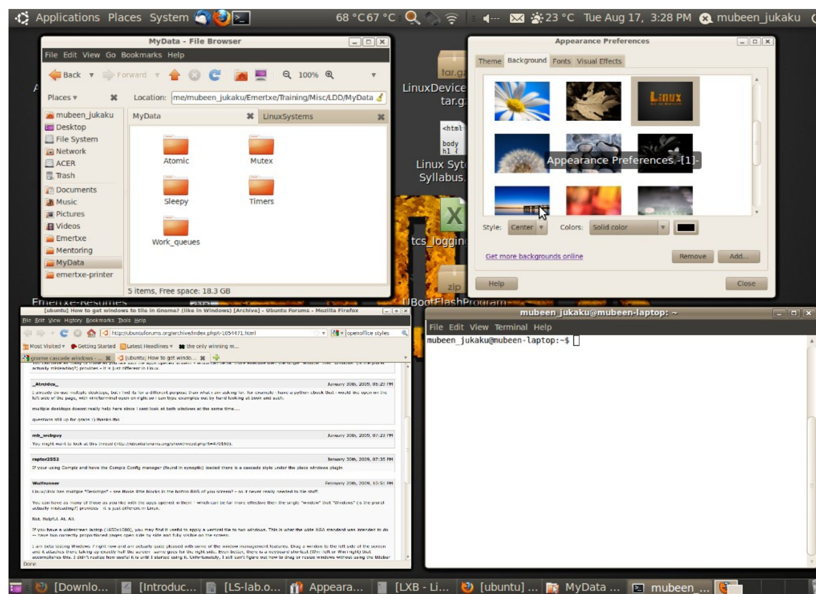
Table of Contents

Introduction 2
Part 1: The console and logging in 2
Part 2: Linux fundamentals 3
About the command line 3
Run Command (ALT – F2)..4
Principles of unix commands 5
Documentation..6
Files and directories..7
File and path names..8
File system permissions...8
File types...9
Manipulating access rights..9
Symbolic links..10
File and Directory Manipulation Commands..10
The Command shell...12
Using the shell efficiently...12
Command history...12
Command line editing...13
Tab completion..13
Environment and shell variables...13
Recording Commands and Output..14
Redirecting output..14
Processes and jobs..16
Processes and terminals..16
Signals...17
Foreground, background and suspended processes..17
Part 3: Archives and compressed files..18
Compressed files...18
Archives..19
Part 4: Editing and viewing files..19
Looking at files...20
Non-interactive text editors..20
Part 5: System logging...21
Part 7: The Linux boot process...21

Linux Systems LAB

Introduction



This lab will introduce you to the basics of using Linux systems. If you are already comfortable with Linux systems, you will find the lab easy. This lab is a prerequisite to any lab using the Linux systems, and you will be expected to know everything in the lab by heart.

This lab is mandatory, even for students who feel they already know everything they need to know about using Unix.

Start the lab by sitting down at any lab workstation. The workstation should be displaying a single window where you can type your username and password. If that is not the case, or the workstation appears to be running Microsoft Windows, please alert the lab assistant.

This lab is not supposed to be a stumbling stone – it is supposed to help you complete the rest of the labs a little faster. If you find an exercise particularly difficult, and feel you are getting nowhere with it on your own, please talk to a lab assistant, who will try to guide you through the exercise.

Part 1: The console and logging in

Linux (indeed all Unix-like) systems make heavy use of text consoles. In modern environments, text consoles are usually implemented using windows in a graphical desktop environment, whereas in the past, consoles were dedicated devices, typically capable of displaying 24 or 25 lines of 80 or 132 characters each.

In a standard Linux installation, you can use both graphical and text-based displays. Most Linux installations offer eight text-based consoles and one graphical display, all on the same screen (only one can be displayed at a time). Each console has its own settings (character/graphical mode, graphical parameters, font etc). These devices are known as virtual consoles or vc:s, since they aren't physical devices like the dedicated terminals of the past. You can switch between virtual consoles using Ctrl-Alt-F1 (hold CONTROL and ALT, and then F1), Ctrl-Alt-F2, Ctrl-Alt-F3, Ctrl-Alt-

F4, Ctrl-Alt-F5, Ctrl-Alt-F6, Ctrl-Alt-F7, Ctrl-Alt-F8 and Ctrl-Alt-F9 for vc:s 1 through 9.

Exercise 1: Console switching

Use the keyboard combinations listed above to switch between virtual consoles. Note that it may take a few seconds for the change to be completed, particularly when the graphical console is involved. After you are done, switch to the graphical console.

1.1 Which virtual console displays the graphical environment?
1.2 Which virtual consoles display a login prompt?
1.3 Are the virtual consoles that are not graphical and do not display a login prompt used for anything? If not, can you think of a use for them?

Report: Answers to the questions above.

Note:
In order to use a Linux workstation, you are required to present your user name and password. You can select your login name from the Graphical menu. Or you can type your login ID. Login ID is of the form batch-id_Name (eg: 10099_mubeen), where 10099 is the batch-id and mubeen is the name.

Part 2: Linux fundamentals

This section covers the fundamentals of using Linux (Unix-like) systems. If you are already comfortable with Unix or Linux, most of the exercises should be easy. Note that this section is required, regardless of your previous experience.

About the command line

Although the command line does take longer to learn than good graphical tools do, the command line has several distinct advantages over graphical tools, particularly on Linux-like systems. The command line offers speed and flexibility that graphical tools have been unable to meet. For simple operations, the difference may be difficult to notice, but as needs and experience increases, the difference becomes obvious. What graphical file browser (or other standard tool) would be capable of editing all user's web browser configuration files, perhaps to update the list of available printers or the location of shared plug-ins, in a single operation? Using the command line, such operations are not only possible: with experience they become second nature.

Finally, graphical tools aren't always available. Many system administrators regularly work with remote systems, sometimes located on the other side of the world and sometimes in the next room. In many cases, graphical tools are not available or unusable (e.g. when connecting through a serial line); in others they are inconvenient due to network lag or other reasons. Under such circumstances, the command line is the only available option.

KDE includes a program called Konsole, which emulates a text terminal. Use Konsole (or at your own option, some other terminal emulator) to run commands. Konsole can manage several terminal sessions at once, shown as tabs at the bottom of the window. Gnome desktop has a terminal emulator called gnome-terminal-emulator.

Exercise 2: Using konsole

2-1 Start Konsole by clicking the Kde start menu -> Tools -> Konsole
2-2 How can you create additional tabs in konsole?

2-3 How can you rename tabs?
2-4 How do you close tabs?
2-5 Find out what Office applications are installed by clicking KDE Menu -> Office

Report: No report required.

Note: Openoffice is the most commonly used office application in Linux. You can view and edit Microsoft Word, Powerpoint, Excel documents using openoffice.

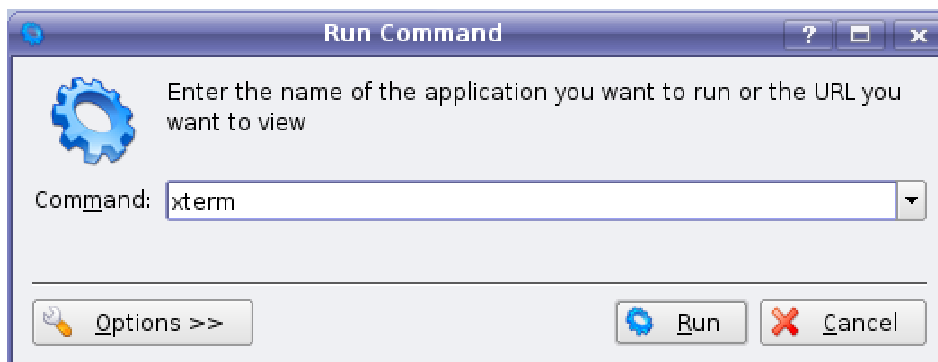
MS Windows Office Program	OpenOffice Program	Openoffice command
MS Word	OO Writer Word Processor	oowriter
MS Powerpoint	OO Impress Presentation	ooimpress
MS Excell	OO Calc Spreadsheet	oocalc

Other Programs:

Application	Program name	Command
PDF Viewer	Okular (KDE Desktop) Evince (Gnome Desktop) Acrobat reader	okular evince
File Browser	Dolphin (KDE) Konqueror (KDE) Nautilus (GNOME)	dolphin konqueror nautilus
Web Browser	Firefox	firefox
Calculator	Kcalc (KDE) gcalctool (GNOME)	kcalc gcalctool
Graphics Drawing	Gimp Image Editor Inkscape Vector Graphics	gimp
Email/Calendaring software (MS Outlook)	Evolution Thunderbird	

Run Command (ALT – F2)

The quickest way to run a command on Linux is to use the Alt+F2 shortcut key. This pops up a dialog that looks like this in KDE:



You can type any command that you would regularly use from a terminal window and it will run.

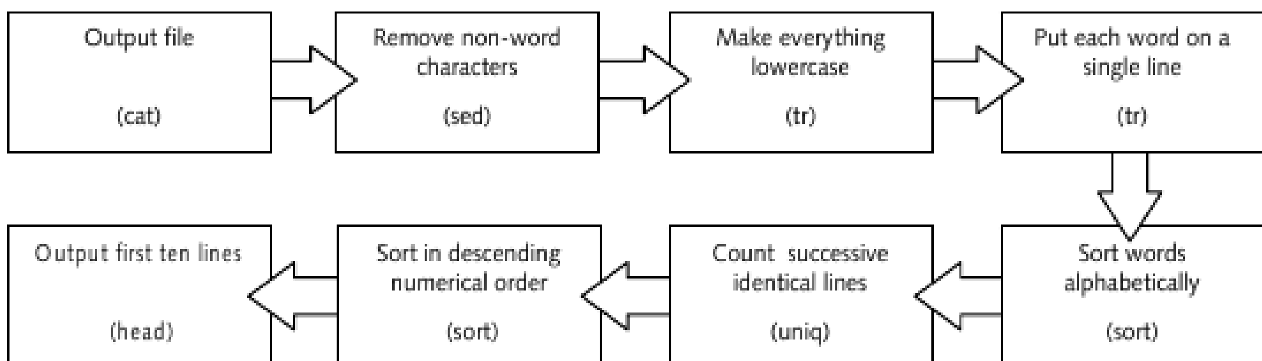
Exercise 3: Run Command (ALT+F2)

3-1 Start Konsole using Run command
3-2 Start the PDF Viewer application
3-3 Start the Oo Writer application
3-4 Start the File explorer application
3-5 Start the web-browser

Note: Running all the applications simultaneously can slow down the system. Hence, close unwanted programs.

Principles of unix commands

One of the fundamental principles of Unix is that tools should be small, designed to do one thing only, and do it well. Complex functions could then be built by combining tools by directing the output of one tool to another. For example, Unix has no command that will display all words in a text file, including the number of time each word appears. Instead, one would combine several commands to accomplish the same thing:




```
cat textfile | sed ‘s/[^[:alpha:][:space:]]/ /g’ | \tr ‘[:upper:]’ ‘[:lower:]’ | tr –s ‘ \t’ ‘ ‘ | sort | uniq –c \
sort –rn | head -10
```

Each command does one simple thing: cat reads a file from disk; sed edits the file; tr changes one set of characters to another; sort sorts the data; uniq removes duplicates from sorted data and counts occurrences; and head outputs the first several lines of its input. Connected in the proper order, they perform a far more complex task: list the ten most common words in a text file. Several commands connected like this are often called a pipeline. Data flows through the commands like oil through a pipeline, and is processed at various points. The connectors between commands are called pipes.

Pipes are the most common method for connecting commands. The command to the left of the pipe produces output that the command on the right uses as its input. Sometimes commands (and API functions) are written as name(n), where n is a number, possibly followed by a letter or two. This notation is used to distinguish between different things that have the same name. The name is simply the name of the command, file, or API function. The number indicates the manual section (see below) in which the name is documented. For example, tty(1) refers to the user command tty, whereas tty(4) refers to the device driver with the same name.

Documentation

The skill to rapidly and effectively navigate, read and understand documentation is probably the most important skill a system administrator can have. Unix documentation is organized in so called man pages. Each man page documents a command, API call, file format, device or concept. Pages are divided into sections; section one is for user commands, section two for system calls, section three for higher-level API calls and so on.

Before moving on to more advanced tasks, you should become comfortable reading man pages, and referring to the man pages should 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.

You read man pages using the man command. In traditional Unix systems, the man command would in turn execute a pipeline consisting of nroff (to format the text) and more (to display text one page at a time). The man command itself was responsible for locating the correct file.

Exercise 4: The man command

4-1 Execute the command man man. What do you see?
- (a) What does the –a option to man do?
- (b) What does the –k option to man do?
- (c) What option should you use to just print a short description of a command?
- (d) What options shows the location of the man page rather than its contents?

4-2 Display the man page for the ls command.
- (a) What does the ls command do?
- (b) What option to ls shows information about file sizes, owner, group, permissions and so forth?
- (c) What does the –R option to ls do? (Don't forget to try it.)

4-3 To display a list of all manual pages containing the keyword "date", what command would you type?

4-4 The Linux Manual is divided into nine main sections.
- (a) What type of information does section 1 contain?
- (b) What type of information does section 4 contain?
- (c) What type of information does section 5 contain?
- (d) What type of information does section 8 contain?

4-5 Every man page is divided into several parts.
- (a) What information can be found in the FILES part?
- (b) Which part is used to describe what a command does?
- (c) In which part can you find details about what command-line options a command accepts?
- (d) Which part lists related commands?
- (e) Many man pages include examples. Which part are they found in, and approximately where in the man page do you usually find it?

4-6 What does the apropos command do?

Report: Brief written answers to all questions.

In addition to man pages, systems that use the Gnu utilities (such as Linux) also have an info manual. Info files are typically more suitable for on-line browsing than man pages are. Many of the Gnu commands have brief man pages, but are fully documented in the info manual. Such commands usually have a reference to the appropriate info file in the man page. Use the command info to display info files. Within info, type a question mark to see help on using info.

Exercise 5: Using info

5-1 Start info without any options. What do you see? What info commands are mentioned on the page you are shown?

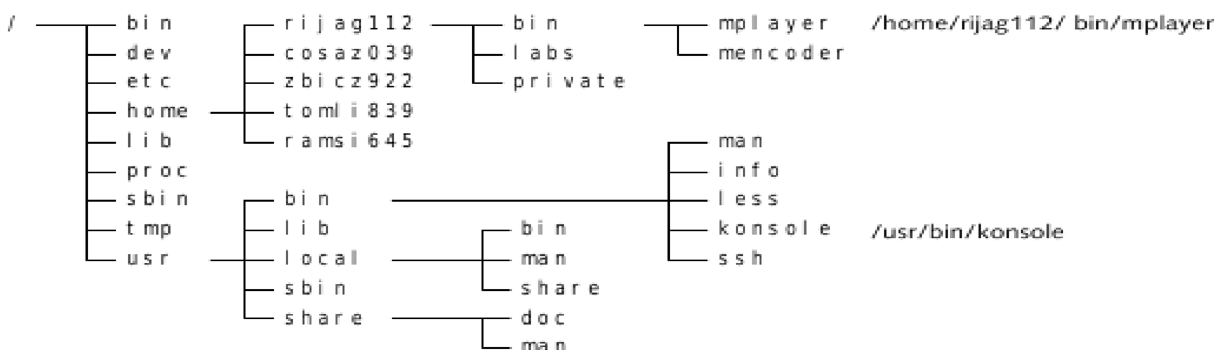
5-2 View the documentation for ls in info (using the command info coreutils ls).You can find the reference to info at the bottom of the ls man page. Give at least three examples of information that is in the info manual, but not in the man page.

5-3 When browsing info, how can you search for text?

Files and directories

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 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.*



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.

Exercise 6: Absolute and relative path names

6-1 In the example above:
- (a) What is the absolute path name of mplayer?
- (b) What is the absolute path name of ssh?

6-2 In the example above name at least one relative path name indicating ssh if
- (a) The current working directory is /usr/bin.
- (b) The current working directory is /usr/local/bin.
- (c) The current working directory is /home/rijag112/bin.

Report: Answers to all questions.

File system permissions

Linux, as many other modern operating systems have methods of administrating permissions or access rights to individual or groups of users. These permissions affect how users can make changes to the file system.

There are three groups of permissions on UNIX type systems: "user", "group" and "others". The "user" group grants permissions for the owner of a file or directory. The "group" group grants permissions for members of the file or directory's group and the "others" group grants permissions for all other users. Each group can have three main permission bits set (there are others, but that is an advanced topic): "read", "write" or "executable". The read bit (r bit) grants permission to read a file or directory tree. The write bit (w bit) grants permission to modify a file. If this is set for a directory it grants permission to modify the directory tree, including creating or (re)moving files in the directory or changing their permissions. Finally, the executable bit (x bit) grants permission to execute a file. The x bit must be set in order for any file to be executed or run on a system (even if the file is a executable binary). If the x bit is set on a directory, it grants the ability to traverse the directory tree.

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**). For example, to see the permissions set for the file "foobar" in the current directory has, write:

```
% 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 individual permissions. Permissions can be specified using either “long” format or a numeric mode (all permission bits together are called the files 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 foobar we would do as follows to disable write permission, then verify that the change has taken place:

Exercise 7: Long format chmod

7-1 It is possible to set individual permissions for user, group and others using chmod. Review the documentation and answer the following questions:

- (a) How can you set the permission string to user read/write, group read, others read using chmod in long format?
- (b) How can you revoke group write permissions on a file without changing any other permissions?
- (c) How can you grant user and group execute permissions without changing any other permissions?

Report: Answers to the questions above.

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.

Exercise 8: Numeric file modes

8-1 What do the following numeric file modes represent:
- (a) 666
- (b) 770
- (c) 640
- (d) 444

8-2 The claim that there are only nine permission bits (three each for user, group and others) is not quite true. There are more permission bits. What are they?

8-3 What command-line argument to chmod allows you to alter the permissions of an entire directory tree?

8-4 A user wants to set the permissions of a directory tree rooted in dir so that the user and group can read and write files, but nobody else has any access. Which of the following commands is most appropriate? Why?
- (a) chmod –R 660 dir
- (b) chmod –R 770 dir
- (c) chmod –R u+rw,g+rw,o-rwx dir

Report: Answers to the questions above.

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.

Exercise 9: Owner and group manipulation

9-1 How can you change the owner and group of an entire directory tree (a directory, its subdirectories and all the files they contain) with a single command?

Report: Answers to the questions above.

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.

Unix also supports a concept called “hard linking”, which makes it possible to give a file several different names (possibly in different directories) that are entirely equal (i.e. there is no concept of “target”, as all names are equally valid for the file).

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 the exercise.

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 C	*Creates a symbolic link, so that oldname can also be accessed as newname.*

Exercise 10: File manipulation commands

10-1 What does cd .. do?
10-2 What does cd ../.. do?
10-3 If you do cd / followed by pwd, what will happen?
10-4 What information about a file is shown by ls -laF?

10-5 In the following example, explain the fields of the output from ls -laFd dir dsp:
drwxr-xr-x 22 dave staff 4096 Jan 12 2001 dir/
crw-rw---- 1 root audio 14, 3 Jan 22 2001 dsp
10-6 If you have two files, a and b, and you issue the command mv a b, what happens? Is there an option to mv that will issue a warning in this situation?
10-7 What is the command to duplicate the contents of /dir1 to /dir2, preserving modification times, ownership and permissions of all files?
10-8 How do you make the file secret readable and writable by root, readable by the group wheel and inaccessible to everybody else?
10-9 How can you remove a directory, including its contents, with a single command?
10-10 What does chown –R user.user /path/to/directory/ do?
10-11 How can you recognize symbolic links when using ls?
10-12 How can you see what a symbolic link points to?
10-13 What happens if you attempt to create a symbolic link to a file that doesn't exist?

Report:Written answers to all questions.

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 (tcsh).

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 tcsh 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.

Exercise 11: Shell init files

11-1 Run echo $SHELL to find out what shell you are using.
11-2 What init files does your shell use, and when are they used?

Report: The answer to question 14-2.

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 <Ctrl-P> and <Ctrl-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 <Ctrl-R> and then the word you want to search for.

To simply repeat the previous command, you can type !! as the command. This is a throwback to older shells, where an exclamation mark meant you wanted to repeat an old command in some way.

Command line editing

Edit the command line using emacs-like key bindings: <Ctrl-A> moves to the beginning of the line, <Ctrl-E> to the end. Move forward and backwards using <Ctrl-F> and <Ctrl-B>. <Ctrl-D> deletes the character under the cursor and <Ctrl-K> deletes to the end of the line.

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.

Exercise 12: Tab completion

12-1 Create the following files by using the touch command: READFIRST, READSECOND README, READMORE, and GUGGENHEIM.
12-2 What happens if you type cat R and then hit <TAB>?
12-3 What happens if you hit <TAB> twice more?
12-4 What happens if you type an M, then hit <TAB>? If you hit <TAB> again?
12-5 What happens if you delete the M, then type an S, then hit <TAB>?
12-6 Abort the current command by typing <Ctrl-C>. Make sure you are at an empty prompt.
12-7 What happens if you type au, then hit <TAB> twice? What is going on here?

Report: Answers to the questions above.

Environment and shell variables

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. Shell variables are often used as temporary variables when writing shell scripts (programs written in the shell command language).

All (useful) Unix shells support variable expansion. This process replaces part of a command line with the contents of an environment 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.

Environment and shell variables are altered using shell syntax:

NAME = VALUE	POSIX syntax. Sets the variable NAME to VALUE. Does not necessarily set the environment variable (shell dependent).
export NAME	POSIX 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).
setenv NAME VALUE	C shell syntax. Sets the environment variable NAME to VALUE. Use set instead of setenv to set a shell variable.

Exercise 13: Manipulating environment variables

13-1 Use the env command to display all environment variables. What is PATH set to (you might want to use grep to find it)? What is this variable used for (the man pages for your shell might be helpful in answering this question)?
13-2 Use echo to display the value of HOME. What does the HOME variable normally contain?
13-3 Set the variable TEST to the string “This is a test”. Note that you will have to quote the string since it contains space characters.
13-4 Output the value of TEST.
13-5 Prepend /home/TDDI05/bin to the variable PATH. The easiest way to accomplish this is to use variable expansion in the right-hand side of the assignment.

Report: Answers to 13-1 and 13-2. The commands used in 13-3, 13-4 and 13-5

Recording Commands and Output

If you want to record all commands you run, and the output they produce, the script utility is very helpful. When run, it executes a shell, but captures all input to and output from the shell, recording it to a file. Read the man page for the script utility and complete the following exercises.

Exercise 14: Using the script utility

14-1 Set the value of TEST to the string “noscript”.
14-2 Run script ~/typescript to start the script utility.
14-3 Execute ifconfig –a in the command shell.
14-4 Set the value of TEST to the string “script”.
14-5 Terminate the script utility by typing exit.
14-6 Check the value of TEST. Explain why it is set to “noscript” and not “script”.
14-7 View the file ~/typescript and compare with the commands you ran.

Report: The contents of the typescript file and the answer to 17-6.

In future exercises use the script utility whenever asked for commands and their corresponding output. Use output redirection or the tee command when only output is requested.

Redirecting output

Consistent with the idea that every command should do one well-defined task well, and that commands should be combined to perform more complex tasks, 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. More advanced

operations involve direct manipulation of file descriptors from the command line. 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).

You can redirect output from a command to a file using the > or >> operators. Redirection is only valid for a single execution of the command; there are no facilities for permanently redirecting the output of a particular command.

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.

command >> filename

The output of command is appended to filename. The file will be created if it doesn’t already exist. These basic redirection commands only redirect standard output (file descriptor one); they do notredirect 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 file descriptor 2 (stderr) is written to filename. The file will be created if it does not already exist, and any previous contents will be overwritten.

command 2>> filename

The output of command to file descriptor 2 (stderr) is appended to filename. The file will be created if it does not already exist.

command 2>&1

The output of file descriptor 2 (stderr) is redirected to whatever file descriptor 1 (stdout) points to. Technically, file descriptor 1 is duplicated to file descriptor 2. This is often used to redirect stderr and stdout to the same place. Note that the order of redirections is important!

Exercise 15: Redirecting output

15-1 Where will stdout and stderr be redirected in the following examples? If you want to test your theories, use /home/TDDI05/bin/stdio for command. This program outputs a series of E:s to stderr (file descriptor 2) and a series of O:s to stdout (file descriptor 1).

(a) command >file1
(b) command 2>&1 >file1
(c) command >file1 2>&1

Report: The answers to 18-1.

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 (stdout) from command1 is used as the input (stdin) of command2. Note that this connection is made before any redirection takes place.

Exercise 16: Pipelines

16-1 What do the following commands do?
- (a) ls | grep –i doc
- (b) command 2>&1 | grep –i fail
- (c) command 2>&1 >/dev/null | grep –i fail

16-2 Write composite commands to perform the following tasks:
- (a) Output a recursive listing (using ls) of your home directory, including invisible files, to the file /tmp/HOMEFILES.
- (b) Find any files (using find) on the system that are world-writable. Error messages should be discarded (redirected to /dev/null).
- (c) Find all files in /etc that contain either the string "10.17.1" or the string "130.236.189" and output their names to /tmp/FILES. Any error messages should be discarded. For this exercise you may want to use egrep and a regexp containing the infix operator "|".
- (d) Output a recursive listing (using ls) of your home directory, including invisible files, to the file /tmp/HOMEFILES and to the screen. You may find the tee command useful here.

16-3 Output the contents of the first file found in /etc that contains the string "10.17.1" or the string "130.236.189". You can combine find, grep, head, xargs and cat to get the job done. Read the man pages for the commands you aren't familiar with.

Report: Answers to 16-1 and the solutions in 16-2 and 16-3.

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. All major general-purpose operating systems today share these properties. Every program that is executed results in at least one process. Each process has a process identifier, has its own memory area not shared with other processes, and shares resources with other processes based on its priority. Jobs are processes that are under the control of a command shell (command shells read and respond to user input, and are discussed later). Jobs 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. Terminals (more accurately pseudo-terminals or ptys) are also an inter-process communications channel: a program may open a pty for writing and another may open the same pty for reading; as far as both are concerned they are communicating with a terminal,

and can use all terminal control features, such as enabling and disabling echo, timeouts and so forth.

A process in Unix may have a controlling terminal. The controlling terminal is inherited when a new process is created, ensuring that 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).

Signals

The most fundamental 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, a the operating system signals a process that it has violated memory access rules by sending it a SEGV signal (known as 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 signaled when a terminal window changes its size), while others terminate the receiving program (e.g. HUP, which is signaled 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).

When a process is attached to a terminal, the terminal driver can send signals to the process in response to user key presses. The default settings in Unix are that <Ctrl-Z> sends TSTP (suspend),<Ctrl-C> sends TERM (terminate) and <Ctrl-\ > sends ABRT (abort).

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.

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 <Ctrl-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.

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.
jobs	Display running jobs.
<Ctrl-C>	Interrupts (terminates) the process currently in the foreground.
<Ctrl-Z>	Suspends the process currently running in the foreground.
<Ctrl-S>	Stops output in the active terminal (this is not strictly process control, but output control).
<Ctrl-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.

Exercise 17: Processes and jobs

17-1 Create a long running process by typing ping 127.0.0.1. Suspend it with vZ and bring it to the foreground with fg. Terminate it with vC.

17-2 Create a long running process in the background by typing ping 127.0.0.1 >/dev/null&. Find out its process id using ps and kill it using kill.

17-3 Sometimes ps aux truncates the process name. How can you get ps to display the full process name and its arguments?

17-4 What does the command kill -9 pid do, where pid is the number of a process? Are there other options than -9 that might be useful? What does kill -9 -1 do?

Report: Answers to the questions above.

Part 3: Archives and compressed files

When working with Unix (or Linux) you are bound to encounter archives and compressed files (and compressed archives). For example, most of the Debian package documentation is compressed to save space, and source code is typically distributed in archive form.

Compressed files

In the Linux world the two most popular compression standards are gzip and bzip2. A gzip compressed file usually has a .gz file name extension, whle a bzip2 compressed file ends in .bz2. In more venerable Unix-like systems, you will see the .Z file extension, which indicates a file compressed with the compress command.

Command	Purpose
zcat FILENAME.gz *bzcat FILENAME.bz2*	Output the uncompressed contents of FILENAME.gz or FILENAME.bz2 to stdout.
gzip –d FILENAME.gz *bzip2 –d FILENAME.bz2*	Uncompress FILENAME.gz or FILENAME.bz2, removing the compressed file and leaving the uncompressed file in its place.
gzip FILENAME *bzip2 FILENAME*	Compress FILENAME using gzip or bzip2.

Note that unlike compression utilities that are popular on the Windows platform, gzip and bzip2 compress single files. Combining several files into an archive is the job of another program. The choice of gzip or bzip2 depends on how portable you need to be. At the moment, gzip has a far larger installed base than bzip2. If portability is not a consideration, bzip2 performs slightly better than gzip.

Archives

To combine several files into a single archive, Unix users almost exclusively use the tar utility. tar was originally designed to create a byte stream that could be written to tape in such a way that individual files could be extracted. Hence the name – Tape ARchiver. Today, tar is mostly used to combine files into a single disk-based archive, but you will still find tar used to make tape backups in many smaller systems.

Recent versions of tar have gzip and bzip2 compression built-in. The compressed archives can be read directly by tar or decompressed using gzip and bzip2.Here are some examples of typical uses of tar. See the man page for details.

tar xvf FILENAME.tar
tar xvfz FILENAME.tar.gz
tar xvfj FILENAME.tar.bz2

Extract (x) all files from FILENAME.tar (f) and print information about every file (v). If the archive is compressed with gzip, use the z option. If it is compressed with bzip2, use the j option.

tar tvf FILENAME.tar
tar tvfz FILENAME.tar.gz
tar tvfj FILENAME.tar.bz2

Display the contents (t) of the file FILENAME.tar (f) in verbose (v) mode.The z and j options are used for compressed archives.

tar cvf FILENAME.tar DIRECTORY
tar cvfz FILENAME.tar.gz DIRECTORY
tar cvfj FILENAME.tar.bz2 DIRECTORY

Create (c) a tar archive named FILENAME.tar (f) containing the directory (and its contents) DIRECTORY. The z and j options result in compressed archives.

Part 4: Editing and viewing files

It is very important to be able to use at least one text mode editor. Knowing how to use an editor on your system is the first step to independence.

Vim is a text editor which includes almost all the commands from the LINUX program **vi** and a lot of new ones.Commands in the **vi** editor are entered using only the keyboard, which has the advantage that you can keep your fingers on the keyboard and your eyes on the screen. You can

use the vimtutor to learn first Vim commands. This is a thirty minute tutorial that teaches the most basic Vim functionality. You can start this program from the shell or command line, entering the **vimtutor** command.

Looking at files

Inexperienced Unix users tend to load text files into editors to view them. This works well enough that some never learn a better way. The problem with opening text files in an editor is that you might accidentally change it. To display a short file, use the cat command. Simply typing cat filename will display the file named filename.

A pager: more

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.

A better pager: less

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.

Exercise 18: Using the pager less... eh, using the pager named less

18-1 What keystroke in *less* moves to the beginning of the file?
18-2 What keystroke in *less* moves to the end of the file?
18-3 What would you type in *less* to start searching for "baloney"?
18-4 What would you type in *less* to move to the next match for "baloney"?
18-5 How can you use *less* to monitor a log file, so less keeps displaying new lines in the file as they are written to the end of the file?
18-6 How would you read the compressed file README.Debian.gz in less?

Report: Answers to the questions above.

Non-interactive text editors

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 paste 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).

Part 5: System logging

System logs are some of the most important source of information when troubleshooting a problem, or when testing a system. Most Unix services print diagnostic information to the system logs.

Logging is managed by the syslogd process, which is accessed through a standard API. By default, the syslog process outputs log messages to various log files in /var/log, but it is also possible to send log messages over the network to another machine. It is also possible to configure exactly which log messages are sent to which files, and which are simply ignored. For the purpose of this course, the default configuration is sufficient. It creates a number of log files, the most important of which are: /var/log/auth.log for log messages related to authentication (e.g. logins and logouts); /var/log/syslog and /var/log/messages contain most other messages; mail.log contains log messages from the mail subsystem. For details on what goes where, see /etc/syslog.conf.

Since log files grow all the time, there needs to be a facility to remove old logs. In Debian/Gnu Linux, a service called logrotate is commonly used. It "rotates" log files regularly, creating a series of numbered log files, some of which are compressed. For example, you may see the files /var/log/auth.log, /var/log/auth.log.0, /var/log/auth.log.1.gz and /var/log/auth.log.2.gz on a system. /var/log/auth.log is the current log file. /var/log/auth.log.0 is the next most recent and so forth.

To test these exercises you may need to use a UML system as you may lack sufficient permissions to see the log files on the lab workstation.

Exercise 24: Log files

24-1 What does less –F /var/log/syslog do?
24-2 If you want to extract the last ten lines in /var/log/syslog that are related to the service cron, what command would you use?

Report: Answers to the questions above.

In the labs, please use the log files as much as possible. When you encounter problems, chances are that there will be information related to the problem in the log files. Make it a habit to always scan the end of /var/log/syslog every time you reconfigure and restart a service to see that the service is not reporting any problems.

Part 7: The Linux boot process

When the Linux kernel loads, it starts a single user process: init. The init process in turn is responsible for starting all other user processes. This makes the Linux boot process highly configurable since it is possible to configure the default init program, or even replace it with

something entirely different.

The init process reacts to changes in run level. Run levels define operating modes of the system. Example include “single user mode” (only root can log in), “multi-user mode with networking”, “reboot” and “power off”. In Debian/Gnu Linux, the default run level is run level 2. Other Linux distributions and other Unix-like systems may use different default run levels.

The actions taken by init when the run level changes are defined in the /etc/inittab file. The default configuration in most Linux distributions is to use something called “System V init” to manage user processes. When using System V init, init will run all scripts that are stored in a special directory corresponding to the current run level, named /etc/rcN.d, where N is the run level. For example, when entering run level 2, init will run all scripts in /etc/rc2.d.

Scripts are named SNNservice or KNNservice. Scripts whose names start with K are kill scripts and scripts whose names start with S are start scripts. When entering a run level, init first runs all kill scripts with the single argument stop, and then all start scripts with the single argument start.

For example if the directory /etc/rc5.d contains the following scripts: K10nis, K20nfs and S10afs, init would first execute /etc/rc5.d/K10nis stop, then /etc/rc5.d/K20nfs stop, then /etc/rc5.d/S10afs start.

When Linux boots it start by changing to run level S (single user mode), then to run level 2. This implies that all scripts in /etc/rcS.d and in /etc/rc2.d are run when the system boots, and more importantly that all services that are started are started by scripts in one of these directories.

In Debian/Gnu Linux, all the scripts in /etc/rcN.d are actually symbolic links to scripts in /etc/init.d. For example, /etc/rc2.d/S20ssh is a symbolic link pointing to /etc/init.d/ssh. This is so that changes to the scripts need to be made in a single file, not in one file per run level. It also means that if you want to start or stop a service manually, you can use the script in /etc/init.d rather than try to remember its exact name in any particular run level.

/etc /init.d /SERVICE start
Start the service named SERVICE (e.g. ssh, nis, postfix).
/etc /init.d /SERVICE stop
Stop the service named SERVICE.
/etc /init.d /SERVICE restart
Restart SERVICE (roughly equivalent to stopping then starting).
/etc/ init.d/ SERVICE reload
Reload configuration for SERVICE (does not work with all services).

Sometimes it is useful to see exactly how a service is started or stopped (e.g. when startup fails). To see all the commands run when a service starts, run the script using the sh –x command (works for nearly all startup scripts, but is not guaranteed to always work).

sh –x /etc/init.d/SERVICE start
Start SERVICE, displaying each command that is executed.

Exercise 19: Mostly hard stuff

19-1 Where will stdout and stderr be redirected in the following examples
(a) command 2>&1 >file1

(b) command >file1 2>&1

19-2 Write a command that lists, for the last ten unique users to log in to the system, the last time they logged in using ssh (users can be found using last, and ssh logins in /var/log/auth.log, which is typically only readable by root).

19-3 Explain the following fairly contrived code, in particular all the I/O redirections:

```
#!/bin/sh
exec 23<&0 24>&1 <<__EOG > /tmp/RECORD
`find / -name "*.bak" –print`
__EOG
while read f ; do
        echo –n "Record $f" >&24
        read ans <&23
        [ "$ans" = "y" ] && echo $f
done
```

19-4 Using only ps, grep and kill, write a command that kills any process with “linux” in the name. Note that you must avoid killing the grep process itself!

19-5 For each file on the system with a .bak suffix, where there either is no corresponding file without the .bak suffix, or the corresponding file is older than the .bak file, ask the user whether to rename the .bak file to remove the suffix. If the answer begins with Y or y, rename the file appropriately. You must handle file names containing single spaces correctly.

19-6 Write a command or short script that replaces all occurrences of “10.17.1” with “130.236.189” in all files in /etc or a subdirectory thereof. The only output from the command must be a list of the files that were changed.

19-7 Write a command or short script that replaces all occurrences of “10.17.1” with “130.236.189” in all files in /etc or a subdirectory thereof. The only output from the command must be a list of the files that were changed.

19-8 Examine the files ~/TDDI05/lxb/passwd and ~/TDDI05/lxb/shadow. Use paste and awk to output a file where each line consists of column one from passwd and column two from the corresponding line in shadow. The printf function in awk is helpful here.

19-9 Write a command that renames a bunch of files to be all lowercase (e.g. BOFH.GIF is renamed to bofh.gif). If there already is a file with the all-lowercase name, do not rename, print an error message and proceed with the next file. You do not have to worry about spaces (but you should, just on general principle).