LXB

Linux Basics

Goals of this lab:

- ❖   Learn the basics of the Linux command line
- ❖   Learn to manage files and directories
- ❖   Learn about the command shell
- ❖   Learn about environment variables
- ❖   Learn about processes and job control

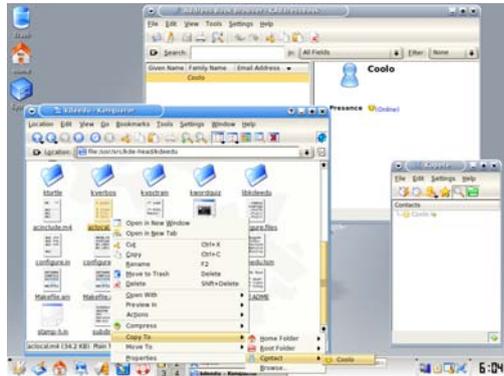Prerequisites: None

Table of Contents

PRELAB

Exercise 1: Learning the basics

1-1   Read chapter 1 of *Essential System Administration* (or the equivalent)*.*

1-2   Read the KDE Quick Start guide at http://www.kde.org/documentation/

Report: No report is required.

MAIN 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. In the past, when this lab was optional, even most students who *didn't* know enough about using Unix would skip the lab, resulting in problems further on in the course. Because of this, we have decided to make the lab mandatory. If you already know everything, you should be able to work through these exercises in less than two hours.

However, you also have the choice of doing only part part 8: expert exercises and skipping the rest. You should only consider this if your Unix skills are already very good, because there are some *hard* exercises in there.

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.

Time taken 2005: 1.5-10 hours, average 5 hours

Past problems: There haven't been many significant problems with this lab, but some people find it too easy (while other find parts of it too hard), and the variation in difficulty among the exercises is disliked by some (and praised by others). The fact that part 3 is so much larger than the others came as a nasty surprise for some. All in all, if you follow the instructions and use the lab assistant when appropriate, you shouldn't have too much trouble with this lab.

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], then type [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 2: 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.

2-1   Which virtual console displays the graphical environment?

2-2   Which virtual consoles display a login prompt?

2-3   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.

In order to use a Linux workstation, you are required to present your user name and password. Depending on precisely how the computer is set up the dialog used to enter this information may differ. A workstation using a graphical login, like the ones in the lab, present a single window where you can enter your user name and password, and also select the type of session you want – more on that later. Note that unlike web-based services and certain other operating systems, Unix systems often lack feedback when you type at the password prompt.

Use your regular student user name and password to access the Linux workstation. Using the graphical login, enter your user name and password. Check that "KDE" or "Default" is selected in the session menu, then click "Login". You should eventually be presented with a desktop environment.

Part 2: The desktop environment

The X Window System

Unlike Microsoft Windows, the window system is not an integral part of a Linux system, but an independent component, much like any other. The base for most graphical user interfaces in Linux and other Unix-like systems is the X Window System version 11, also known as X Windows, X11 or simply X. X provides the core functions needed to build a complete graphical user interface, but it is not, in itself, a complete environment. A complete X-based environment includes a number of other components that are based on X: toolkits provide widgets such as buttons and menus that users can interact with, and a window manager draws window decorations and manages policies for manipulating windows, input focus and much more.

One of the most powerful features of the X Window System is that it is a network-based client-server system. Clients in X are end-user applications such as word processors and photo editors. The X server manages input and output devices, and typically resides on the hardware device the end user interacts with. This means that users can run X applications on any computer connected to the internet, and have the graphical interface displayed on their personal workstation.

On the client side, the X server and the X protocol used to communicate with the server are abstracted by Xlib, which provides a consistent API for applications. Xlib is a low-level API, so most applications are written to a toolkit that abstracts and extends Xlib.
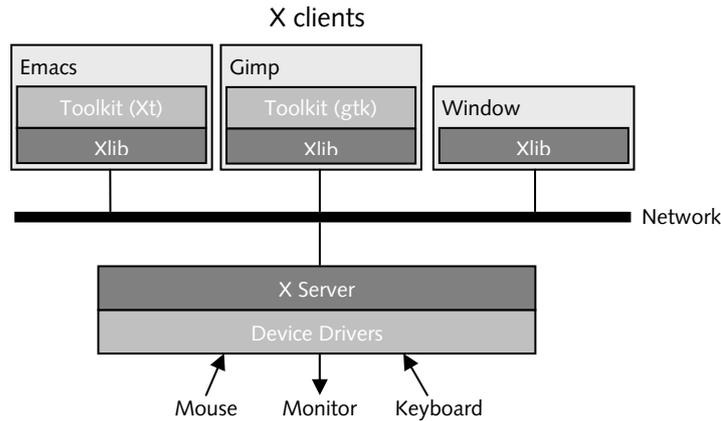
X clients

The fact that X is network-based has led people to believe that it is slow, since using the network prevents clients from accessing hardware and memory on the display device directly. The reality is that most X servers today include extensions that allow clients running locally (on the same hardware as the server) to access graphics memory and hardware directly. A well-written application for the X Window System can be take advantage of local access when available, and revert to slower network-based methods when running on a remote system. Furthermore, certain hardware-accelerated features, such as OpenGL, can be accessed remotely, allowing applications on remote systems to take full advantage of the hardware where the application output is displayed.

The K Desktop Environment

Although X with a window manager creates a fully functional graphical user interface, users and developers have come to expect more. KDE is a product that is designed to meet these expectations. KDE provides standard applications, such as a web and file browser (Konqueror), an office suite (KOffice), network transparency (applications can open files through a variety of network services), a multimedia architecture and various high-level APIs and components. Overall, KDE gives provides users with a consistent experience.

Since KDE is based on X, standard X applications, as well as applications designed for other desktop environments such as Gnome or CDE can run within KDE, but they will not take advantage of the services offered by KDE.
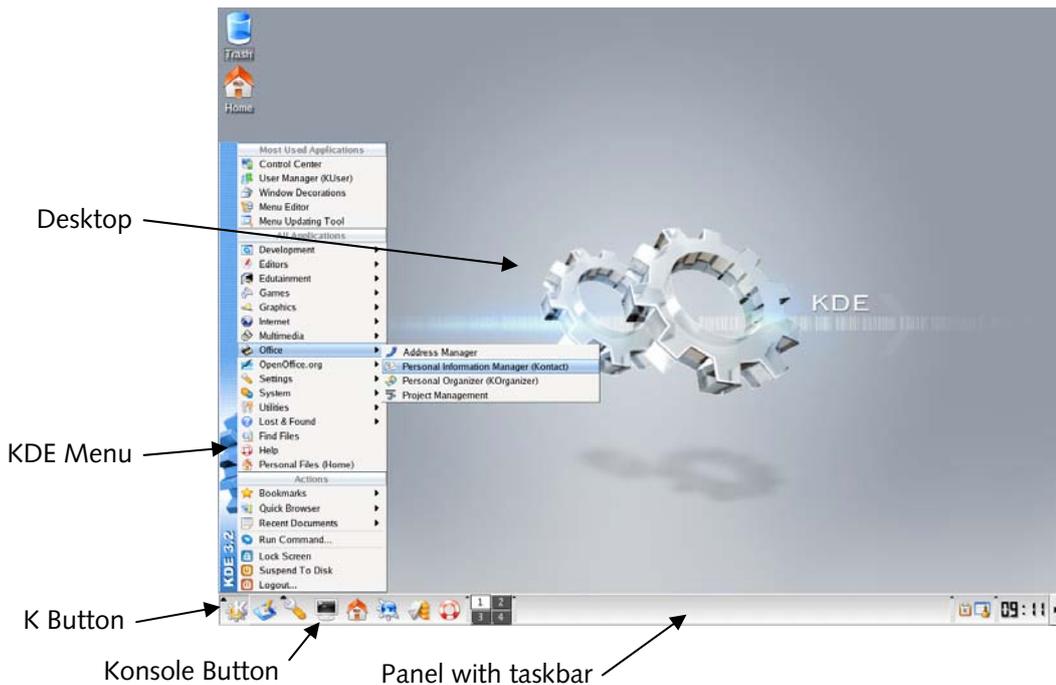
KDE comes with extensive user documentation. The documentation is available both on-line and on the web. Since KDE will be an important tool for you, you should make sure that you are comfortable using it. Read the on-line documentation, and most importantly, *try* the things you read about.

The following exercises are optional in TDDI05, as most people already know how to use a window system. However, if you're not used to KDE, doing these might be a good idea anyway.

Exercise 3: KDE basics

3-1   Where can you get help for KDE and KDE applications?

3-2   Change the background picture of KDE. How did you do it?

3-3   What are the rectangles labeled 1, 2, 3 and 4 in the panel used for?

3-4   What happens when you double-click the title bar of a window?

3-5   Place the cursor over a window, hold down the Alt key, then drag the mouse while pressing the right mouse button. What happens?

3-6   Sometimes it can be difficult to hit the title bar of a window to move it. How can you move a window without having to first move the cursor to the title bar?

3-7   What do the various buttons in the standard window title bars do?

Report: Answers to the questions above, if you want to.

Although graphical user interfaces are very useful tools, they are rarely enough for system administrators. Most system administrators will use graphical environments for everyday office tasks such as writing reports, reading e-mail and goofing off, but use command-line tools for most system administration tasks, from moving files to adding users to running backups. Some extremists simply view the graphical environment as a way to fit more terminal windows onto the same screen.

Part 3: Unix fundamentals

This section covers the fundamentals of using 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 Unix-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.

In these labs, you will be working with virtual Linux systems that behave as if they were remote computers accessed through a serial line. Until you have configured networking and installed the

requisite software, these systems have no graphical tools at all. You have no choice other than to use the command line.

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.
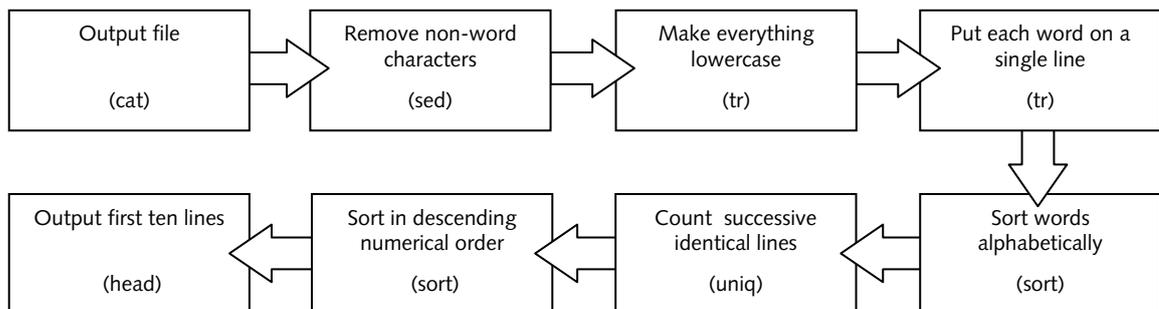
Exercise 4: Using konsole

4-1   Start Konsole by clicking the Konsole icon in the lower panel.

4-2   How can you create additional tabs?

4-3   How can you rename tabs?

4-4   How do you close tabs?

Report: No report required.

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 print 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:

Output file (cat)	→	Remove non-word characters (sed)	→	Make everything lowercase (tr)	→	Put each word on a single line (tr)

Output first ten lines (head)	←	Sort in descending numerical order (sort)	←	Count successive identical lines (uniq)	←	Sort words alphabetically (sort)

```
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 5: The man command

5-1   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?

5-2   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.)

Report: Brief written answers to all questions.

Exercise 6: The Unix manual

6-1   The Unix 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?

6-2   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?

6-3   What does the `apropos` command do?

Report: Brief written answers to the questions above.

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 7: Using info

7-1   Start info without any options. What do you see? What info commands are mentioned on the page you are shown?

7-2   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.

7-3   When browsing info, how can you search for text?

Report: Written answers to the questions above.

From this point on you will be expected to read documentation to solve most of your problems. In general, if you have any questions, try to get the answers from the documentation before calling on a lab assistant. If you haven't checked, or haven't checked thoroughly enough, you will be directed to the documentation by the assistant.

Package documentation

Every Debian package comes with its own documentation. These files are located in subdirectories of /usr/share/doc/. For every package that you install, you will want to look in this directory for README files, Debian-specific documentation (very important) and examples.

Most of these files are normal text files. If you do not know how to navigate the file system and display text files yet, finish the rest of these exercises, than return to this section.

Exercise 8: Reading package documentation

8-1   Locate the documentation for the iproute package and answer the following questions:

(a)   What kernel version is required to run the Debian iproute package?

(b)   Is DECNET name resolution supported in iproute?

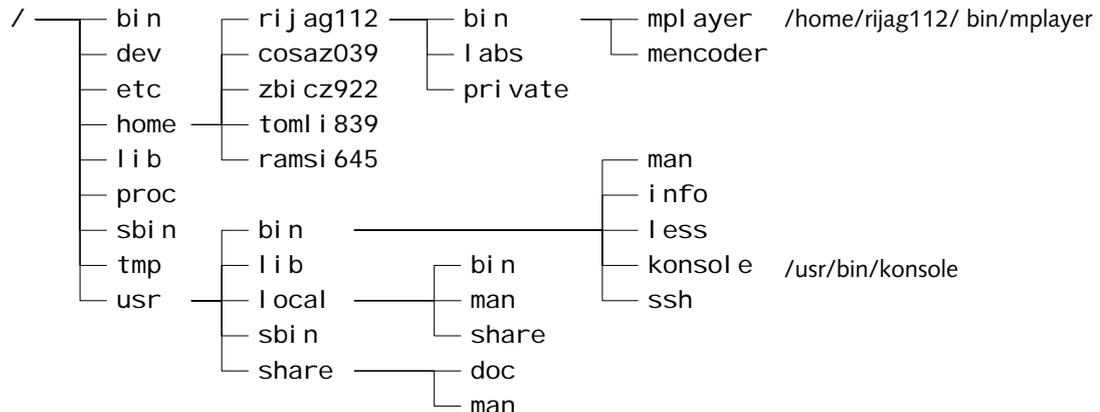**Report:**   Answers to the questions above.

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.

The figure below shows part of a Unix system.

```
/ ──── bin       rijag112 ──── bin      mplayer    /home/rijag112/ bin/mplayer
     ─ dev       cosaz039     ─ labs     mencoder
     ─ etc       zbicz922     ─ private
     ─ home      tomli839
     ─ lib       ramsi645
     ─ proc                              man
     ─ sbin      bin ───────────────     info
     ─ tmp       lib                     less
     ─ usr       local ──── bin          konsole    /usr/bin/konsole
                 sbin      ─ man          ssh
                 share      share
                           doc
                          ─ man
```

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 9: Absolute and relative path names

9-1   In the example above:

> (a)   What is the absolute path name of mplayer?

> (b)   What is the absolute path name of ssh?

9-2   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:

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

To enable write access again, replace the minus sign with a plus sign (chmod +w foobar).

Exercise 10: Long format chmod

10-1   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 11: Numeric file modes

11-1   What do the following numeric file modes represent:

      (a)   666

      (b)   770

      (c)   640

      (d)   444

11-2   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?

11-3   What command-line argument to chmod allows you to alter the permissions of an entire directory tree?

11-4   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 12: Owner and group manipulation

12-1　　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*	Creates a symbolic link, so that *oldname* can also be accessed as *newname*.

Exercise 13: File manipulation commands

13-1   What does `cd ..` do?

13-2   What does `cd ../..` do?

13-3   If you do `cd /` followed by `pwd`, what will happen?

13-4   What information about a file is shown by `ls -laF`?

13-5   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
```

13-6   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?

13-7   What is the command to duplicate the contents of /dir1 to /dir2, preserving modification times, ownership and permissions of all files?

13-8   How do you make the file `secret` readable and writable by `root`, readable by the group `wheel` and inaccessible to everybody else?

13-9   How can you remove a directory, including its contents, with a single command?

13-10   What does `chown -R user.user /path/to/directory/` do?

13-11   How can you recognize symbolic links when using `ls`?

13-12   How can you see what a symbolic link points to?

13-13   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`).

You *must* run these exercises in the bash shell, or results will not be as expected. You have two simple options: either run the exercises in a UML instance (this assumes that you know how to start your UML systems) or start bash manually, by issuing the command `bash` in a terminal window.

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 `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 14: Shell init files

14-1   Run `echo $SHELL` to find out what shell you are using.

14-2   What init files does your shell use, and when are they used?

Report: The answer to question 14-2.

Using the shell efficiently

⚠ You *must* run these exercises in the bash shell, or results will not be as expected. You have two simple options: either run the exercises in a UML instance (this assumes that you know how to start your UML systems) or start bash manually, by issuing the command `bash` in a terminal window.

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 [CONTROL][P] and [CONTROL][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 [CONTROL][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: [CONTROL][A] moves to the beginning of the line, [CONTROL][E] to the end. Move forward and backwards using [CONTROL][F] and [CONTROL][B]. [CONTROL][D] deletes the character under the cursor and [CONTROL][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 15: Tab completion

15-1   Create the following files by using the touch command: READFIRST, READSECOND README, READMORE, and GUGGENHEIM.

15-2   What happens if you type `cat R` and then hit [TAB]?

15-3   What happens if you hit [TAB] twice more?

15-4   What happens if you type an M, then hit [TAB]? If you hit [TAB] again?

15-5   What happens if you delete the M, then type an S, then hit [TAB]?

15-6   Abort the current command by typing [CONTROL][C]. Make sure you are at an empty prompt.

15-7   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

⚠️ You *must* run these exercises in the bash shell, or results will not be as expected. You have two simple options: either run the exercises in a UML instance (this assumes that you know how to start your UML systems) or start bash manually, by issuing the command `bash` in a terminal window.

Unix, and many other operating systems, including Windows NT/2000/XP/2003 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. 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 16: Manipulating environment variables

16-1   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)?

16-2   Use `echo` to display the value of `HOME`. What does the `HOME` variable normally contain?

16-3   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.

16-4   Output the value of `TEST`.

16-5   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 16-1 and 16-2. The commands used in 16-3, 16-4 and 16-5

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 17: Using the script utility

17-1   Set the value of `TEST` to the string "noscript".

17-2   Run `script ~/typescript` to start the script utility.

17-3   Execute `ifconfig -a` in the command shell.

17-4   Set the value of TEST to the string "script".

17-5   Terminate the script utility by typing exit.

17-6   Check the value of TEST. Explain why it is set to "noscript" and not "script".

17-7   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

You *must* run these exercises in the bash shell, or results will not be as expected. You have two simple options: either run the exercises in a UML instance (this assumes that you know how to start your UML systems) or start bash manually, by issuing the command `bash` in a terminal window.

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 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 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 18: Redirecting output

18-1   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 19: Pipelines

19-1   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`

19-2   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.

19-3   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 19-1 and the solutions in 19-2 and 19-3.

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 ⌨CONTROL Z sends TSTP (suspend), ⌨CONTROL C sends TERM (terminate) and ⌨CONTROL \ 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 ⌨CONTROL 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.

Process-related commands

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.
CONTROL C	Interrupts (terminates) the process currently in the foreground.
CONTROL 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.

Exercise 20: Processes and jobs

20-1   Create a long running process by typing `ping 127.0.0.1`. Suspend it with CONTROL Z and bring it to the foreground with `fg`. Terminate it with CONTROL C.

20-2   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`.

20-3   Sometimes `ps aux` truncates the process name. How can you get `ps` to display the full process name and its arguments?

20-4   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 4: 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.

There are a multitude of other ways to use tar, but the three variants listed above are sufficient for most users.

Part 5: Editing and viewing files

Although `emacs` is the undisputed king of all text editors, you will sometimes be relegated to something inferior. In these cases, it is useful to be familiar with the workings of the editor, if only to avoid disasters. There are lots of editors available for Unix, but the one that regrettably never seems to let go is the `vi` editor. It is a display-oriented interactive text editor with a somewhat odd user interface that is available in different variants on every Unix dialect known to man (and probably on some others as well). 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`.

In many distributions of Linux, there are simpler editors available for those of us who just can't stand using `vi`. On the lab systems, `nano` is the simple editor of choice. `nano` is more or less self-explanatory, but you should be aware that by default `nano` wraps long lines, which is a very bad idea when editing configuration files and scripts.

Exercise 21: Using the full-screen text editor

21-1   Open the file `/home/TDDI05/lxb/nano-tutorial` in `nano` and follow the instructions in the file.

21-2   Run `/home/TDDI05/bin/vilearn` to start an extensive `vi` tutorial and follow the instructions. Go through as much or as little as you want.

21-3   Modify your shell init files so `/home/TDDI05/bin` is included in the `PATH` environment variable. Ensure that "." (the current directory) is *not* included in `PATH`.

21-4   Create a configuration file for `nano` that disables automatic line breaking.

Report: The changes made to your shell init files and your `nano` configuration file.

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 22: Using the pager less… eh, using the pager *named* less

22-1   What keystroke in `less` moves to the beginning of the file?

22-2   What keystroke in `less` moves to the end of the file?

22-3   What would you type in `less` to start searching for "baloney"?

22-4   What would you type in `less` to move to the next match for "baloney"?

22-5   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?

22-6   How would you read the compressed file README.Debian.gz in less?

22-7   Locate the package documentation for the ssh package and answer the following questions by reading the README.Debian.gz file:

      (a)   What is the default setting for ForwardX11?

      (b)   If you want X11 forwarding to work, what other package(s) need to be installed on the server?

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).

Exercise 23: Using non-interactive text editors

23-1   Use `sed` to change all occurrences of "/bin/tcsh" to "/bin/sh" in /etc/passwd (output to a different file).

23-2   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. *This exercise is optional since it goes beyond the basics.*

Report: The command line used in 23-1.

Part 6: 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/rc*N*.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 S*NN*service or K*NN*service. 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.

Debian/Gnu Linux includes a command named update-rc.d that can be used to manipulate start scripts.

Part 8: Expert exercises

This section is optional. If you do complete this section correctly in less than four hours, you do not have to do any of the other sections. Note that the lab assistant will *not* help you solve any of the following exercises. Your solutions must not only be correct; they also need to be good.

Note that although you are not required to do the other exercises in this lab, you are expected to know how to do them.

Exercise 25: Mostly hard stuff

25-1   Where will stdout and stderr be redirected in the following examples

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

25-2   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).

25-3   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
```

25-4   I/O redirection in the shell is somewhat limited. In particular, you can't handle programs that write directly to the terminal device rather than use the file descriptors. For situations like that, there's socat. Since you're an advanced user, you really should learn about socat.

        (a)   Use socat so you can log in and execute commands on a host non-interactively using the password authentication method. Essentially, you should be able to do the equivalent of cat commandfile | ssh user@remote.host, where commandfile contains the stuff you would normally type interactively.

        (b)   Do something else with socat that's neat. Come up with something on your own, perhaps using the network.

25-5   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!

25-6   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.

25-7   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.

25-8   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.

25-9   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).

FEEDBACK FORM   LXB

Complete this feedback form **individually** at the end of the lab and hand it to the lab assistant when you finish. Your feedback is essential for improving the labs. Each student should hand in a feedback form. Do not cooperate on completing the form.

You do not need to put your name on the feedback form. Your feedback will be evaluated the same way regardless of whether your name is on it or not. Your name is valuable to us in case you have made and comments in the last section that need clarifications or otherwise warrant a follow-up.

For each section, please rate the following (range 1 to 5 in all cases).

❖ **Difficulty:** Rate the degree of difficulty (1=too easy, 5=too difficult)

❖ **Learning:** Rate your learning experience (1=learned nothing, 5=learned a lot).

❖ **Interest:** Rate your interest level after completing the part (1=no interest, 5=high interest).

❖ **Time:** How long did the part take to complete (in minutes)?

	Difficulty	Learning	Interest	Time (minutes)
Prelab				
Part 1: The console and logging in				
Part 2: The desktop environment				
Part 3: Unix fundamentals				
Part 4: Archives and compressed files				
Part 5: Editing and viewing files				
Part 6: System logging				
Part 7: The Linux boot process				
Overall				

Please answer the following questions:

❖ What did you like about this lab?

❖ What did you dislike about this lab?

❖ Make a suggestion to improve this lab.

Complete this feedback form **individually** at the end of the lab and hand it to the lab assistant when you finish. Your feedback is essential for improving the labs. Each student should hand in a feedback form. Do not cooperate on completing the form.

You do not need to put your name on the feedback form. Your feedback will be evaluated the same way regardless of whether your name is on it or not. Your name is valuable to us in case you have made and comments in the last section that need clarifications or otherwise warrant a follow-up.

For each section, please rate the following (range 1 to 5 in all cases).

❖ **Difficulty:** Rate the degree of difficulty (1=too easy, 5=too difficult)

❖ **Learning:** Rate your learning experience (1=learned nothing, 5=learned a lot).

❖ **Interest:** Rate your interest level after completing the part (1=no interest, 5=high interest).

❖ **Time:** How long did the part take to complete (in minutes)?

	Difficulty	Learning	Interest	Time (minutes)
Prelab				
Part 1: The console and logging in				
Part 2: The desktop environment				
Part 3: Unix fundamentals				
Part 4: Archives and compressed files				
Part 5: Editing and viewing files				
Part 6: System logging				
Part 7: The Linux boot process				
Overall				

Please answer the following questions:

❖ What did you like about this lab?

❖ What did you dislike about this lab?

❖ Make a suggestion to improve this lab.

Complete this feedback form **individually** at the end of the lab and hand it to the lab assistant when you finish. Your feedback is essential for improving the labs. Each student should hand in a feedback form. Do not cooperate on completing the form.

You do not need to put your name on the feedback form. Your feedback will be evaluated the same way regardless of whether your name is on it or not. Your name is valuable to us in case you have made and comments in the last section that need clarifications or otherwise warrant a follow-up.

For each section, please rate the following (range 1 to 5 in all cases).

❖ **Difficulty:** Rate the degree of difficulty (1=too easy, 5=too difficult)

❖ **Learning:** Rate your learning experience (1=learned nothing, 5=learned a lot).

❖ **Interest:** Rate your interest level after completing the part (1=no interest, 5=high interest).

❖ **Time:** How long did the part take to complete (in minutes)?

	Difficulty	Learning	Interest	Time (minutes)
Prelab				
Part 1: The console and logging in				
Part 2: The desktop environment				
Part 3: Unix fundamentals				
Part 4: Archives and compressed files				
Part 5: Editing and viewing files				
Part 6: System logging				
Part 7: The Linux boot process				
Overall				

Please answer the following questions:

❖ What did you like about this lab?

❖ What did you dislike about this lab?

❖ Make a suggestion to improve this lab.