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: UML

Table of Contents

PREPARATION

Exercise 1: Learning the basics

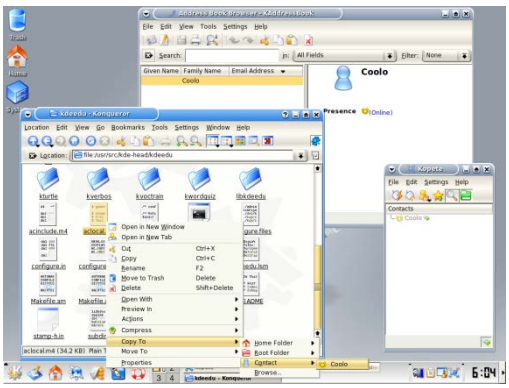
1-1 Read chapter 1 of *Essential System Administration* (or the equivalent).

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

1-3 Read through this lab.

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 6: expert exercises and skipping the rest. You should only consider this if your Unix skills are already good, because there are some hard exercises in there.**

This lab should be done on a Linux system, such as marsix.ida.liu.se or on a virtual machine (some of the exercises may require root access). You should complete the UML lab before this lab.

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 2006: 5-12 hours, average 8.5 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.

⚠ The examples and exercises in this lab are designed for the command interpreter *bash*, which is commonly used in Linux. However, at IDA, the standard command interpreter is *tcsh*, which is not compatible with bash. Furthermore, several of the commands in this lab are only available to the user *root*. If you have problems with the labs, try using the lab-1 UML instances, where bash is standard and you can log in as root.

Part 1: 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 – a fairly typical situation in many real-world installations. 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.

Your desktop environment will have a program that emulates a terminal: in KDE it is console; in Gnome it is gnome-terminal; generic programs like xterm or rxvt work on all X-Windows-based systems (rxvt also runs on Microsoft Windows).

Exercise 2: Using the terminal emulator

2-1 Start a terminal emulator, such as console, gnome-terminal, rxvt or xterm.

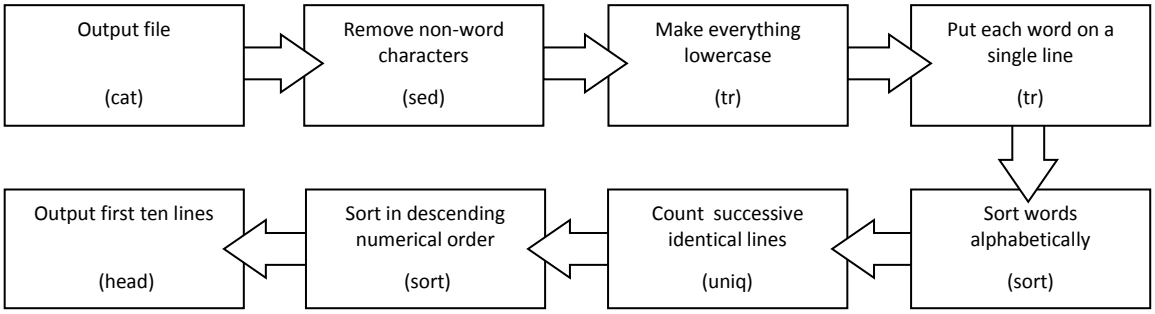
2-2 Does the program support multiple sessions (e.g. tabs)? If so, how do you create new sessions? How do you delete them?

2-3 What character encoding does the terminal expect (common ones are UTF-8 and ISO-8859-1, also known as latin-1)?

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 can then be built by combining tools by directing the output of one tool to another. For example, Unix has no command that will print the ten most common words in a text file, including the number of time each word appears. Instead, one would combine several commands to accomplish the task:



```
cat textfile | sed 's/[^[:alpha:][:space:]]/ /g' | \
    tr '[:upper:]' '[:lower:]' | tr -s ' \t' '\n' | 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 (other methods include pipes and named sockets). The command to the left of the pipe produces output that the command on the right uses as its input.

Writing complex commands like this is a bit like programming, but with the fundamental Unix commands and the built-in features of the command interpreter as the programming language.

Documentation

The skill to rapidly and effectively navigate, read and understand documentation is probably the most important skill a system administrator can have. A Gnu/Debian Linux system has three main sources of documentation: man pages, info manuals and package documentation. You need to become familiar with all three kinds!

The Unix manual

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

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

Exercise 3: Navigating the Unix manual

3-1 Answer the following questions concerning sections of the Unix Manual. Most of the information you need can be found in the man page for the `man` command (you will need to think a little too):

(a) Which are the nine sections of the Unix manual?

(b) Which section of the manual contains user commands such as `cat` and `ls`?

(c) Which section documents file formats, such as configuration files?

(d) Which section contains system administration commands, such as `halt`?

Report: Brief written answers to the questions above.

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

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

Exercise 4: Man pages

4-1 The SYNOPSIS section briefly lists how a command is invoked. Read the man page for `man` and explain what the following command synopsis texts mean. Explain how each command may or must be invoked; you don't need to know what the commands actually do:

(a) `mkpasswd PASSWORD SALT`

(b) `uniq [OPTION]... [INPUT [OUTPUT]]`

(c) `gzip [ -acdfhlLnNrtvV19 ] [-S suffix] [ name ... ]`

(d)
```
chcon [OPTION]... CONTEXT FILE...
chcon [OPTION]... [-u USER] [-r ROLE] FILE...
chcon [OPTION]... --reference=RFILE FILE...
```

4-2 Read the man page for man, as well as some other man pages (e.g. for ls, uniq, chmod, and adduser) and answer the following questions:

(a) What do you usually find in the DESCRIPTION section?

(b) Which section(s) usually document, in detail, what each command-line option does?

(c) Let's say you're reading the man page for a command and the information you are looking for isn't there. In which part can you find references to other man pages that might contain what you are looking for (have a look at the man page for reboot, imagining that you are trying to find a command that will turn the computer off, for an example)?

(d) In which section do you find information about which configuration files a program uses?

4-3 Use the man page for `man` to answer the following questions:

(a) Sometimes there are several man pages (located in different sections) for the same keyword. Which command-line option to `man` can you use to display all of them?

(b) Sometimes you don't know the name of a command you are looking for, but can guess at a keyword related to the command. Which command-line option can you use to have `man` list all pages related to a specific keyword?

4-4 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 the questions above.

The `man` command is an example of the Unix philosophy. The man command itself is only capable of locating the file containing the man page. It relies on other programs to format (traditionally `nroff`) and display them (usually `more` or `less`).

The Unix Manual is divided into nine main sections. When searching for e.g. the syntax of a configuration file, it helps to know which section such information is found in, particularly if the same name is present in more than one section. Each man page is in turn divided into several parts, such as "SYNOPSIS", "DESCRIPTION", "EXAMPLES" and "FILES". If you are familiar with the more common sections of man pages you can find information a lot faster than by trying to read the whole thing from beginning to end.

Info manuals

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.

When reading man pages, you might see something like this (in the SEE ALSO part):

```
SEE ALSO
The full documentation for ls is maintained as a Texinfo
manual. If the info and ls programs are properly installed at
your site, the command

       info ls

should give you access to the complete manual.
```

That means that the full documentation is in the info manual, not the man page. You should read the info manual instead; most of the time, info manuals are more comprehensive and more well written than man pages.

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 should look in this directory for README files, Debian-specific documentation (very important) and examples.

You can save a lot of time by making sure you always check the package documentation when you want to know something! There can be a lot of information in there, including complete configuration examples, troubleshooting tips and more.

Most of these files are normal text files, but there are often compressed files and HTML files.

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.

Files and directories

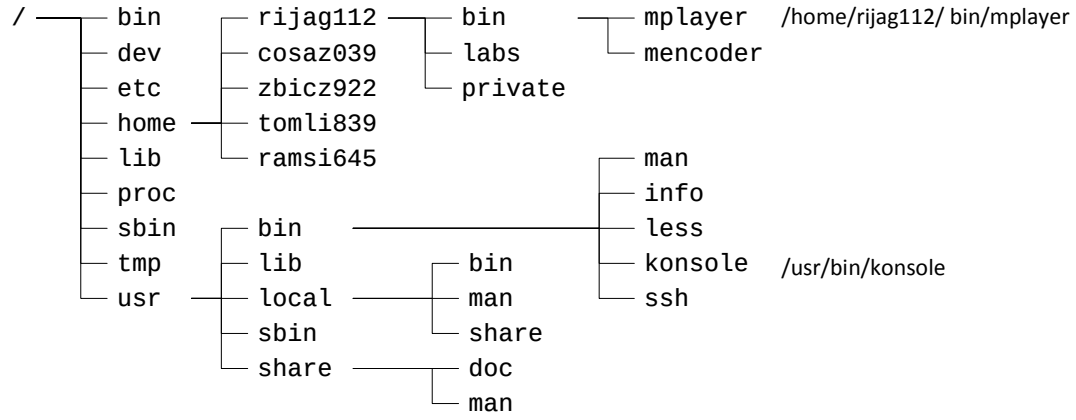
Understanding how files and directories are organized and can be manipulated is vital when using or managing a Linux system. All files and directories in Linux are organized in a single tree, regardless of what physical devices are involved (unlike Microsoft Windows, where individual devices typically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



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 5: Absolute and relative path names

5-1 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.

Report: Answers to all questions.

File system permissions

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

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

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

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

To list the permissions of a file or directory, use the `ls` command with the `-l` option (to enable long file listing; see the man page for `ls`). 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 permissions. Permissions can be specified using either "long" format or a numeric mode (all permission bits together are called the file's mode). The long format takes a string of permission values (r, w or x) together with a plus or minus sign. For example, to prevent any user from changing the file foobar we would do as follows to disable write permission, then verify that the change has taken place:

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

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

Exercise 6: Long format chmod

6-1 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 7: Numeric file modes

7-1 What do the following numeric file modes represent:

(a) 666

(b) 770

(c) 640

(d) 444

7-2 What command-line argument to `chmod` allows you to alter the permissions of an entire directory tree?

7-3 What does execute (x) permission mean on directories?

7-4 A user wants to set the permissions of a directory tree rooted in *dir* so that the user and group can list, read and write (but not execute) 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

When answering this question, consider how the execute permission is handled by the various choices, and what importance the execute permission has on directories.

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 8: Owner and group manipulation

8-1 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 9: File manipulation commands

9-1 What does `cd ..` do?

9-2 What does `cd ../..` do?

9-3 What information about a file is shown by `ls -laF`?

9-4 In the following 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
```

Explain the following:

(a) What does the "c" at the beginning of the second line mean?

(b) What do "dave" and "staff" mean on the first line, and "root" and "audio" on the second?

(c) Which users may write to the file dsp?

9-5 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?

9-6 What option(s) to `cp` allows you to copy the contents of /dir1 to /dir2, preserving modification times, ownership and permissions of all files?

9-7 How do you make the file `secret` readable and writable by `root`, readable by the group `wheel` and completely inaccessible to everybody else?

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 10: Shell init files

10-1 Run `echo $SHELL` to find out what shell you are using.

10-2 What init files does your shell use, and when are they used? (Hint: your shell has a man page, and somewhere near the end there is a section that lists which files it uses).

Report: The answer to question 10-2.

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.

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.

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

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

Environment and shell variables are altered using shell syntax:

NAME=*VALUE*

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

`export` *NAME*

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

`set env` *NAME VALUE*

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

Exercise 11: Manipulating environment variables

11-1 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)?

11-2 Use `echo` to display the value of `HOME`. What does the `HOME` variable normally contain?

11-3 Prepend `/data/kurs/adit/bin:/data/kurs/TDDI09/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 11-1 and 11-2. The commands used in 11-3

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

Redirecting I/O

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

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

I/O redirection simply is a matter of changing what the file descriptors point to.

You can redirect output from a command to a file using the > or >> operators.

command `>` *filename*

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

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

command `>>` *filename*

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

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

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

command 2> *filename*

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

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

command 2>> *filename*

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

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

command 2>&1

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

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

Exercise 12: Redirecting output

12-1 Where will stdout and stderr be redirected in the following examples? If you want to test your theories, use /data/kurs/TDDI09/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

When answering these, remember that the order of redirections matters!

Report: The answers to 12-1.

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

command1 | *command2*

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

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

command1 2>&1 | *command2*

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

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

Exercise 13: Pipelines

13-1 What do the following commands do? If you want to test your theories, use /data/kurs/TDDI09/bin/stdio for *command* and grep for "E" rather than "fail".

(a) `ls | grep –i doc`

(b) *command* 2>&1 | `grep –i fail`

(c) *command* 2>&1 >/dev/null | `grep –i fail`

13-2 Write command lines 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 (i.e. the write permission for "others" is set). Error messages should be discarded (redirected to `/dev/null`). This command is actually useful for auditing the security of a system – world-writable files can be security risks.

Report: Answers to 13-1 and the solutions in 13-2.

Processes and jobs

Linux is a multi-tasking, multi-user operating system. Several users can use the computer at once, and each user can run several programs at the same time. Every program that is executed results in at least one *process*. Each process has a process identifier and has its own memory area not shared with other processes. A *job* is a processes that is under the control of a command shell. Since jobs are connected to command shells, they are slightly easier to manipulate than other processes.

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

Processes and terminals

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

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

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

Foreground, background and suspended processes

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

```
[1] + Stopped(SIGTTIN)          command
```

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.

Signals

The simplest form of inter-process communication in Unix are signals. These are content-free messages sent between processes, or from the operating system to a process, used to signal exceptional conditions. For example, if a program attempts to violate memory access rules, the operating system sends it a SEGV signal (known as a segmentation fault).

There is a wide range of signals available, and each has a predefined meaning (there are two user-defined signals, USR1 and USR2 as well) and default reaction. By default, some signals are ignored (e.g. WINCH, which is 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).

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.
`pkill pattern`	Kill all processes that match *pattern*. By default, only the command name is searched for *pattern*.
`jobs`	Display running jobs.
vC	Interrupts (terminates) the process currently in the foreground.
vZ	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 14: Processes and jobs

14-1 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.

14-2 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.

14-3 What does the command kill -9 *pid* do, where *pid* is the number of a process? What does kill -9 -1 do? Read the documentation to figure the last one out as it is a somewhat dangerous command.

14-4 Create a long running process in the background by typing ping 127.0.0.1 >/dev/null &. Kill it using pkill. The pkill command is very useful when you need to kill several processes that share some attribute (such as a command name).

Report: Answers to the questions above and the commands executed.

Part 2: 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, while 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, usually tar, but sometimes cpio.

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 3: Editing and viewing files

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

In many distributions of Linux, there are simpler editors. 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 breaks (wraps) long lines, which is a very bad idea when editing configuration files and scripts.

Exercise 15: Using the full-screen text editor (this exercise is recommended but optional)

15-1 Open the file /data/kurs/TDDI09/labs/lxb/nano-tutorial in nano and follow the instructions in the file.

15-2 Run /data/kurs/TDDI09/bin/vilearn to start an extensive vi tutorial and follow the instructions. Go through as much or as little as you want.

15-3 Modify your shell init files so /data/kurs/TDDI09/bin is included in the PATH environment variable. Ensure that "." (the current directory) is *not* included in PATH. Hint: all the information you need for this exercise can be found in your answers to earlier exercises.

15-4 Create a configuration file for nano that disables automatic line breaking.

Report: No report is required.

Looking at files

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

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

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
```

Or to read a compressed file:

```
zcat filename.gz | 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 16: Using the pager less… eh, using the pager *named* less

16-1 What keystroke in `less` moves to the beginning of the file?

16-2 What keystroke in `less` moves to the end of the file?

16-3 What would you type in `less` to start searching for "option"?

16-4 What would you type in `less` to move to the next match for "option"?

16-5 Locate the package documentation for the ssh package and answer the following questions by reading the README.Debian.gz file (hint: remembering the answers to these questions may be useful in the project):

(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 17: Using non-interactive text editors (this exercise is optional)

17-1 Use `sed` to change all occurrences of "/bin/tcsh" to "/bin/sh" in /etc/passwd (output to a different file). *This exercise is optional.*

17-2 Examine the files shadow and passwd in the directory /data/kurs/TDDI09/labs/lxb. 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: No report is required.

Part 4: 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. A good habit to develop is to *always* look in the log files when you have reconfigured or restarted a service, just to make sure that there are no obvious problems.

Logging is managed by the `syslogd` process, which is accessed through a standard API. By default, the `syslogd` 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 server.

Exercise 18: Log files

18-1 What does `tail -f /var/log/syslog` do?

18-2 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? (Hint: the `grep` command can search for matching lines in a file).

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 5: 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.

Exercise 19: Booting Linux

19-1 What services are started when your system boots into run level 2 (i.e. not only services *exclusive* to run level 2, but *all* services started as the computer boots into run level 2)?

19-2 If you wanted to restart the ssh process, what command would you use?

Report: Answers to the questions above.

Part 6: Expert exercises

This section is optional. If you do complete this section correctly, 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 20: Mostly hard stuff

20-1 Where will stdout and stderr be redirected in the following examples

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

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

20-2 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 (this can be done by crafting the regexp appropriately)! This command line can be used to rapidly terminate any UML running on the system.

20-3 Use your shell's alias or function features to create a shell command named `gzless` that decompresses a gzip-compressed file and pipes it into `less`.

20-4 I/O redirection in the shell is somewhat limited. In particular, you can't handle programs that write directly to the terminal device (i.e. use a pseudo-terminal) 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.

20-5 Write a command that renames a bunch of files to be all lowercase (e.g. BOFH.GIF is renamed to bofh.gif). You do not *have* to worry about spaces (but you *should*, just on general principle). The `tr` command may be useful here.

20-6 Examine the files /data/kurs/TDDI09/labs/lxb/passwd and shadow (same directory). 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` may be helpful here.

20-7 Sometimes you need to change the IP address of a computer. Assuming that all the files that need to be changed are somewhere in /etc (or a subdirectory thereof), what command will *list* all relevant files and not print any error messages?

20-8 Write a command line that changes the IP address stored in the variable (shell or environment) OLD to the IP address stored in the variable NEW in all (regular) files in /etc or its subdirectories. The commands `find`, `xargs` and `sed` may be useful here.

20-9 Write a command that uses `ssh` to log in to all computers w0001 through w2000 and runs the `w` command on each one. You might want to read about arithmetic substitution in bash for this one. How could you avoid having ssh ask for a password for each computer without setting an empty password or resorting to host-based authentication?

Report: The commands used in exercises above, and answers to any questions above.

Here are some more hard exercises. You don't have to do them, but if you want a challenge, these should fit the bill pretty well.

Exercise 21: Quite hard stuff (optional)

21-1 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).

21-2 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
```

21-3 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.

Report: No report is required.

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)
Preparation				
Part 1: Unix fundamentals				
Part 2: Archives and compressed files				
Part 3: Editing and viewing files				
Part 4: System logging				
Part 5: 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.

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)
Preparation				
Part 1: Unix fundamentals				
Part 2: Archives and compressed files				
Part 3: Editing and viewing files				
Part 4: System logging				
Part 5: 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.