COMP302
Operating Systems

including:
An introduction to Unix,
and
Operating Systems Theory

Hugh Murrell

February, 2010

Contents

Chapter 1

Introduction to Unix

1.1 Credits

These notes closely resemble an introduction to UNIX course maintained by Luis Balona at the South African Astronomical Observatory. This is no accident and I would like to thank Luis for giving me permission to use his notes as a base for constructing this set.

1.2 Introduction

The UNIX operating system started off in 1969 in an attempt to design a portable operating system. In other words, the idea was to have UNIX running on any type of machine. Up to that time, each brand of computer such as IBM, DEC, etc. used its own particular software. If you switched machines, you had to re-learn all the commands. The idea was that with UNIX all this would disappear and it didn't matter what machine was being used - the commands would be identical.

At that time there was no computer screen. Instead, you entered commands on a device similar to a typewriter which printed on paper roll. This was called a "terminal". The concept of a terminal is still used, but usually in a different context. By the 1980's, video screens started replacing terminals. At first, the video screen acted just like a terminal in the sense that only pure text could be displayed. For most purposes this is quite adequate.

However, the visual impact of graphics is so high that very soon software was developed which could display graphics on the screen. For example, Microsoft developed its `windows` operating system to replace the purely text-based `DOS`. In the same way, the developers of UNIX created additional software which could be used to display graphics. This is called the `X-windows` system. Today, most users of UNIX use the `X-windows` rather than basic UNIX. Note that `X-windows` is part of UNIX (it doesn't replace UNIX). As such it can run on any machine which supports UNIX

provided the hardware (graphics screen) is available.

Your instructor for the Operating Systems course has access to a unix box on his desk:

`hughm.cs.ukzn.ac.za`	A 10 year old pentium running RedHat 10.0 used by Hugh Murrell as a UNIX work station

The web-site for this course can be found at:

`hughm.cs.ukzn.ac.za/~murrellh`	follow the os link to the operating systems course

Students taking the Operating Systems course will be given UNIX accounts on the following UNIX machine

`mars.cs.ukzn.ac.za`	running linux

Once you have been given an account you will be able to log in to `mars` from the `senior-lab` by running `ssh` or `putty`. For example, to log in to my account on mars I would type:

```
ssh hughm@mars.cs.ukzn.ac.za
```

and then supply my password to log in and use the UNIX command `exit` to log out. Note that you will have to login to your lab machine in the usual way before you can run `ssh` or `putty`. We will be studying some basic UNIX commands and we assume that you have logged in to your UNIX machine and have the UNIX prompt on your screen.

1.2.1 Commands

Associated with commands in UNIX is a *manual page*. This is a document which tells you in full detail what that particular command does. For example, the command which lists the files in a directory is called `ls`. To find out what it does and how it works, you can look at the manual page on your screen by typing:

```
man ls
```

The command `man` picks out the page of the manual which deals with the command.

Of course, this is all very well, but `ls` isn't a very obvious command and you would be very lucky to guess that this could be the command to list the file names in your directory. Fortunately, there is a way in which you might be able to find out the name of the command. This is the `apropos` command. For example, if you are looking for a command which lists files, you could type in:

```
apropos files
```

You will get the following among a long list:

```
...
lorder (1)  - Finds the best order for member files in an object library
lpr (1)     - Sends files to spooling daemon for printing
ls (1)      - Lists and generates statistics for files
m4 (1)      - Preprocesses files, expanding macro definitions
make (1)    - Maintains up-to-date versions of target files and performs
          shell commands
makedepend (1X) - create dependencies in makefiles
...
```

Among this lot is the command you are looking for and a short description of what it does.

The manual page of `ls` is typical of the manual page for all UNIX commands. You will find associated with `ls` a set of options which are used to modify the behaviour of the command. For example, `ls` by itself simply lists the files in the current directory (try it). However, you may want more information about these files, for example when they were created or how large they are. This is done by adding the option `-l` to the command:

```
ls -l
```

There are many other options associated with `ls` which act in the same way. Look at the manual page and try them out.

1.2.2 Some common commands

Here is a list of commands which are most often used. I suggest you look at the manual pages of each of these commands for more details and experiment with some of the various options. The term "standard output" refers to the screen. Later on you will see how standard output can be re-directed to a file or device.

cal - Displays a calendar

Typing `cal` without any arguments displays a calendar for the current month. Typing `cal 1996` displays a calendar for the year 1996. `cal 6 1996` displays a calendar for June 1996.

cat - Concatenates or displays files

Typing cat foo displays the contents of a file `foo` on the screen. It takes no notice of the fact that you can only see one screen at a time. A better way of displaying a file is to use the `more` command. The value of `cat` is its ability to concatenate files. For example:

```
cat foo1 foo2 foo3 > foo123
```

produces file `foo123` which consists of file `foo1` follwed by `foo2` and `foo3`.

clear - Clears terminal screen

That's all it does!

cd - Changes the current working directory

The command `cd foo` changes the current directory to `foo`. `cd ..` takes you to the directory above the current working directory. `cd` without arguments takes you to your home directory.

cmp - Compares two files

`cmp foo1 foo2` prints nothing if file `foo1` is identical to file `foo2`. If there are differences, it prints the byte and line number where the differences occur.

compress, uncompress - Compresses and expands data

The command `compress foo` compresses the file `foo` to file `foo.Z` which is smaller in size. The original file, `foo` is deleted. To restore `foo`, use `uncompress foo.Z`.

cp - Copies files

`cp foo1 foo2` creates a new file `foo2` which is an identical copy of `foo1`. If `foo2` already exists, it is lost. The normal usage is `cp foo1path/foo1 .` which takes file `foo1`, residing in another directory `foo1path`, and creates a file of the same name in the current directory. Another use is to copy whole directory trees to another location. For example, `cp -r foopath .` copies everything in `foopath`, including its subdirectories and their contents, to the current directory.

du - Displays a summary of disk usage

This command lists the number of blocks of disk usage in the current directory and all subdirectories.

echo - Writes its arguments to standard output

For example `echo "Hello World!"` will display the message `Hello World!`. What appears to be a rather useless command at first turns out to have a great many uses, as you will find out.

find - Finds files matching an expression

This command searches for a file matching a name or part of a name in the current directory and its subdirectories. It has a rather obscure syntax. For example, to search for files in directory `/user2/vk` which have the characters "foo" somewhere in there names, type:

```
find /user2/vk -name "*foo*" -print
```

Files with names such as `foo1, newfoo, old.foo` etc will all be listed.

grep - Searches a file for patterns

This is one of the most useful commands. It allows you to look for a string of characters inside files. For example, suppose you have forgotten the name of the file in which you were writing a letter to Karen. Suppose you remember that in the file somewhere you had mentioned Karen by name. Then you may type:

```
grep -i Karen *
```

which will search for all files in the current directory looking for the name `Karen` inside them. The `-i` flag instructs `grep` to ignore capitalization, so that it will find names such as `Karen, KAREN, karen` etc.

kill - Sends a signal to a running process

`kill 999` stops and deletes the process with process number 999. Use this command to stop a job running in the background, but to use it you need to know its process number. For more details see `ps`.

ln - Link to a file

UNIX has a rather peculiar feature - the ability to link to a file. Essentially, this means that you can have one copy of a file sitting in a certain directory but you can have what amounts to a copy of the same file in another directory. The difference is that this copy takes up almost no disk space if it is a `link`. For example, you may have a very large file, `foo` sitting in directory `foo1dir`, but you need to use it another directory `otherdir`. You could, of course, use the `cp` command to make a duplicate of `foo` in `otherdir`, but this will use the same amount of disk space as the original file `foo`. To conserve disk space, you might decide to make a link to `foodir/foo` in directory `otherdir` using:

```
ln -s foodir/foo foo
```

which creates what appears to be file `foo` in `otherdir`. However, it is not a file, but a link. But it behaves just as if it were the original file `foodir/foo`. Deleting the link using `rm` does not delete the original file in `foodir/foo`.

lpr - Sends files to spooling daemon for printing

The standard UNIX print command is `lpr`. However, in our lab the user must download print files to their local Microsoft machine and print from there. That way all accounting is kept in place.

ls - Lists and generates statistics for files

One of the most common commands. It just displays a listing of all files in the current direcory. Some of the more useful options are as follows:

`ls -l` - Gives full information for each file.

`ls -1` - Lists files, one per line.

`ls -F` - Appends a slash if the file is a directory.

`ls -a` - Lists "invisible" files as well (files begining with a dot).

The options may be concatenated, e.g. `ls -laF`. It is annoying that there is no option for listing only directories. One way of doing this is the command `ls -F | grep /`. This uses two commands linked by a `pipe` (the symbol |). You will learn about pipes later.

mkdir - Makes a directory

The same as the familiar DOS command. `mkdir foo` creates subdirectory `foo` in the current directory.

more - Displays a file one screenful at a time

The standard way of displaying the contents of files on the screen, one screen at a time, e.g. `more foo`. Press the spacebar to view another screen or type `q` to quit.

mv - Moves files and directories

This command is useful to rename files or directories. For example, `mv foo1 foo2` renames file or directory `foo1` to `foo2`. Too bad if `foo2` already exists; it is deleted without warning.

passwd - Changes password file finformation

Use `passwd` to change your password. It will ask you for the new password and for a confirmation.

ps - Displays current process status

This command is normally used to get the process number of some job you want to kill. For example, suppose you are logged in twice and you want to be only logged in at the current terminal. Suppose your user name is `vk`. First, you find all the processes associated with `vk` using:

```
ps -a | grep vk
```

The number in the first column is the process number, say it is 999. Next you kill the process using `kill 999`. There are many other uses for it, but this is one of the

more common ones. Of course, you need to be sure that the process number is the correct one or you may find you have killed some other process instead of your login shell. Fortunately, you can only kill processes that belong to you (unless you are the superuser).

pwd - Displays the pathname of the current working directory

Tels you in what directory you happen to be in, but gives the full path name.

rm - Removes (unlinks) files and directories

Very useful, but dangerous command. The command `rm foo` deletes the file `foo`; `rm *` deletes all files in the current directory, but not subdirectories. You should use `rm -i *` if you want to be prompted before each file is deleted. `rm -r foo` deletes the directory `foo` and all its contents (including subdirectories).

tail - Writes last few lines of a file to standard output

`tail foo` displays the last 10 lines of file `foo` on the screen.

tar - Manipulates tape archives

This command was originally developed to spool data on or off mag tapes, but has other uses which have nothing to do with tapes. To use this command to read data from a tape, the tape must have been written with tar (perhaps on another machine).

The usage for reading a tape is `tar xvf tape.device.name`. The tape device name is usually something like `/dev/rmt1h`, but can be omitted if it is the default name `/dev/rmt0h`. In what follows I will assume the tape device is the standard one and omitt its name

To write the whole of the current directory to tape, use `tar cv .`. To write subdirectory `foo`, use `tar cv foo`.

To list the contents of a tape archive (does not write to disk), usr `tar t`.

A very common, non-tape usage is to bundle-up a whole directory tree into a single file. You may want to do this when copying a directory tree to another machine, for example. The command is: `tar cvf foo.tar foo` where `foo` is the directory and `foo.tar` is the name of the file to contained the bundled-up directory. Often this is `compress`ed to give `foo.tar.Z`. After transfer to another machine, it may be un-bundeled using `tar xvf foo.tar` (you need to uncompress it first if it was

originally compressed). This command creates the original directory tree on the new machine.

w - Prints a summary of current system activity

Use this command to find out who is logged on.

1.2.3 The `vi` editor

One of the most common things you will want to do is to create files and to modify them. You do this by using an *editor*. There are many kinds of editors. In the early days of hardcopy terminals, it was important to minimize time spent in printing to paper. The editor would do this by operating on one line at a time - a line editor. Most of the common UNIX editors started in this way, but gradually changed to accomodate screen editing.

There are two standard UNIX editors – `vi` and `emacs`. People who use the one violently condemn the other. Editors are a matter of personal taste. Most people use neither `vi` or `emacs`, though both are available on the system and you can read how to use them with the aid of the respective manual pages. We suggest that you use `vi`, the standard in UNIX editing. At the end of this document, in appendix A, you will find a short introduction to `vi` editing. Read it and try editing some text files and save them in your home directory on your assigned UNIX box.

1.2.4 Filename shorthand

Suppose you're typing a large document like a book. You might have separate files for each chapter, called `ch1, ch2,` etc. Or, if each chapter were broken into sections, you might have files called `ch1.1, ch1.2,` etc. What if you wanted to print the whole book? You could say `lpr ch1.1, lpr ch1.2 ...` but this would soon get rather boring. This is where filename shorthand comes in. If you say:

```
cat ch*
```

the asterisk, `*`, is taken to mean "any string of characters", so `ch*` is a pattern that matches all filenames that begin with `ch`. The above command will print to the screen all the chapers of the book in alphabetical order. The `*` can be anywhere and can occur several times. Thus

```
rm *.save
```

removes all files that end in `.save`. The strings `ch*` and `*.save` are examples of *regular expressions*. Many unix command allow their parameters to be specified as regular expressions. To find out more about regular expressions read appendix B at the end of these notes.

1.2.5 Input-output redirection

Most of the commands we have seen so far produce output on the terminal screen. The terminal screen is called *standard output*. In the same way, the keyboard is called *standard input*. There is also *standard error*. This is the place where system error messages are displayed. Normally, this is also the screen so that standard output and standard error are the same (it would be most unfortunate if error messages disappeared into some obscure file).

It is a feature of UNIX that standard input, output or error can nearly always be replaced by a file. For example, `ls` normally lists the files in your current directory on the screen. If you prefer this listing to be a file called `foo`, you would type:

```
ls > foo
```

Now, `foo` will contain the file listing that would normally go to the screen. The symbol > means "redirect the output to the file that follows". The file `foo` will be created if it doesn't exit or the previous contents overwritten if it does. No output appears on your screen.

The symbol >> operates in the same way as >, except that it stands for "add to the end of the following file". For example,

```
cat foo1 foo2 foo3 >> temp
```

will append files `foo1, foo2, foo3`, in that order, to the end of `temp`. If `temp` doesn't exist, it will be created.

Some commands produce output on standard error, even when they work properly. For example, `find` will do this if a directory you are searching is protected. It may be desireable in this case to re-direct standard error to a file. The command:

```
find /usr -name foo -print 2> foo.error
```

will direct all error messages from this command to the file `foo.error`. The *file descriptor* number for standard error is 2. Sometimes you want both standard output and standard error directed to the same file. Here is how you do it:

```
find /usr -name foo -print 1> foo.outerror 2>&1
```

This doen't make all that much sense, but it works because the file descriptor number for standard output is 1. The file descriptor number for standard input is 0.

In a similar way, the symbol < alows you to redirect input which would normally be from the keyboard to a file. For example, if file `foo` contains a list of users produced by the command `w > foo`, you could use:

```
grep vk < foo
```

to display information for user `vk` only.

It is also possible to include standard input for a command along with the command itself. In other words, if you have a program which requires some keyboard input, you can include it as in this example:

```
grep "$*" <<Junk
search pattern 1
search pattern 3
search pattern 3
Junk
```

What is happening here is that the input is on three separate lines, `search pattern 1, search pattern 2, search pattern 3`. The block of input is demarcated by the word `Junk` at the begining and at the end. This can be any unique character or string of characters – it doesn't have to be `Junk`. The symbol `"$*"` tells `grep` to look for standard input within the lines demarcated by `Junk`. This might seem very complicated and rather useless, but it can be very useful indeed. This construction is called a *here document*. It means that standard input is right here instead of the keyboard.

1.2.6 Pipes

In the last section we have shown how the output from one command (`w > foo`) can be used as input to another command (`grep vk < foo`) using the intermediate file `foo`. You can do away with such intermediate files by using a *pipe*. In this example, you could more conveniently type:

```
w | grep vk
```

to get the same result. The symbol | is called a pipe and is used to connect the standard output of one command to the standard input of another command. Any number of commands may be connected by pipes.

1.2.7 Processes

When you log in, a program know as a *shell* takes over. When you type in commands, you are communicating with the shell. The shell is an interface between you and the fundamental system which does all the work called the *kernel*. As far as you are concerned, it is the shell that is doing the work, but in reality all the shell does is translate your commands to something the kernel can understand. There are many types of shell, all doing the same thing but in slightly different ways. We will talk about this later.

One of the things the shell can do is to run more than one program at a time. For example, suppose you have a huge, time consuming program called `foo`. You could run it and wait for several munites for it to complete. This is inconvenient because you won't be able to use that terminal screen until the program has finished. It is better to set this big program running in the background. This allows you to continue working with the computer. When the program finishes, the shell will alert you by printing an appropriate message on the screen.

As an example, suppose you want to locate a certain file `foo` on the disk. There may be thousands of files, so that the `find` command may take a long time before finishing. You may set it running in the background as follows:

```
find / -name foo -print > foo.list &
```

The ampersand sign, `&`, at the end of a command line says to the shell "start this command running, then take further commands from the terminal immediately", that is, don't wait for it to complete. The command will begin, but you can do something else while it is running. Directing the output to file `foo.list` keeps it from interfering with whatever you are doing at the same time.

Notice that as soon as you start this command, the shell responds with:

```
[1] 8413
```

or some such number. The number `8413` is the *process-id* of the job. If the program running in the background is out of control, you can kill it using

```
kill 8413
```

If you don't know the process number, you have to look for it using the `ps` command.

It is important to distinguish between programs and processes. `find` is a program; each time you run `find`, or any program, it creates a new process. There is an unique process-id number associated with each running process. The shell is itself a process. When you log in, you set the shell program running. When you logout, you kill the shell process. You can log yourself out by using `kill` if you know what the process id of the shell is (run `ps` to find out).

When you logout, all running processes are killed. This includes any processes that you might have set running in the background using `&`. Sometimes the program takes so long to run to completion that you may want it running even after you have logged out. The command `nohup` was created to deal with this situation. If you say

`nohup` *command* `&`

the command will continue to run if you log out. Any output from the command will be saved in a file called `nohup.out`.

Sometimes you start a program running normally, but decide that it was a bad idea and it would have been better to run it in the background. This can be done without re-starting the program by typing `Ctrl-Z`. In other words, hold down the `Ctrl` key on the keyboard and type `Z`. This temporarily halts the program. Then you continue execution of the program in the background by typing:

```
bg
```

Now you can do other things. To return the background process to the foreground, type

```
fg
```

at any time.

1.2.8 The environment

One of the virtues of the UNIX system is that there are several ways to bring it closer to your personal taste. This is done by "tailoring the environment". The method for tailoring your environment depends on what "shell" you are running.

Unix provides for many diferent shells. Here are the three most common shells:

Shell	Name	Executed at login
`bash`	GNU Bourne Shell	.profile
`csh`	C Shell	.cshrc and .login
`tcsh`	Tc Shell	.cshrc and .login

On `mars` users will run `bash` by default. `bash` is the GNU version of the Bourne shell `sh` based and it has more powerful features. To see all the features available consult the `man` pages.

When you log on, the `bash` reads a file called `profile` stored in a special directory (`/etc`) which can only be modified by the superuser. Then it reads a file called `.profile` in your home directory. This is one of the "invisible" files (because it starts with a dot). You are not allowed to change `/etc/profile`, which sets up the basic environment for all users, but you may change `.profile` to suit your own taste.

One of the most common environment variables is `PATH`. This specifies the path that the shell will take to look for programs or commands. In general, the `PATH` environment variable in `/etc/profile` is set up so that the shell first looks in the current directory for the program or command. If it cannot find it in the current directory, it will look in `/bin`, then in `/usr/bin`, then in `/usr/local/bin`. This is an example of a path. You can display the path by typing:

```
echo $PATH
```

If you want the shell to search for some particular place in your directory for a program, you can add to the search path by including these lines in your `.profile` file:

```
PATH=$PATH":/user1/vk/foo"
export PATH
```

where `/user1/vk/foo` is the full pathname of the directory `foo` where the command is stored. The command `export` ensures that processes spawned by the current process have the new `PATH` environment.

You may want to look at all the environmental variables that have been set. Do this using: `env`

1.3 The UNIX file system

1.3.1 The file system

Everything in the UNIX system is a file. A file is just a sequence of bytes (a byte is a unit of data 8 bits long – think of it as a character). This is true not only of disk files, but also of tape files, line printer files, etc. Because most of the UNIX commands deal with files, a good understanding of the file structure is important.

When a UNIX system is created on an empty disk, two directories are created. These are called `/`, the *root directory*, and `/usr`, the *user directory*. All directories, including `/usr` can be thought of as subdirectories of `/`.

The machine expects to see the *kernel* as a file in the root directory. (If you remember, the kernel is a program which executes commands passed to it by the shell.) This file is called `vmunix` on most machines. When the machine boots (i.e. starts up), it loads the kernel into memory.

There are several directories which are of vital importance and which are present on all UNIX machines.

`/bin`: This subdirectory of `/` contains most of the unix commands, such as `ls, grep, more,` etc. (Remember, a command is just a program which you run.)

`/etc`: This subdirectory of `/` contains important files (not programs) needed by the system. This includes files such as `passwd`, (the username and password file), `hosts` (a file listing the addresses of all machines on the network), `printcap` (a file listing the characteristics of printers connected to the system), `motd`, "message of the day" – the file containing the text which is displayed when a user first logs in, `profile`, which sets up the environment for all users, etc.

`/dev`: A directory containing "device drivers" for various peripherals (disks, tapes, etc.). Rather than having special system routines to, for example read a magnetic tape, there is a file called `/dev/rmt0h`. Inside the kernel, references to the file are converted into hardware commands to access the tape.

`/tmp`: A directory used for temporary storage. When you edit a file, for example, the editor saves a copy of the original file in this directory which has write and read permission set for all users. When the editor has finished its job, it deletes this temporary file.

`/usr/bin`: This subdirectory of `/usr` contains other important unix commands. In some systems, such as on ours, it is this directory which actually contains all the UNIX commands: `/bin` is just a link to `/usr/bin`.

`/usr/man`: This subdirectory of `/usr` contains the manual pages in eight subdirectories: `man1, man2, ....`

We will discuss some of these directories and files in later chapters. In small systems, individual users are allocated space in the `/usr` directory. In other words, the home

directory of `vk`, for example, will be a subdirectory of `/usr` and will have the path `/usr/vk`. On our system, we have separate disks set aside to hold private user directories. In this case, a separate file system is created for each disk (`/user1` and `/user2` on our machine). This filesystem is then *mounted* as a subdirectory of /, the root directory. This is done by creating empty subdirectories `user1` and `user2` in `/` and then using these as *mount points* for the two disks. Each disk, then, appears as a subdirectory of root when it is mounted. You do the same thing when you mount a magneto-optical disk or cdrom on our system.

A single disk can, in fact, be subdivided so that to the system it appears as different disks. This is called *partitioning*. Our system disk is divided into four partitions: `/, /usr, /proc` and `/catalogs` which all share the same physical disk. On booting, `/usr, /proc` and `/catalogs` and other file systems are mounted on root. `/proc` is used by the system as memory swapping storage; `/catalogs` is a file system I created to accommodate astronomical catalogues. The kernel reads the file `/etc/fstab` to determine what filesystems to mount. The command `df` lists the filesystems and their mount points.

1.3.2 Displaying the contents of files

The `od` command allows any file to be displayed as a sequence of bytes ("a bag of bytes"). Normally, many files cannot be displayed on the screen because they are binary files. `od` converts each byte to a printable character. To display the file `foo`, use the command:

`od -c foo` or `od -cb foo`

The former displays the contents as characters, the latter as octal numbers. As an exercise, you should display the contents of directory names, links, etc. Note that each one of these is just "a bag of bytes" and nothing more.

1.3.3 Permission

Every file has a set of *permissions* associated with it, which determine who can do what with the file. If you keep your love letters in a directory, you probably do not want anybody to read them. To do this you need to change the read permission of the directory. But let me warn you that the superuser can still read them!

When you login, you type your username and password. The system actually recognises you by a number, called the user id or *uid*. Besides the uid, you are assigned a group identification or *gid* which places you in a class of users. On our system, all users have been assigned the same gid (which is represented by a number but given the name *users*), but of course the uid's are different. It may be desirable in

some large systems which have, say physicists and astronomers, to have a gid for physicists (say *phys*) and a different one for astronomers (say *astros*).

The file `/etc/passwd` is the *password file*; it contains all the login information about each user. You can discover your uid and gid, as does the system, by looking at this file:

```
aavdw:4OeLd.nWLZV6g:24:15:Audrey van der Wielen:/user2/aavdw:/bin/ksh
vk:mKaffnwrg8azY:30:15:Veronique K:/user2/vk:/bin/ksh
```

The fields are separated by colons and laid out like this:

login-id:encrypted-password:uid:gid:your-name:login-directory:shell

Note that the password appears here in encrypted form. Anyone can read it, but it is impossible to decode it, so it is useless to try. On our system all users have the same gid = 15 (*users*).

When you try to read, write or execute a file, the kernel looks at the permissions set for the file and decides on this basis whether you are allowed to do any of these three these operations. As file owner, you have one set of read, write and execute permissions. Your group has a separate set. Everyone else ("world") has a third set.

When you use the command `ls -l` to display your files, the output looks something like this:

```
-rw-r--r--   1 lab      users        184 Nov 20 11:09 foo
-rw-r--r--   1 lab      users         33 Nov 18 11:53 foo.list
-rwxr-xr-x   1 lab      users       3086 Nov 20 08:47 foox
```

These lines tell us that all three files are owned by uid `lab`, gid `users`. The first file, `foo`, is 184 bytes long, was created on November 20 at 11:09 and has only one link – the number just before the uid.

The string `-rw-r--r--` is how `ls` represents permissions on the file. The very first `-` indicates that it is an ordinary file. If it were a directory, there would be a `d` here. The next three characters encode the owner's (based on the uid `lab`) read, write and execute permissions. The next three characters encode the group permissions. Finally, the last three characters encode permissions for everyone else. In this example, `lab` can read and write to `foo`, but group and world can only read. This is fine - you don't want some other user to overwrite your file. You could do this by changing the permissions of `foo` to `-rw-rw-rw-`. Anyone can read the contents of all three files. Also, everyone can read and execute `foox`.

Anyone can execute the `passwd` command to change her password. This modifies

the password file `/etc/passwd`. Don't get confused by these two files with identical names – one is a command `/bin/passwd`, the other is a file `/etc/passwd`. Since the file `/etc/passwd` can only be modified by the superuser, you may wonder how anyone can change his password. If you do a `ls -l /bin/passwd` you get:

```
-rws--x--x   3 root     bin        16384 May 20  1996 /bin/passwd
```

This tells us that only the owner, `root`, has permission to write to the file; everyone else can only execute the file. But note the execute permission character for the owner, which is supposed to be `x`, has been replaced by `s`. This tells the system that when `/bin/password` is executed by anyone, it must behave as if that person where `root`. In other words, the uid is set to `root` on execution. This is called making the command "set-uid". Since on execution, `/bin/passwd` has `root` privileges, it can modify the `/etc/passwd` file.

Directory permissions behave a little differently, but the basic idea is the same. Here is the output of `ls -ld /usr/bin`:

```
drwxr-xr-x 5 root system 7168 Oct 18 10:44 /usr/bin
```

An `r` field means that you can read the directory, so that you can find out what files it contains using `ls`. A `w` means you can create or delete files in this directory, because that requires modifying and therefore writing the directory file. The `x` field does not mean execute in this case. It stands for "search". If a field is set to `--x`, you can no longer use `ls` to list the contents of the directory or read any files, but you can access any file that you know is there. Similarly, with `r--` users can see (`ls`) the files, but not use the contents of a directory.

1.3.4 Setting permissions

The `chmod` (change mode) command changes permissions on files. The syntax of this command is very clumsy. The most often used options are:

```
chmod +x foo
```

which allows everyone to execute the file `foo`, and:

```
chmod -w foo
```

which turns off write permission to everyone, including the owner of the file. Apart from the superuser, only the owner can change permissions on a file.

These options are fine for simple permission changes, but get very complex when several permission fields have to be changed at the same time. In this case it is much easier applying another option of `chmod` – using numerical values. To use these you need to know something about `octal notation`. Each of the three permission fields, `rwx`, can be set or unset if there is a one or zero in that particular bit position. For example `001 010 100` stands for `--x-w-r--`; `011 110 101` stands for `-wxrw-r-x`, etc.

Now, a triplet such as `011` can be converted to a single octal number. To do this, you must multiply each bit by the numbers 4, 2, and 1 respectively and add up the result. For example:

$110 = 1 \times 4 + 1 \times 2 + 0 \times 1 = 6;$
$011 = 0 \times 4 + 1 \times 2 + 1 \times 1 = 3.$

The octal number that represents `-wx rw- r-x` or `011 110 101` is 365; `--x -w- r--` is represented by `124` etc. The command to change these permissions is therefore `chmod 365 foo` and `chmod 124 foo` respectively.

1.3.5 Running sequences of commands

Sometimes it is useful to be able to run a file containing a sequence of commands. For example, if you need to run these commands very frequently, it is easier if these commands are stored in a file. To run the sequence all you need to do is type in the name of the file. However, before this can be done, the file must be made *executable* using `chmod +x`.

1.3.6 Changing owners

Only the superuser can change the ownership of a file. This is done using:

```
chown vk foo
```

which changes the owner of file (or directory) `foo` to user `vk`. The wild card symbol may be used: `chown vk *`. All files and subdirectories in directory tree `foodir` may be changed using `chown -R vk foodir`.

1.3.7 Inodes

A file has several components: a name, contents and administrative information such as permissions and modification times. The admin information is stored in the inode (should be called "i-node", but the hyphen has fallen away).

It is important to understand the concept of inodes because in a sense, inodes *are* the files. All the directory hierarchy does is provide convenient names for files. Files are recognised by the computer not by their names, but by their inode numbers (i-numbers). You can obtain the i-number of `foo` using `ls -i foo`. If `foodir` is a subdirectory, you can display the i-numbers of all files in the directory using `ls -i foodir`. A directory name is nothing more than a file giving the i-number for each file belonging to it.

A link is just a filename which has the same i-number as another file.

1.4 Software maintenance with the make utility

Compiling your source code files can be tedious, specially when you want to include several source files and have to type the compiling command every time you want to do it. Well, I have news for you... Your days of command line compiling are (mostly) over, because YOU will learn how to write Makefiles. Makefiles are special format files that together with the make utility will help you to automagically build and manage your projects.

For this session you will need the following c++ program and header files:

```
maze.cc
nwin.cc
nwin.h
mazeGame.cc
```

I recommend creating a new directory and placing all the files in there.

note: We will use g++ for compiling. You are free to change it to a compiler of your choice.

1.4.1 The make utility

If you run `make`, this program will look for a file named `Makefile` in your directory, and then execute it. If you have several makefiles, then you can execute them with the command: make -f MyMakefile There are several other switches to the make utility. For more info, `man make`.

1.4.2 Build Process

Compiler takes the source files and outputs object files

Linker takes the object files and creates an executable

1.4.3 Compiling by hand

The trivial way to compile the files and obtain executables, is by running the commands:

```
g++ -o maze maze.cc
g++ -c -o nwin.o nwin.cc
g++ -c -o mazeGame.o mazeGame.cc
g++ -o mazeGame mazeGame.o nwin.o -lncurses
```

1.4.4 Using a Makefile

The basic makefile is composed of:

target: dependencies [tab] system command

This syntax applied to our example would look like:

```
all:    maze win mazeG mazeGame

maze:
        g++ -Wall -o maze maze.cc

win:
        g++ -Wall -c -o nwin.o nwin.cc

mazeG:
        g++ -Wall -c -o mazeGame.o mazeGame.cc

mazeGame:
        g++ -Wall -o mazeGame mazeGame.o nwin.o -lncurses
```

To run this makefile on your files, type:

```
make -f Makefile
```

On this first example our target is called all. This is the default target for makefiles. The make utility will execute this target if no other one is specified. We also see that there are no dependencies for target all, so make safely executes the system commands specified. Finally, make compiles the program according to the command line we gave it.

1.4.5 Using dependencies

Sometimes it is useful to use different targets. This is because if you modify a single file in your project, you don't have to recompile everything, only what you modified. Here is our example with dependencies:

```
# make clean gets rid of previous object files and executables

# make will generate maze and mazeGame executables

OBJS = mazeGame.o \
        nwin.o

all:    maze mazeGame

maze:
        g++ -Wall -o maze maze.cc

mazeGame: $(OBJS)
        g++ -Wall -o mazeGame $(OBJS) -lncurses

clean:
        rm -rf $(OBJS) mazeGame maze
```

Now we see that the target all has only dependencies, but no system commands. In order for make to execute correctly, it has to meet all the dependencies of the called target (in this case all). Each of the dependencies are searched through all the targets available and executed if found. In this example we see a target called `clean`. It is useful to have such target if you want to have a fast way to get rid of all the object files and executables and force recompilation and linking at your next make. Also note the use of comments.

As you can see, variables can be very useful sometimes. To use them, just assign a value to a variable before you start to write your targets. After that, you can just use them with the dereference operator `$(VAR)`.

Where to go from here With this brief introduction to Makefiles, you can create some very sophisticated mechanism for compiling your projects. However, this is just a tip of the iceberg.

1.5 Networking and the Internet on UNIX machines

1.5.1 Introduction

When computers first became available, the typical institute consisted of one single large, expensive machine which catered for many users, perhaps hundreds of users at a university campus. This single machine is usually called a *mainframe*. Each user was connected to the mainframe by means of a serial, low speed, line and a terminal (which simply displayed the information in text mode). No connection existed between the mainframe and any other computer.

This is quite adequate for many purposes and was the only practical solution at a time when computers were very expensive items. By the mid 1970's, experiments were made in connecting computers to each other. The benefit of having computers talking to each other is that instead of duplicating valuable and expensive resources on each machine, it makes it possible for these resources to be made available on all machines which are on the network. The connections that were made in those early days involved computers situated in close proximity to each other (in the same building). The connections were accomplished by connecting each computer with coaxial cable. This allows much higher data transfer rates than the serial lines which connect terminals to computers.

Local area networks (LANS) are those networks usually confined to a small geographic area, such as a single building or a college campus. LANs are not necessarily simple in design, however, as they may link many hundreds of systems and service many thousands of users. The development of various standards for networking protocols and media has made possible the proliferation of LANs worldwide for business and educational applications.

1.5.2 Protocols

Network protocols are standards that allow computers to communicate. A typical protocol defines how computers should identify one another on a network, the form that the data should take in transit, and how this information should be processed once it reaches its final destination. Protocols also define procedures for handling lost or damaged transmissions or "packets". IPX, TCP/IP, DECnet, AppleTalk and LAT are examples of network protocols. At SAAO we used DECnet and LAT, but these have now been abandoned and only TCP/IP is used. TCP/IP is also the protocol used by the Internet.

Although each network protocol is different, they all use the physical cabling in the same manner. This common method of accessing the physical network allows multiple protocols to peacefully coexist, and allows the builder of a network to use common hardware for a variety of protocols. This concept is known as "protocol independence", meaning that the physical network doesn't need to concern itself with the protocols being carried.

1.5.3 Ethernet

Ethernet is the most popular LAN technology in use today. Other LAN types include Token Ring, Fiber Distributed Data Interface (FDDI), and LocalTalk. Each has its own advantages and disadvantages. Ethernet strikes a good balance between speed, price and ease of installation. These strong points, combined with wide acceptance into the computer marketplace and the ability to support virtually all popular network protocols, makes Ethernet the perfect networking technology for most computer users today.

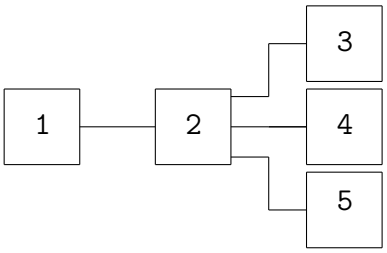
An important part of designing and installing an Ethernet is selecting the appropriate Ethernet medium for the environment at hand. There are four major types of media in use today: ThickWire, thin coax, unshielded twisted pair and fiber optic. Each type has its strong and weak points. Careful selection of the appropriate Ethernet medium can avoid recabling costs as the network grows.

Ethernet media are used in two general configurations or topologies: "bus" and "star". These two topologies define how "nodes" are connected to one another. A node is an active device connected to the network, such as a computer or a piece of networking equipment, for example, a repeater (hub), a bridge or a router.

A bus topology consists of nodes strung together in series with each node connected to a long cable or bus. Many nodes can tap into the bus and begin communication with all other nodes on that cable segment. A break anywhere in the cable will usually cause the entire segment to be inoperable until the break is repaired. The diagram below shows four computers connected together in a bus topology. A cable break between 1 and 2, for example, renders the whole system unworkable until it has been repaired.

Star media links exactly two nodes together. The primary advantage of this type of network is reliability. If a point to point segment has a break, it will only affect the two nodes on that link. Other nodes on the network continue to operate as if that segment were nonexistent. The diagram below shows computer `1` connected to a *hub* `2` which in turn links three other computers `3, 4, 5` in a star topology. A break between the hub and `3` will not affect the connection between the remaining

computers, `1, 4, 5`.



1.5.4 Types of cabling

ThickWire, or 10BASE-5 Ethernet, is generally used to create large "backbones". A network backbone joins many smaller network segments into one large LAN. ThickWire makes an excellent backbone because it can support many nodes in a bus topology and the segment can be quite long. It can be run from workgroup to workgroup where smaller departmental networks can then be attached to the backbone. A ThickWire segment can be up to 500-m long and have as many as 100 nodes attached. ThickWire is a thick, hefty, coaxial cable, and can be expensive and difficult to work with. A thick coaxial cable is used because of its immunity to common levels of electrical noise, helping to ensure the integrity of the network signals.

Thin coax, or 10BASE-2 Ethernet, offers many of the advantages of ThickWire's bus topology with lower cost and easier installation. Thin coax coaxial cable is considerably thinner and more flexible than ThickWire, but it can only support 30 nodes, each at least 0.5-m apart. Each segment must not be longer than 185 m. Subject to these restrictions, thin coax still can be used to create backbones, albeit with fewer nodes. A thin coax segment is actually composed of many lengths of cables, each with a BNC type connector on both ends. Each cable length is connected to the next with a "T" connector wherever a node is needed. Nodes can be connected or disconnected at the "T" connectors as the need arises with no ill effects on the rest of the network. Thin coax's low cost, reconfigurability, and bus topology make it an attractive medium for small networks, for building departmental networks to connect to backbones and for wiring a number of nodes together in the same room, such as a computer lab.

Unshielded twisted pair, or UTP, cable offers many advantages over the ThickWire and thin coax media. Because ThickWire and thin coax are coaxial cables, they are relatively expensive and require some care during installation. UTP is similar to, if not the same as, the telephone cable. A UTP or 10BASE-T Ethernet, uses a star topology. Generally a computer is located at one end of the segment, and the other end is terminated in a central location with a repeater or hub. UTP segments are limited to 100 meters, but UTP's point-to-point nature allows the rest of the network to function correctly if a break occurs in a particular segment.

Fiber Optic, or 10BASE-FL Ethernet, segments are similar to twisted pair. Fiber optic cable is more expensive, but it is invaluable for situations where electronic emissions and environmental hazards are a concern. The most common situation where these conditions threaten a network is in LAN connections between buildings. Lightning strikes can wreak havoc and easily destroy networking equipment. Fiber optic cables effectively insulate networking equipment from these conditions since they do not conduct electricity. Fiber optic cable is used between domes at Sutherland and to connect the electronics workshop and the computer room in Cape Town. The Ethernet standard allows for fiber optic cable segments up to 2km long.

1.5.5 TCP/IP Internet addresses

Information is sent by one computer to another in "packets". Each packet has the address of the machine for which it is destined. Only that machine gets the packet. This is the same as sending a letter with the address written on the envelope. A packet is of fixed size, so the information being sent probably consists of a large number of packets. These packets may or may not arrive in the proper sequence. The machine which is receiving the packets has the responsibility of assembling them in the correct order.

An IP (Internet Protocol) address uniquely identifies a node or host connection to an IP network. System administrators or network designers assign IP addresses to nodes. IP addresses are configured by software; they are not hardware specific. An IP address is a 32 bit binary number usually represented as 4 fields each representing 8 bit numbers in the range 0 to 255 (sometimes called octets) separated by decimal points.

An IP address consists of four decimal numbers (called *octets*) separated by dots. For example, the IP address of `hughm.cs.unp.ac.za` is `143.128.82.130`. Each of the four numbers can be in the range 0 – 255. The IP address consists of two parts: one part identifies the *domain* and the other the *node*. All machines on the same LAN have the same domain address. For example, the domain address for computer science staff is `143.128.82.nnn`. When a new computer is added to our LAN, a new node address `nnn` must be allocated. That means we can have a maximum of 256 computers or network devices attached to the computer science staff LAN.

Obviously, this is not sufficient for the whole of CompSci on the PmB campus. To cater for CompSci we have been allocated a domain *class* which in this case is `143.128.nnn.nnn`. This allows CompSci to have a maximum of $256 \times 256 = 65536$ computers in their network. The class of the address determines which part belongs to the domain address and which part belongs to the node address. Classes can be distinguished by the first number of the IP address. If that number is between:

- 1 and 126 it is a Class A address.
- 128 and 191 it is a Class B address.

- 192 and 223 it is a Class C address.
- 224 and 239 it is a Class D address.
- 240 and 255 it is a Class E address.

127 is reserved for loopback and is used for internal testing on the local machine.

The following list shows which part belongs to the domain (D) and which part belongs to the node (n).

- Class A – DDD.nnn.nnn.nnn
- Class B – DDD.DDD.nnn.nnn
- Class C – DDD.DDD.DDD.nnn

150.215.17.9 is a Class B address so its domain is defined by the first two octets and it's node is defined by the last 2 octets. Class D addresses are reserved for multicasting and Class E addresses are reserved for future use so they should not be used.

It is obvious that Class A networks can accommodate a huge number of nodes, Class B networks a smaller number. Our Class C network can accommodate a maximim of 256 nodes (because there are 8 bits in the node and $2^8 = 256$). Class A and Class B networks might be assigned to large institutes like Universities. There is a central registry in the USA which assigns domain addresses. Of course, no two domain addresses can be the same. Each machine in a domain must be assigned a unique node number. This is the responsibility of the network manager. A list of all the addresses in use at compsci in PmB can be found on the name server `eagle.und.ac.za`

It is not easy to remember these numerical addresses, therefore names can been assigned to each address. For example, our domain name is `cs.unp.ac.za`: the `ac` stands for "academic" and `za` is the country code. The domain name must be registered by a central registry in the USA. In the United States, the domain names do not contain the country code and have the following suffixes:

- com - COMmercial
- edu - EDUcation
- gov - GOVernment
- net - NETwork
- org - ORGanization

Other countries have their own country codes: United Kingdom - `uk`, France - `fr`, etc. The node names can be chosen at will by the network manager. For example,

our machine has been called `mars`, so instead of specifying its numerical address, `143.128.82.4`, it is easier to remember `mars.cs.unp.ac.za`.

Although people use these easy to remember names, machines actually need the numerical address. A computer which can translate an alphabetical address to a numerical address and vice versa is called a *nameserver*.

1.5.6 Gateways and Routers

A machine which connects a LAN to the Internet is called a *gateway*. The gateway machine is responsible for routing packets which are destined for a domain outside the local domain. These machines are called `routers`. Our gateway used to be a PC running a public domain program called PCROUTE. This PC has now been replaced by a commercial CISCO router doing the same job, but much faster.

1.5.7 Telnet

A basic Internet service is the provision of interactive login to a remote host. `Telnet` is both a protocol and a program that enables you to do so. It is the standard TCP/IP remote login protocol.

You must know the address of the remote host computer before you can initiate a session with telnet. Once you know the address, you can use telnet. Most remote hosts require you to have an account to log in (you must have a user id and a password). However, some remote hosts do not require that users have accounts. Users can log in with a general user id such as info (or some other word that is published in guides to the Internet). Passwords are usually not required.

If you are in telnet mode (i.e., the *telnet* > prompt is on the screen and you want to return to the UNIX prompt without initiating an interactive session, type `quit` and press return.

1.5.8 Anonymous ftp

A great deal of useful information is stored in files at computers throughout the country and the world. Many of these file are freely available to users of the Internet. A simple method for transferring such files from a remote computer to a users computer is *anonymous ftp*. Anonymous ftp allows a user to transfer files without having an account at the remote computer (i.e. the user is anonymous.)

To access an anonymous ftp site you must know the address of the site. For example, `ftp.sun.ac.za`, is the address of the Stellenbosch University ftp server. To connect to this server, for example, type:

```
ftp ftp.sun.ac.za
```

You will be asked for your username; type `anonymous`. As a password, type your e-mail address, e.g. `jones@mars.cs.unp.ac.za`. Once you have gained access to the site, the `ftp >` prompt returns and acknowledges that the system is ready to use.

Once you have accessed the ftp site, to transfer a file, you may have to change directories to the directory that your file is located in. Many sites store public access files in a directory called `pub`. To access this directory you would type `cd pub` at the `ftp >` prompt. It is a good idea to list the contents of the directory before attempting to transfer a file. This is done by `ls` at the `ftp >` prompt. When you have determined that the file you want to retrieve is there, you can transfer it to your computer using the `get` command. Before you do this, however, you need to know whether the file you are transferrin is an ASCII file or a binary file. It is nearly always safe to transfer in binary mode, and most ftp sites have this as the default mode. To be on the safe side, you should enforce binary mode transfer. Here is how you would transfer the file `foo`:

```
ftp > binary
200 Type set to I.
ftp > get foo
200 PORT command successful.
150 Opening BINARY mode conection for foo (137 bytes).
226 Transfer complete.
137 bytes received in 2.37 seconds (0.3 Kbytes/s).
ftp >
```

The commands that can be used within `ftp` are the same, or similar, to the UNIX commands which do the same thing. Here is a list of the most commonly used commands:

`ls:` lists the contents of the active directory.

`cd foo:` enables the user to change to directory `foo`.

`cd ..:` allows the user to return to the previous directory.

`lcd foo:` changes to directory `foo` on the *local* machine.

`binary:` chnages to binary mode transfer.

`axcii:` changes to ASCII mode transfer.

`get foo:` transfers file `foo` to the local machine.

`mget *:` transfers all files to the local machine.

`put foo:` transfers file `foo` from the local machine.

`mput *`: transfers all files in the local machine.

When using `mput` or `mget`, you are prompted after each file. To avoid this, you should start your ftp session with the command:

```
ftp -i ftp.sun.ac.za
```

To view a document, `foo`, while still connected to the ftp site, type:

```
ftp > get foo | less
```

Chapter 2

Operating Systems Theory

2.1 Process Synchronization

A *process* is a program whose execution has started but not yet terminated. At any one moment a process need **not** be actually executing.

Three possible states for a process are:

- *Running:* the process is executing on a processor.
- *Ready:* the process is ready to execute but all the processors are in use.
- *Blocked:* the process is waiting for some event to occur before it can continue with its execution.

An operating system must maintain a data structure that describes the current status of all live processes. This structure is called a *process control block* or *pcb* and commonly contains the following fields:

- *Name:* an identifier for the process.
- *State:* running, ready or blocked.
- *Re-Start Info:* program counter, register contents, interrupt masks, etc.
- *Priorities:* for deciding who gets the resources next.
- *Permissions* for access control of resources.
- *Ownerships:* list of resources currently held by process.
- *Accounting:* time used, memory held, I/O volume logged, etc.

During the life of any particular process an operating system must perform operations that can result in a change to the information in the PCB of the process. Typical operations include:

- *Create:* start a new PCB.
- *Delete:* remove a PCB.
- *Signal:* Indicate that a specific event has occured.
- *Wait:* Stop execution until an event is signaled.
- *Schedule:* Assign a runable process to an available processor.
- *Change-Priority:* to influence future resource acquisition rights.
- *Suspend:* either *suspend-ready* or *suspend-blocked* depending on the current state of the process.
- *Resume:* change suspend-ready to ready or suspend-blocked to blocked.

2.1.1 Common synchronization problems

A set of processes is called *determinate* if given the same input, the same results are produced regardless of the order of execution of the processes. Determinate systems are easy to control, the operating system can let them execute in any order and the results are always the same. However in real life sets of processes are usually **not** determinate and the operating system must *synchronize* their execution in order that a prefered result is attained. Some common synchronization problems are:

- *Mutual exclusion problem:* In many computer systems, processes must cooperate with each other when accessing shared data. The designer must ensure that only one process at a time modifies the shared information. During the execution of such critical sections of code mutual exclusion must be ensured.
- *Producer-Consumer problem:* In this problem a set of producer processes provide messages to a set of consumer processes. They all share a common pool of space into which messages may be placed by the producers and removed by the consumers.
- *Reader-Writer problem:* In this problem reader processes access shared data but do not alter it while writer processes change the contents of the shared data. Any number of readers should be allowed to proceed concurrently in the absence of a writer, but writers must insist on mutual exclusion while in their critical section.

It turns out that if one can solve the *mutual exclusion problem* then all the other common sychronization problems are solvable. First we will examine some historical attempts to solve the mutual exclusion problem via software then we will introduce the hardware concept of a semaphore which provides a modern solution.

2.1.2 Mutual exclusion

Consider a system of two cooperating processes P_0 and P_1. Each process has a segment of code, called a *critical section*, in which the process may be reading or writing common variables. The important feature of such a system is that when one process is executing in its critical section the other process must be prevented from executing its critical section. We say that the execution of critical sections must be mutually exclusive in time.

To solve the mutual exclusion problem we must design a protocol which the process must use to cooperate and ensure mutual exclusion. Each process must request permission to enter its critical section by executing socalled *entry code* and after completing its critical section must execute *exit code* so that the next process can enter its critical section. The following constraints must be observed by any practical mutual exclusion solution.

1 Only basic machine language instructions are atomic.

2 No assumptions may be made concerning the relative execution speeds of the cooperating processes.

3 When one process is in a non-critical section of code it may not prevent the other process from entering its critical section.

4 When both processes want to enter their critical section the decision about which one to grant access to cannot be postponed indefinately.

Our first attempt at a solution to the mutual exclusion problem with constraints is to let both processes share a common variable called `turn` initialized to `0` or `1`, and then use the protocol that if `turn` = i then process P_i is allowed to execute in its critical section. Each cooperating process would loop as follows:

```
program P(i)
  common variable: turn: 0..1 = 0;

repeat

  while turn <> i do nothing;

     critical section

  turn = j;

     non-critical section

until false;
```

This solution ensures that only one process at a time can be in its critical section, however, constraint number 3 is not satisfied since strict alternation of processes in the execution of thier critical section is required.

The problem with this attempted solution is that it fails to remember the *state* of each process but remembers only which process is next. To remedy the situation we could use a common flag for each process. The idea is for a process to set its flag before entering its critical section and only do this if the other process's flag is unset. The cooperating processes would loop as follows:

```
program P(i)
  common variables:
    flag: array[0..1] of boolean = {false, false};

repeat

  while flag[j] do nothing;
  flag[i] = true;

     critical section

  flag[i] = false;

     non-critical section

until false;
```

Unfortunately this algorithm does not ensure mutula exclusion. Consider the following sequence of events:

P_0 enters the `while` statement and finds `flag[1]=false`.

P_1 enters the `while` statement and finds `flag[0]=false`.

P_1 sets `flag[1]=true` and enters its critical section.

P_0 sets `flag[0]=true` and enters its critical section.

This sequence of events allows P_0 and P_1 to enter their critical sections at the same time and mutual exclusion is not ensured. The problem is with the non-atomic nature of the entry code. It does not help to interchange the order of the assignment and the `while` loop in the entry code since in that case both processes may exclude each other indefinately and thus violate constraint number 4.

It appears that no simple solution to the mutual exclusion problem exists but a correct solution was discovered in 1964 by the dutch mathematition, *Dekker*. This solution combines both of the previous attempts as follows:

```
program P(i)
  common variables:
    flag: array[0..1] of boolean = {false, false};
    turn: 0..1 = 0;

repeat

  flag[i] = true;
  while flag[j] do
    if turn = j then
      begin
        flag[i] = false;
        while turn = j do nothing;
        flag[i] = true;
      end;

       critical section

  turn = j;
  flag[i] = false;

     non-critical section

until false;
```

We leave it to the reader to convince himself that Dekker's solution ensures mutual exclusion and that indefinite blocking cannot occur.

When more that two processes are involved in the mutual exclusion problem the solution is more complicated. In 1965 another Dutch mathematition, *Dijkstra* solved the n process problem. His solution was refined by *Knuth* and then *DeBruijn*, and finally by *Eisenberg* and *McGuire* in 1972 to produce the following n process solution that satifies not only all the constraints 1 to 4, but is also *fair* in the sense that every process can eventually enter its critical section even if access requirements are greater than total time allows.

```
program P(i)
  common variables:
    flag: array[0..n-1] of (idle, want, in) = {idle, ...};
    turn: 0..n-1 = 0;

 ordinary variables:
    j: integer;

repeat

  repeat
```

```
      flag[i] = want;
      j = turn;
      while j <> i do
        if flag[j] <> idle then
          j = turn
        else
          j = j+1 mod n;
      flag[i] = in;
      j = 0;
      while ( j<n ) and ( j=i or flag[j]<>in ) do inc(j);
    until (j >= n) and ( turn = i or flag[turn] = idle);
    turn = i;

        critical section

    j = turn+1 mod n;
    while (j<>turn) and (flag[j] = idle) do j = j+1 mod n;
    turn = j;
    flag[i] = idle;

      non-critical section

  until false;
```

We leave it to the reader to convince himself that the Eisenberg and McGuire solution ensures mutual exclusion and that indefinite blocking cannot occur.

2.1.3 Semaphores

In modern computer instruction sets, the problem of *entry* and *exit* code for the mutual exclusion is solved by supplying an atomic instruction that does the job. The data structure associated with this instruction is called the *semaphore* and a full definition is as follows:

A semaphore is an integer variable, S, and an associated group of waiting processes, $Q(S)$, upon which only two operations, P and V, may be performed.

$P(S)$ `if` $S \geq 1$ then $S = S - 1$ `else` the executing process places itself in $Q(S)$ and goes to sleep.

$V(S)$ `if` $Q(S)$ is non-empty then wake up one waiting process and make it available for execution `else` $S = S + 1$.

The operating system must offer P and V as indivisible instructions. This means that once they start executing they cannot be interrupted until they have completed. Note also that in the definition of $V(S)$, no rules are laid down to identify

which waiting process is reactivated. In most operating systems this decision is implementation dependent.

To solve the mutual exclusion problem using a semaphore the following scheme can be used:

```
        shared vars:  S: semaphore = 1;

Process i:                          Process j:
loop                                loop
  ...                                 ...
  P(S)                                P(S)
    ...                                 ...
    critical section                    critical section
    ...                                 ...
  V(S)                                V(S)
    ...                                 ...
    non-critical section                non-critical section
    ...                                 ...
endloop                             endloop
```

To see that this scheme will work consider the following two senarios:

1: Process `i` goes in and out of its critical section while processes `j, k, ...` do not attempt entry. That is: $S = 1$; $i : P(S)$, $S = 0$; `i enters`; i:V(S), $S = 1$; `i exits`; and the initial configuration is restored.

2: Prosess `i` goes in, `j` attempts entry and `k, l, ...` are not interested. That is: $S = 1$; $i : P(S)$; $S = 0$; `i enters`; $j : P(S)$; `j waits`; $i : V(S)$; `i exits, j enters`; $j : V(S)$; $S = 1$; `j exits`; and the initial configuration is restored.

2.1.4 Producer/Consumer problem via semaphores

In this problem we have many producers producing messages which are consumed by many consumers. Howeve, there are only a finite number of message buffers:

```
            shared vars: nrfull: semaphore = 0
                         nrempty: semaphore = N
                         mutexP: semaphore = 1
                         mutexC: semaphore = 1
                         buff: array [0..N-1] of message
                         in, out: 0..N-1 = 0

producer i:                          consumer j:
```

```
loop                                  loop
  ...                                   ...
  create a message m

  P(mutexP)                             P(mutexC)
  {one producer at a time}              {one consumer at a time}

    P(nrempty)                            P(nrfull)
    {wait for an empty cell}              {wait for a message}

      buff[in] = m                          m = buff[out];
      in = in + 1 mod N                     out = out + 1 mod N

    V(nrfull)                             V(nrempty)
    {signal a full cell}                  {signal an empty cell}

  V(mutexP)                             V(mutexC)
  {let the next producer in}            {let next consumer in}

  ...                                   consume message
endloop                               endloop
```

2.1.5 Reader/Writer problem via semaphores

In this problem we have a number of *writer* programs that must exclude all readers and other writers when in their critical section. We also have a number of reader routines who can perform their read operations concurrently but writer routines must be excluded. The following semaphore solution gives priority to the readers:

```
            shared vars:  mutexW, mutexR: semaphore = 1
                          nr: integer = 0

 reader i:                                writer j:

 loop                                     loop
   ...                                      ...
   P(mutexR)                                P(mutexW)
   {readers enter one at a time}            {wait}

     if nr=0 then P(mutexW)                   critical section
     {first reader inhibits writers}          for writers
     nr = nr+1
                                            V(mutexW)
                                            {signal}
   V(mutexR)
   {allow other readers in/out}           endloop
```

```
      critical reader section

   P(mutexR)
   {readers exit one at a time}

     nr = nr-1
     if nr=0 then V(mutexW)
     {last out allows writers in}

   V(mutexR)
   {allow other readers in/out}

endloop
```

2.1.6 Exercises

1: Use a semaphore with P and V operations to control the trafic flow at the intersection of two **one-way** streets. The following rules should be satisfied:

- Only one car can be crossing at any given time.
- When cars are approaching the intersection from both directions they should take turns at crossing so as to prevent indefinite postponements in either street.
- A car approaching from one street should always be allowed to cross the intersection if there are no cars approaching from the other street.

A solution to this problem is two algorithms for crossing. One algorithm for cars comming from one direction and another algorithm for cars comming from the other direction.

2: Consider a **barbershop** that has three barber chairs, three barbers, one till, and one cashier. The shop also contains a sofa that can seat four waiting customers, and standing room area for further waiting customers. Assume that at most twenty customers can be inside the barbershop at any one time.

A customer enters the shop, provided it is not full and once inside takes a seat on the sofa or stands if the sofa is fully occupied. When a barber is free the customer who has been on the sofa for the longest is served and if there are any standing customers the one who has been in the shop the longest takes a seat on the sofa. Whan a customer's haircut is finished the cashier accepts payment and gives the customer his recipt. Because there is only one till payment is accepted for one customer at a time. The barbers divide their time between cutting hair and sleeping in their barber chair if there are no customers to be served.

Solve this concurrency problem by writing three algorithms. One each for customers, barbers and cashiers. Make a table of all the semaphores you use indicating what the P and V operations denote for each.

3: There are five **philosophers** sitting at a round table. On the table are five plates, five forks (one to the left of each plate), and a bottomless serving bowl of spaghetti at the center of the table. The philosophers spend their time either thinking or eating. Thinking is easy as the philosopher does not require any utensils to do it. Eating on the other hand requires **two** forks, one from the left and one from the right. On completion of an eating period the philosopher will replace the two forks in their original positions and return to thinking.

Design an algorithm for a philosopher to follow that allows all philosophers to think and eat to their hearts content.

2.2 InterProcess Communication under UNIX

UNIX now offers new powerful interprocess communication (IPC) facilities. These include message queues, shared memory segments and semaphores. In this section we show how to call UNIX IPC routines to implement shared memory and semaphores. UNIX offers two shell commands to monitor the current IPC state and to delete unwanted IPC structures. The formats are:

```
ipcs [options]
ipcrm [options]
```

Use the man pages to get information on the available options.

To tackle the group project given later in this document, you will have to implement shared memory and semaphores in C++. Make sure you can write simple C++ programs before you start with IPC.

2.2.1 Shared Memory

Shared memory is a section of main computer memory which can be shared by one or more independent processes. Shared memory is attached to the data segment of a process and can appear at a different address in each process. Shared memory accesses by different processes must be synchronized through the use of semaphores.

UNIX provides the system call `shmget` to create a shared memory segment along with its associated data structures. The segment is then attached to a processes address space through the use of the system call, `shmat`. `shmget` returns the shared memory identifier `shmid` which is later used by the system call `shmctl` to update the contents of shared memory. The system call, `shmdt` is used to detach a shared memory segment from a processes address space.

Full specifications for the above system calls can be found in the `man` pages. A sample program, written in C, (the source of which can be downloaded from my web site) shows you how to create and use a shared memory segment.

2.2.2 Semaphores

UNIX supplies a system call, `semget` to set up a semaphore with its associated data structure. `semget` returns a unique positive integer known as the semaphore identifier, `semid`. The `semid` is subsequently used by the `semop` system call which updates the values in the semaphore data structure.

The UNIX semaphore structure contains the variables, `semval`, `semzcnt` and `sempid`. The variable `semval` is a non-negative integer whose value is changed by the `semop`

system call. `semval` corresponds to the semaphore integer described above in these notes. `semzcnt` is an unsigned short integer that represents the number of processes that are suspended waiting for `semval` to reach zero. `sempid` holds the id of the process that performed the last semaphore operation on this semaphore.

Full specifications for `semget` and `semop` can be found in the `man` pages. A sample program, written in C, (the source of which can be downloaded from my web site) shows you how to create and destroy a semaphore and also how to implement the `P` and `V` mutual exclusion operators.

The above sample codes will help you solve IPC problems. To get a nice window like interface on aa ascii terminal, it is advisable to learn the `ncurses` library. This library offers a host of system calls that allows you to obtain keystrokes from the user and update an ascii terminal screen as appropriate to your application. Further documentation on `ncurses` can be found on the web. A sample program, written in C++, (the source of which can be downloaded from my web site) shows you how to implement a simple ncurses program.

2.2.3 Pair Project 2010: two-player maze game:

Consider a two-player *mazeGame*.

Players should try to navigate the maze from start to goal as quickly as possible. The first player to reach the goal wins the game. Invent your own rules for navigating the maze. (such as no two players may be in the same position at the same time). Your program must make sure that the players do not make illegal moves when navigating the maze.

Make use of ncurses, semaphores and shared memory to construct the *mazeGame*. Create one code for each player to execute. When either player makes a move the move must reflect on both player's code. ie: communication must take place. Make sure that you clearly specify all shared memory data with their initial values.

2.3 Deadlock

At the end of the previous section we saw that a simple sychronization algorithm can end up in a *deadlocked* state when more than one processes are competing for a few resources.

2.3.1 A definition for deadlock

A set of processes is in a state of *deadlock* when every process in the set is waiting for a resource that can only be released by another process in the set.

A deadlock situation may arise *iff* the following necessary condition holds:

- *circular hold and wait:* There must exist a set of waiting processes $\{p_1, p_2, \ldots, p_n\}$ such that p_1 is waiting for a resource held by p_2, p_2 is waiting for a resource that is held by p_3, ..., p_n is waiting for a resource that is held by p_1. The resources involved must be held in a non-sharable mode.

2.3.2 Resource Allocation Graphs

Deadlocks can be described more precisely in terms of a directed bipartite graph $G(V, E)$, called a *resource allocation graph*, where the set of vertices V is partitioned into two types, $P = \{p_1, p_2, \ldots, p_n\}$ the set of processes and $R = \{r_1, r_2, \ldots, r_m\}$ the set of resource types.

Each element in the set E of edges is an ordered pair (p_i, r_j) or (r_j, p_i). If $(p_i, r_j) \in E$ then there is a directed edge from process p_i to resource type r_j indicating that process p_i has requested an *instance* of resource type r_j and is currently waiting for that resource. If $(r_j, p_i) \in E$ then there is a directed edge from resource type r_j to process p_i indicating that an instance of resource type r_j has been allocated to process p_i. These edges are called *request* edges and *assignment* edges respectively.

Pictorally we represent each process p_i as a circle and each resource type as a square. Since a resource type r_j may have more than one instance we represent each such instance as a dot within the square. A request edge points to a square while an assignment edge starts at one of the dots and points to a circle.

When a request edge is fulfilled it is *instantaneously* transformed into an assignment edge which is deleted when the resource is released.

Here are some facts about resource allocation graphs:

- If G contains no cycles then no process in the system is deadlocked.
- If G contains a cycle then a deadlock **may** exist.

- If each resource type has exactly one instance then a cycle implies that a deadlock has occurred.
- If any resource involved in a cycle has more than one instance then the cycle does not necessarily imply a deadlock.
- A system is deadlocked *iff* when resources are partitioned according to instances there is no way to draw request edges without introducing a cycle.

2.3.3 Resource allocation examples

2.3.4 Deadlock Prevention

The only way to prevent deadlock from occuring is to ensure that a circular hold and wait condition never occurs. We will investigate three methods for doing this:

Prevention by Preemption

To prevent the *hold and wait* condition we allow implicit preemption. If a process that is holding some resources requests another resource that cannot be immediately allocated to it then all resources currently held are preempted and implicidly released. The preempted resources are added to the list of resources for which the process is waiting. The process will only be restarted when it can regain all its old resources as well as the new one that it requested.

This is not the best deadlock prevention scheme available. For example if a line printer is continually preempted the the operator whould have a terrible time sorting out which printed pages belonged to which process.

Prevention by Linear Ordering of Resources

In order to ensure that the circular *hold and wait* condition never happens one can impose a linear ordering of all resource types. To do this one must define a $1-1$ function F that maps resource types to integer numbers. For example suppose our resource types are *card readers*, *disk drives*, *tape decks* and *line printers* and:

$$F(cr) = 1$$

$$F(dd) = 5$$

$$F(td) = 7$$

$$F(lp) = 12$$

and suppose that we insist that each process can only request resources in increasing order of enumeration. To do this we require that whenever a process requests a resource r_j it first releases any resource r_i such that $F(r_i) > F(r_j)$. If this protocol is followed then a circular hold and wait condition cannot happen.

To see this assume that a circular hold and wait condition exists with $\{p_0, p_1, \ldots, p_{n-1}\}$ involved. Assume that p_i is waiting for resource r_i which is held by p_{i+1}, (mod n on all the indecies). Thus since p_{i+1} is holding r_i while requesting r_{i+1} we must have $F(r_i) < F(r_{i+1}$ for all i. In other words: $F(r_0) < F(r_1) < F(r_2) < \ldots < F(r_{n-1}) < F(r_0)$. This is a contradiction and thus the circular hold and wait cannot exist if the protocol is adhered to.

The Banker's Algorithm

The following deadlock prevention scheme ensures that a circular hold and wait condition cannot occur by demanding that at any time the system is in a *safe* state. More formally, a system of processes is in a *safe* state if there exists a sequence, $\{p_0, p_1, \ldots, p_{n-1}\}$ such that for each p_i the resources which p_i can still request can be satisfied by the available resources plus the resources held by all the p_j with $j < i$.

To check whether or not a collection of processes is in a safe state the operating system must maintain several data structures containing the current state of resource allocations.. Let n be the number of processes and m the number of resource types. The banker's algorithm will require the following data structures:

- *Available:* A vector of length m with $Available[j] = k$ if there are currently k instances of resource type r_j available.
- *Max:* An $n \times m$ matrix defining the maximum demand for each resource type by each process. If $Max[i, j] = k$ then process p_i may request at most k instances of resource type r_j.
- *Allocation:* An $n \times m$ matrix defining the number of resources of each type currently allocated to each process. If $Allocation[i, j] = k$ then process p_i is currently allocated k instances of resource type r_j.
- *Need:* An $n \times m$ matrix indicating the remaining need of each process. If $Need[i, j] = k$ then process p_i may need k more instances of resource type r_j in order to complete its task. Note that $Need[i, j] = Max[i, j] - Allocation[i, j]$.

Given that the above data structures are kept up to date by the operating system, the algorithm for checking whether or not the system is in a safe state is quite simple:

1: Let $Work$ and $Finish$ be vectors of length m and n respectively. Initialize $Work = Available$ and $Finish[i] = False$ for all i.

2: Find an i such that:

   a: $Finish[i] = False$

   b: $Need[i] \leq Work$

   If no such i exists then goto step 4.

3: Set:

   a: $Work = Work + Allocation[i]$

   b: $Finish[i] = True$

   and goto step 2.

4: if $Finish[i] = True$ for all i then state is *safe*.

Now to complete the banker's algorithm we must specify what is to be done when a request for resources comes in from process p_i. Suppose p_i issues a request vector $Request[i]$ with $Request[i, j] = k$ if process p_i wants k more instances of resource type r_j. The operating system must then take the following action:

1: If $Request[i] > Need[i]$ then we have an error since the process has requested more resources than initially allowed for in $Max[i]$.

2: If $Request[i] > Available$ the process p_i must wait.

3: The operating system **pretends** to allocate the resources as follows:

a: $Available = Available - Request[i]$.

b: $Allocation[i] = Allocation[i] + Request[i]$.

c: $Need[i] = Need[i] - Request[i]$

If the resulting resource state is *safe* then the allocations are made and the process p_i continues. If the new state is *unsafe* then p_i must wait and the old state is restored.

2.3.5 Exercise

1) Consider the following resource allocation data for five processes competing for four resource types.

$$Available = \begin{matrix} 1 & 5 & 2 & 0 \end{matrix} \qquad Max = \begin{matrix} 0 & 0 & 1 & 2 \\ 1 & 7 & 5 & 0 \\ 2 & 3 & 5 & 6 \\ 0 & 6 & 5 & 2 \\ 0 & 6 & 5 & 6 \end{matrix}$$

$$Allocation = \begin{matrix} 0 & 0 & 1 & 2 \\ 1 & 0 & 0 & 0 \\ 1 & 3 & 5 & 4 \\ 0 & 6 & 3 & 2 \\ 0 & 0 & 1 & 4 \end{matrix} \qquad Need = \begin{matrix} 0 & 0 & 0 & 0 \\ 0 & 7 & 5 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 0 & 2 & 0 \\ 0 & 6 & 4 & 2 \end{matrix}$$

a) Show that this system in a safe state?

b) If a request, $[0, 4, 2, 0]$, comes in from the second process, will the operating system grant it?

2.4 Scheduling

2.4.1 Introduction

CPU scheduling deals with the problem of deciding which of the processes in the READY queue is to be allocated the CPU. The criteria used for comparing different CPU scheduling algorithms include:

- *Utilization:* The idea is to keep the CPU as busy as possible. In real systems CPU utilization could vary from 40 to 90 percent.
- *Throughput:* One way to measure the work done by the CPU is to count the number of tasks that are completed per unit of time.
- *Turnaround Time:* The interval of time from submission to completion. Includes, waiting time, executing time and I/O time.
- *Waiting Time* Some scheduling algorithms just try to minimize the waiting time rather than the complete turnaround time.
- *Response Time:* The time from submission of a request until the first response is produced. Often used as a criteria in interactive systems.

In most case an *average* measure is minimized.

2.4.2 F.C.F.S. Scheduling

By far the simplest scheduling algorithm is *First-Come-First-Served*. The implementation is easily managed with a *First-In-First-Out* READY queue.

The performance of FCFS is however often quite poor. Consider the following three tasks with known CPU burst times. We can compute the average turnaround time to service these three CPU bursts:

Task	Burst Time
T_1	24
T_2	3
T_3	3

If the tasks arrive in the order 1, 2 and 3 to a FCFS scheduler then the average turnaround time can be computed with the aid of a *Gantt* chart.

T_1	T_2	T_3

and the *Average turn-around time*, or ATT, can be computed as:

$$ATT = \frac{24 + 27 + 30}{3} = 27$$

On the other hand, if the tasks arrive in the reverse order, 3 then 2 then 1, a much better result is obtained.

T_3	T_2	T_1

$$ATT = \frac{3 + 6 + 30}{3} = 13$$

Thus we conclude that the average turnaround time for FCFS scheduling is not in general minimal.

In addition consider the performance of FCFS in a dynamic situation. Assume 1 CPU bound tasks and many I/O bound tasks. The following scenario often results:

- The CPU bound task gets the CPU and holds it. The I/O bound tasks all wait for the CPU in the READY queue. The I/O tasks are IDLE.
- The CPU task finishes and moves to an I/O device. All the I/O tasks finish quickly and now wait in the I/O queue. The CPU is now IDLE.

There is a *convoy effect* as these two situations repeat resulting in plenty of IDLE time. The way around this problem is to allow shorter tasks to go first!

2.4.3 S.T.F. Scheduling

In *Shortest-Task-First* scheduling the CPU is assigned to that task with the smallest next CPU burst time. For example:

Task	Burst Time
T_1	6
T_2	3
T_3	8
T_4	7

T_2	T_1	T_4	T_3

$$ATT = \frac{3 + 9 + 16 + 24}{4} = 13$$

Theorem

STF scheduling is optimal in that it yields the minimum *average waiting time*.

Proof (due to *P. Somaroo*, 1995.)

Let the execution time for task T_i be denoted as t_i then:

$$\begin{aligned} AWT &= \frac{0+t_1+(t_1+t_2)+\cdots+(t_1+t_2+\cdots+t_{n-1})}{n} \\ &= \frac{(n-1)t_1+(n-2)t_2+\cdots+2t_{n-2}+t_{n-1}}{n} \end{aligned}$$

which is minimised if $t_i < t_j$ whenever $i < j$ since t_i has a larger *weight* than t_j. *Q.E.D.*

Although STF is optimal in the shortest waiting time sense, it is *unimplementable*. There is no way that the operating system knows the length of the next CPU burst time.

One approach is to try and approximate STF scheduling. We try to predict the length of the next CPU burst by considering the *burst time history* of the task.

For example let t_n be the length of the n^{th} CPU burst and let τ_n be our predicted value for the n^{th} CPU burst. Then let

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

where α is a parameter satisfying $0 \leq \alpha \leq 1$ which controls the relative weight of recent and past history in our prediction.

If $\alpha = 0$ then $\tau_{n+1} = \tau_n$ and our estimate never changes. If $\alpha = 1$ then $\tau_{n+1} = t_n$ and only the most recent CPU burst time is taken into account. However if $0 < \alpha < 1$ then the following expansion shows how past history is incorporated.

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2\alpha t_{n-2} + \cdots$$

We see that each successive term has less weight than its predecessor. Note that the initial τ_0 can be a system constant.

2.4.4 Priority Scheduling

In this type of scheduling a *priority* is associated with each task and the CPU is allocated to the task with the highest priority. Equal priority tasks are scheduled according to FCFS. Note that STF scheduling is just priority scheduling with the priority set to $\frac{1}{\tau}$.

Priorities can be defined either internally or externally.

Internal priority algorithms will use factors such as, *time limits*, *memory requirements*, *number of open files* and *ratio of I/O to CPU bursts* to compute the priority of a given task.

External priority algorithms will use factors such as, *funds*, *sponsers* and *politics* to allocate priorities.

A major problem with priority scheduling is *starvation* A priority scheduling algorithm can leave some low priority task waiting indefinitely. A solution to this problem is aging. Gradually increase the priority of a task that stays in the system for a long time.

Rumor has it that when they closed down the IBM 7094 at MIT in 1973 they found a low-priority task that had been submitted in 1967.

2.4.5 Preemptive Scheduling

FCFS, STF and PRIORITY scheduling algorithms are **non-preemptive** by nature. Once the CPU has been allocated to a process it stays allocated until the process releases the CPU, (either by terminating or by requesting I/O).

FCFS is intrinsically non-preemptive but the other two can be modified to be preemptive algorithms. For example if a new task arrives in the queue with a shorter expected burst-time (or a higher priority) than the currently executing task, then the currently executing task is preempted and the new task is assigned to the CPU.

Consider the following example:

Task	Arrival Time	Burst Time
T_1	0	8
T_2	1	4
T_3	2	9
T_4	3	5

Using a *preemptive* scheduling algorithm results in:

T_1	T_2	T_4	T_1	T_3

$$ATT = \frac{17 + 4 + 24 + 7}{4} = 13$$

where as for a *non-preemptive* schedule we have the following situation:

T_1	T_2	T_3	T_4

$$ATT = \frac{8 + 11 + 19 + 23}{4} = 15\frac{1}{4}$$

We see that preemption results in an improved turn-around time. However it must be noted that preemption has a roll-out roll-in *overhead* which has not been reflected in the above calculations:

2.4.6 Round Robin Scheduling

A *Round Robin* scheduling algorithm is usually used for time sharing systems. A small unit of time, called a *time-slice* is defined. The READY queue is treated as a circular queue and the CPU scheduler goes around the READY queue allocating the CPU to each process for *one* time-slice.

For example:

Task	Burst Time
1	24
2	3
3	3

A *Round Robin* schedule with a time-slice of 4 is:

T_1	T_2	T_3	T_1	T_1	T_1	T_1	T_1

$$ATT = \frac{30 + 7 + 10}{3} = 15\frac{2}{3}$$

Note that an infinite time-slice is equivalent to FCFS scheduling while a very small time slice is equivalent to each process running on its own processor at $\frac{1}{n}$ the speed of the real processor. Again there are *overheads* with roll-out that have not been taken into account.

2.4.7 Scheduling Tasks on more than one Processor

In this section we investigate algorithms for scheduling tasks on more than one processor. We assume that the tasks are independent and can run in any order we wish. However, once a task starts it must run to completion so we will not be looking at preemptive algorithms in this section. When processing on more than one processor the measure that is usually optimized is the *total throughput time*. No optimal algorithm is known for minimizing total throughput time. We will investigate one heuristic algorithm.

We define the *largest processing time* schedule, or LPT schedule, as the result of an algorithm which, whenever a processor becomes free, assigns that task whose execution time is the largest of those tasks not yet assigned.. For cases when there is a tie, an arbitrary tie-breaking rule can be employed. Consider the following example:

Task	Processing Time
T_1	6.5
T_2	4
T_3	3.5
T_4	3
T_5	2
T_6	1
T_7	1

An LPT schedule for this set of tasks on 3 processors turns out to be an optimal schedule for 3 processors:

	1	2	3	4	5	6	7	8
P_1								
P_2								
P_3								

We see that processor 1 finishes last and the total throughput time for the set of tasks scheduled in this way is:

$$t(LPT) = 7.5$$

The LPT schedule is **not** always an optimal schedule. Consider the following example on 3 processors:

Task	Processing Time
T_1	8
T_2	6.5
T_3	6
T_4	4
T_5	3
T_6	2.5
T_7	2.5
T_8	1

An LPT schedule for this set of tasks has a total throughput of 12.

	1	2	3	4	5	6	7	8	9	10	11	12
P_1												
P_2												
P_3												

$$t(LPT) = 12$$

whereas an optimal schedule, an OPT schedule, has a total throughput time of 11.5.

	1	2	3	4	5	6	7	8	9	10	11	12
P_1												
P_2												
P_3												

$$t(LPT) = 11.5$$

Just how good is LPT scheduling as compared to an optimal schedule? After a certain amount of experimentation about the possible shortcomings of an LPT schedule one usually arrives at the following example for the case of 2 processors:

Task	Processing Time
T_1	3
T_2	3
T_3	2
T_4	2
T_5	2

A 2 processor LPT schedule for this set of tasks has a total throughput of 6.5

	1	2	3	4	5	6	7
P_1							
P_2							

whereas the OPT schedule has a total throughput of 6.

	1	2	3	4	5	6	7
P_1							
P_2							

Constructing LPT and OPT schedules for the following set of tasks gives a *worst case* scenario when 3 processors are involved:

Task	Processing Time
T_1	5
T_2	5
T_3	4
T_4	4
T_5	3
T_6	3
T_7	3

For the worst case scenario on m processors consider $2m + 1$ tasks with $t_i = 2m - Floor(\frac{i+1}{2})$ for $i = 1, 2, \ldots, 2m$ and with $t_{2m+1} = m$. It can be verified by constructing Gannt charts of the LPT and OPT schedules that:

$$\frac{t(LPT)}{t(OPT)} = \frac{4}{3} - \frac{1}{3m}$$

The main result of this subsection is a generalization of the preceeding result:

Theorem

Given any set of independent tasks and m identical processors:

$$\frac{t(LPT)}{t(OPT)} \leq \frac{4}{3} - \frac{1}{3m}$$

Proof

The theorem is trivally true for $m = 1$ since in that case $t(LPT) = t(OPT)$. So let $m \geq 2$.

Assume the theorem is false. Contrary to the theorem assume that we have a **minimal** set of tasks, $\{T_1, T_2, \cdots, T_n\}$, with execution times, $\{t_1, t_2, \cdots, t_n\}$ and assume that the tasks are ordered so that $t_1 \geq t_2 \geq \cdots \geq t_n$. With these assumptions the LPT schedule will allways assign the tasks in numerical order.

Now assume that T_k finishes last in the LPT schedule with $k < n$. then an LPT schedule for the set of tasks $\{T_1, T_2, \cdots, T_k\}$ would complete at the same time as an LPT schedule for the tasks $\{T_1, T_2, \cdots, T_n\}$ and this smaller set of tasks would also invalidate our theorem. But we assumed that our n tasks was a **minimal** set so we have a contradiction and can safely assume that $k = n$ and T_n finishes strictly last in the LPT schedule for $\{T_1, T_2, \cdots, T_n\}$.

We shall now show that any OPT schedule for $\{T_1, T_2, \cdots, T_n\}$ can have at most **two** tasks per processor. First we note that

$$t(OPT) \geq \frac{1}{m} \sum_{i=1}^{n} t_i$$

Now let τ_n denote the **starting** time of T_n in an LPT schedule for $\{T_1, T_2, \cdots, T_n\}$. Since no processor can be idle before T_n begins execution we have:

$$\tau_n \leq \frac{1}{m} \sum_{i=1}^{n-1} t_i$$

and hence:

$$\begin{aligned} \frac{t(LPT)}{t(OPT)} &= \frac{\tau_n + t_n}{t(OPT)} \\ &\leq \frac{t_n}{t(OPT)} + \frac{1}{mt(OPT)} \sum_{i=1}^{n-1} t_i \end{aligned}$$

$$\begin{aligned} &\leq \frac{(m-1)t_n}{mt(OPT)} + \frac{1}{mt(OPT)} \sum_{i=1}^{n} t_i \\ &\leq \frac{(m-1)t_n}{mt(OPT)} + 1 \end{aligned}$$

and since the theorem does not hold for $\{T_1, T_2, \cdots, T_n\}$ we have:

$$\frac{4}{3} - \frac{1}{3m} < \frac{(m-1)t_n}{mt(OPT)} + 1$$

from which we obtain:

$$t(OPT) < 3t_n$$

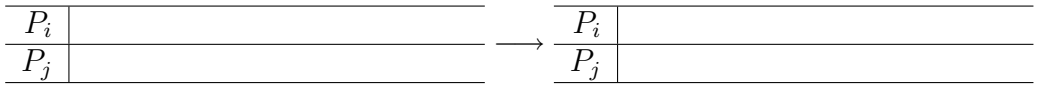
.

Therefore, since T_n has the least execution time we conclude that if the theorem is violated then no processor can execute more than two tasks in an optimal schedule for $\{T_1, T_2, \cdots, T_n\}$

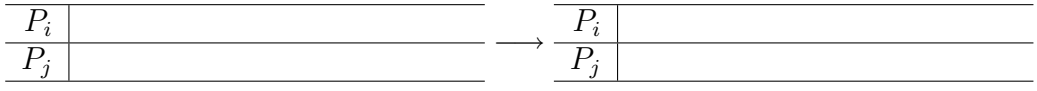
To complete the proof we will show that a two task at most per processor OPT schedule can be transformed into an LPT schedule without increasing the total throughput time which contradicts our assumption that the theorem was invalid.

Consider the following four types of transformations on schedules:

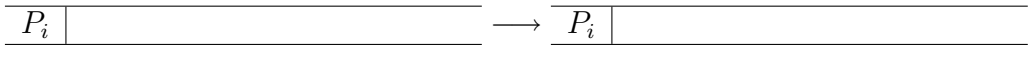
Type A:



Type B:



Type C:



Type D:

P_i	
P_j	

$\longrightarrow$

P_i	
P_j	

To turn an OPT schedule into an LPT schedule without increasing the total throughput time we first employ type A and C transforms to ensure that the m longest tasks are scheduled first with the longest on processor 1 and the m^{th} longest on processor m. We then employ transforms of type A and B to ensure that the $(m+1)^{th}$ longest task is scheduled second on processor m while the $(m+2)^{th}$ longest task is scheduled second on processor $m-1$. We carry on in reverse order up the list of processors untill all tasks are scheduled in an *almost* LPT fashion.

The only situation that could prevent a true LPT schedule from being generated would be if one of the tasks scheduled second on a higher numbered processor completed before one of the tasks scheduled first on a lower numbered processor. In this case a simple downward *shuffle* of type D remedies the problem.

Now, as none of the transforms used increase throughput time, the total throughput time of the resulting LPT schedule will be the same as the total throughput time of the original OPT schedule and the contradiction is established.

This completes the proof of the theorem.

2.4.8 Preemptive Schedules for more than one Processors

If we introduce preemption and remove the restriction that once a task has begun it must run to completion. we shall find that in general total throughput times can be improved. For a simple example, consider three taks of unit execution time that must be scheduled on two processors.

An optimal non-preemptive schedule is as follows:

	1	2	3
P_1			
P_2			

whereas an optimal preemptive schedule would be:

	1	2	3
P_1			
P_2			

Note that in a preemptive schedule, tasks may be stopped and restarted at will and on any processor but there must not be an **overlap** of scheduled execution time for any one task.

For any set of independent tasks the following preemptive scheduling result has been

known for some time:

Theorem

Consider an optimal m-processor **preemptive schedule** for a set of tasks, $\{T_1, T_2, \cdots, T_n\}$, with the execution time of T_i given by t_i, then the total throughput time for the preemptive schedule, t_{min} , is given by:

$$t_{min} = max\{max_i\{t_i\}, \frac{1}{m}\sum_{i=1}^{n} t_i\}$$

Proof

It is clear that t_{min} must be a lower bound for the total throughput time since no schedule can terminate in less time than it takes for the longest task to complete, and since a schedule can not be more efficient than to keep all the processors buzy throughout the duration of the schedule.

To see that t_{min} can actually be achieved by a preemptive schedule consider the following construction:

Sort the list of tasks into execution time order with the longest execution time first so that

$$t_{min} = max\{t_1, \frac{1}{m}\sum_{i=1}^{n} t_i\}$$

Now generate a schedule with total throughput time of t_{min} by scheduling T_1 on the first processor, T_2 in the remaining time on the first processor and any extra time on the second processor, T_3 in the remaining time on the current processor with any extra on the next processor. Continue in this way until all tasks are scheduled. Note that all processors will be *fully booked* if $t_1 \leq \frac{1}{m}\sum_{i=1}^{n} t_i$ but there will be free time on the higher numbered processors if this is not the case.

For an example of this consider the following set of tasks:

Task	Processing Time
T_1	6
T_2	4.5
T_3	4
T_4	3.5
T_5	3

A 3 processor preemptive optimal schedule for this set of tasks has a total throughput of:

$$t_{min} = \frac{6 + 4.5 + 4 + 3.5 + 3}{3} = 7$$

and can be constructed as

	1	2	3	4	5	6	7	8
P_1								
P_2								
P_3								

2.4.9 Scheduling Dependent Tasks

Let $T = \{T_1, T_2, \ldots, T_n\}$ be a set of tasks. Suppose the tasks are constrained to run in some order of *precedence*. The precedences are specified as a directed graph, G whose nodes are the set of tasks T and whose directed edges are members of $T \times T$ with $T_i \rightarrow T_j$ if task T_i must complete before task T_j starts. If there is a sequence of edges $T_i \rightarrow T_j \rightarrow \ldots \rightarrow T_k$ then we say that T_i is a *predecessor* of T_k and that T_k is a *successor* of T_i. Two tasks are said to be *independent* if neither is a successor of the other. Independent tasks may be executed in any order or even at the same time if more than one processor is available. Dependent tasks must be executed in the order specified by the directed edges in the precedence graph.

In the previous section we were concerned with scheduling sets of independent tasks on m processors. In this section we will investigate two algorithms for scheduling dependent tasks on m processors. The particular dependence is always given by a precedence graph.

A-Scheduling

This scheduling algorithm involves a *list ordering* sub-algorithm as follows:

To compare two lists L and L' do the following:

Step 1 Sort the two lists into decreasing order.

Step 2 If L is longer in length than L' then swap the names of the lists.

Step 3 Compare element by element until two unequal elements in the same position are found, call them l_j and l'_j. We say that $L < L'$ if $l_j < l'_j$ else $L > L'$.

Step 4 If all elements are equal up until the last element of L then $L < L'$ if L' is longer in length than L else the lists are equal.

and a *labelling* sub-algorithm which is designed to give each of the n tasks in the precedence graph a label:

Step 1: An arbitrary task T with **no** successor is chosen and given the label, 1.

Step 2: Suppose for some k, the set of labels, $1, 2, \ldots, k-1$ have already be assigned. Consider the set of tasks that have not yet received a label but whose successors have already been labeled. To each of these tasks attach a list of labels of its successors and choose the task T with the *smallest successor list* to receive the label k. Note that the smallest successor list is chosen according to the list ordering algorithm above.

Step 3: Repeat step 2 until all tasks in the precedence graph have received labels.

Once labels have been assigned to each task the scheduling algorithm is simple:

A-Schedule: Whenever a processor becomes free assign that task all of whose predecessors have already been executed and which has the largest label among those tasks not yet assigned.

Theorem

A-schedules for **two processor** systems are optimal when all tasks have equal execution times .

Proof

The proof of the fact that A-schedules are optimal under the above conditions is rather difficult and will not be attempted in this course. The interested student is refered to ***.

Examples

1) Consider a set of 22 tasks that satisfy the following precedence relations and all have unit execution times. $\{T_1 \rightarrow T_4, T_2 \rightarrow T_4, T_3 \rightarrow T_4, T_4 \rightarrow T_5, T_5 \rightarrow T_6, T_5 \rightarrow T_7, T_6 \rightarrow T_9, T_6 \rightarrow T_{10}, T_7 \rightarrow T_{10}, T_8 \rightarrow T_{11}, T_8 \rightarrow T_{12}, T_9 \rightarrow T_{12}, T_{10} \rightarrow T_{12}, T_{10} \rightarrow T_{13}, T_{10} \rightarrow T_{14}, T_{12} \rightarrow T_{15}, T_{12} \rightarrow T_{16}, T_{13} \rightarrow T_{15}, T_{13} \rightarrow T_{16}, T_{14} \rightarrow T_{15}, T_{14} \rightarrow T_{16}, T_{15} \rightarrow T_{17}, T_{16} \rightarrow T_{17}, T_{16} \rightarrow T_{18}, T_{16} \rightarrow T_{19}, T_{18} \rightarrow T_{20}, T_{18} \rightarrow T_{21}\}$.

   a) Draw the precedence graph for this set of tasks.

   b) Label the tasks according to the labeling algorithm.

   c) Produce an A-schedule for the tasks on two processors.

2) Show that an A-schedule is not necessarily optimal when three processors are involved. Use the following precedence relations to provide a counter example. Assume all tasks have equal execution times. $\{T_1 \rightarrow T_4, T_2 \rightarrow T_4, T_3 \rightarrow T_4, T_4 \rightarrow T_6, T_5 \rightarrow T_7, T_5 \rightarrow T_8, T_5 \rightarrow T_9, T_5 \rightarrow T_{10}, T_5 \rightarrow T_{11}, T_5 \rightarrow T_{12}\}$.

3) Show that an A-schedule is not necessarily optimal when the tasks involved do not have equal execution times. Use the following precedence relations to provide a counter example. Assume that all tasks except task T_3 execute in unit time while task T_3 require two units to execute. $\{T_1 \rightarrow T_4, T_1 \rightarrow T_5, T_2 \rightarrow T_4, T_2 \rightarrow T_5, T_3 \rightarrow T_5\}$

B-Scheduling

The B-schedule is optimal on any number of processors for sets of tasks, each of unit execution time, whose precedence graphs are *singly rooted trees*. Each task in the tree except for the root task has exactly one successor task. The structure of the tree must be such that the independent tasks are the leaves of the tree while the root of the tree is a task that can only start once all the other tasks in the set have been completed.

The B-schedule requires the concept of a *level* which is as follows: The root of a tree is at level 0. All tasks that are predecessors of the root are at level 1. All tasks that are predecessors of level 1 tasks are at level 2. etc. etc.

B-Schedule: Whenever a processor becomes free, assign that task if any, all of whose predecessors have already executed and which is at the **highest level** of those tasks not yet assigned. If there is a tie the an arbitrary tie-breaking rule may be used.

Example

1) Consider a set of 12 tasks that satisfy the following precedence relations and all have unit execution times. $\{T_1 \rightarrow T_3, T_2 \rightarrow T_3, T_3 \rightarrow T_9, T_4 \rightarrow T_9, T_5 \rightarrow T_{10}, T_6 \rightarrow T_{10}, T_7 \rightarrow T_{10}, T_8 \rightarrow T_{11}, T_9 \rightarrow T_{11}, T_{10} \rightarrow T_{12}, T_{11} \rightarrow T_{12}\}$
   a) Draw the precedence tree for this set of tasks.
   b) Which task is the root task.
   c) Produce an B-schedule for this set of tasks on three processors.

2.5 Virtual Memory and Paging

2.5.1 Introduction

We consider a system consisting of two memory levels, *main* and *auxiliary*. At time $t = 0$ assume that one program is residing in auxilliary memory. The program is divided into n *pages* each consisting of c contiguous addresses. The program must run in a main memory consisting of m *page frames*. If $m < n$ then a paging algorithm is required to calculate what page must be in which page frame at any particular time t.

Each time the program makes a reference we are only interested in the index of the page or page frame referenced and not with the individual words within the page. Therefore if we regard $N = \{1, 2, 3, \ldots, n\}$ as the set of pages and $M = \{1, 2, 3, \ldots, m\}$ as the set of page frames then at each moment of time there is a *page map*, $f_t : N \to M \cup \{0\}$ such that

$$f_t(x) = \begin{cases} y & \text{if page } x \text{ resides in page frame } y \text{ at time } t \\ 0 & \text{if page } x \text{ is missing from } M \text{ at time } t \end{cases}$$

When the processor generates an address α the hardware computes a memory location $\beta = f_t(x)c + \gamma$, where x and γ are determined from $\alpha = xc + \gamma$ with $0 \leq \gamma < c$. Note that if c is a power of 2 then the hardware can be organized to make this computation very efficient. If $f_t(x) = 0$ then the hardware generates a *page fault interrupt*.

When a page fault interrupt occurs the operating system must find the missing page in auxiliary memory, place it in main memory, update the map f_t, and attempt the reference again. This is the task of the paging algorithm.

Now suppose that the average time to access a word in a page in main memory is Δ_M and that the average time to transfer a page from auxiliary memory to main memory is Δ_A then an important system parameter is the ratio $\Delta = \frac{\Delta_A}{\Delta_M}$. On most operating systems $\Delta > 10^4$ but good paging algorithms should not rely on this assumption.

Now a program's paging behavior is described by its *page reference sequence*:

$$\omega = r_1, r_2, \ldots, r_t, \ldots,$$

where $r_t = i$ if page i is referenced at the t^{th} reference. Corresponding to the reference sequence is a sequence of real times

$$a_1, a_2, \ldots, a_t, \ldots,$$

such that a_t is the actual time at which reference r_t is made. The real time ellapsed between reference r_t and reference r_{t+1} is given by:

$$a_{t+1} - a_t = \begin{cases} \Delta_M & \text{if} \quad r_{t+1} \quad \text{is in memory} \\ \Delta_M + \Delta_A & \text{otherwize} \end{cases}$$

Now the *fault rate*, $F(\omega)$, is defined as the number of page faults encountered while processing reference sequence ω normalized by the length of ω. The expected ellapsed time for a reference is thus:

$$E[a_{t+1} - a_t] = \Delta_M(1 - F(\omega)) + (\Delta_M + \Delta_A)F(\omega) = \Delta_M(1 + \Delta F(\omega))$$

Thus minimizing $F(\omega)$ for all possible ω will minimize the running time of the program.

2.5.2 Demand Paging

In our study of paging algorithms we will only deal with so-called *demand paging*. Only the *missing page* is *fetched* from auxilliary memory and page replacements only occur when main memory is full. In the abstract a *demand paging algorithm*, A, is a mechanism for processing a reference sequence,

$$\omega = r_1, r_2, \ldots, r_t, \ldots,$$

and generating a sequence of *memory states*,

$$S_0, S_1, \ldots, S_t, \ldots.$$

Each memory state S_t is the set of pages from N which reside in M at time t. The memory states satisfy the following conditions:

$$S_0 = \emptyset \quad , \quad S_t \subseteq N \quad , \quad \|S_t\| \leq m \quad , \quad r_t \in S_t \quad \text{and}$$

$$S_t = \begin{cases} S_{t-1} & \text{if} \quad r_t \in S_{t-1} \\ S_{t-1} + r_t & \text{if} \quad r_t \notin S_{t-1} \quad \text{and} \quad \|S_{t-1}\| < m \\ S_{t-1} + r_t - r_s & \text{if} \quad r_t \notin S_{t-1} \quad \text{and} \quad \|S_{t-1}\| = m \quad \text{and} \quad r_s \in S_{t-1} \end{cases}$$

Note that r_t is the page demanded by the next instruction in the program and r_s is the page chosen for overwriting by the operating system's *replacement policy*.

2.5.3 Some Common Demand Paging Algorithms

Before we discuss specific paging algorithms we require four further definitions to do with a reference sequence:

$$\omega = r_1, r_2, \ldots, r_t, \ldots.$$

Firstly, the *forward distance* $d_t(x)$ at time t for page x is the distance to the first reference to x after time t:

$$d_t(x) = \begin{cases} k & \text{if } r_{t+k} \text{ is the first occurrence of } x \text{ in } r_{t+1}, r_{t+2}, \ldots \\ \infty & \text{if } x \text{ does not appear after } r_t \end{cases}$$

Secondly, the *backward distance* $b_t(x)$ is the distance to the most recent reference to x before time t:

$$b_t(x) = \begin{cases} k & \text{if } r_{t-k} \text{ is the last occurrence of } x \text{ in } r_1, r_2, \ldots, r_t \\ \infty & \text{if } x \text{ does not appear in } r_1, r_2, \ldots, r_t \end{cases}$$

Thirdly, the *reference arrival time* $l_t(x)$ denotes the last time before time t that the reference x was fetched from auxilliary memory.

$$l_t(x) = max\{i \leq t \| S_i - S_{i-1} = x\}$$

And fourthly, the *reference frequency* $\#_t(x)$ denotes the number of references to x in $r_1, r_2, \ldots, r_t$,

In the following examples of demand paging algorithms we assume that $\|S_{t-1}\| = m$ and that $r_t \notin S_{t-1}$. Also let $R(S_{t-1})$ denote the page in S_{t-1} that is replaced so that:

$$S_t = S_{t-1} + r_t - R(S_{t-1})$$

Different replacement rules, R, will give rise to different demand paging algorithms:

LRU *Least Recently Used:* The page in S_{t-1} that is replaced is the one with the largest backward distance:

$$R(S_{t-1}) = y \iff b_{t-1}(y) = max\{b_{t-1}(z) \mid z \in S_{t-1}\}$$

LFU *Least Frequently Used:* The page in S_{t-1} that is replaced is the one having received the least use. (The tie-breaking rule is usually LRU)

$$R(S_{t-1}) = y \iff \#_{t-1}(y) = min\{\#_{t-1}(z) \mid z \in S_{t-1}\}$$

FIFO *First In First Out:* The page replaced is the one that has been in memory for the longest time:

$$R(S_{t-1}) = y \iff l_{t-1}(y) = min\{l_{t-1}(z) \mid z \in S_{t-1}\}$$

LIFO *First In First Out:* The page replaced is the one that has been in memory for the shortest time:

$$R(S_{t-1}) = y \iff l_{t-1}(y) = max\{l_{t-1}(z) \mid z \in S_{t-1}\}$$

BEL Belady's Optimal Algorithm: The page replaced is the one with the largest forward distance in the sequence $r_{t+1}, r_{t+2}, \ldots$.

$$R(S_{t-1}) = y \iff d_{t-1}(y) = max\{d_{t-1}(z) \mid z \in S_{t-1}\}$$

If two or more pages have infinite forward distance then the page with the smallest page number is chosen for replacement. This rule cannot effect the fault-rate performance as any page with infinite forward distance is never used again.

Note that Belady's algorithm is *unrealizable* since it requires a look into the future operation of the program. However it does provide a useful benchmark against which to measure the performance of the other *realizable* algorithms.

2.5.4 The Optimality of Belady's Algorithm

Theorem

Belady's demand paging algorithm is optimal in the sense that it results in the minimum achievable *paging cost* when processing any reference sequence ω. (paging costs are measured in units of page replacements and they only start mounting up once memory is full)

Proof

Let $>$ denote a linear ordering of the references in ω such that $y > z$ if y has greater forward distance than z at time t.

Let $C_k(S + r_t - y, t)$ denote the cost of processing the references, $r_{t+1}, r_{t+2}, \ldots, r_{t+k}$ starting from state S at time t. Note that page r_t is entering S and overwriting page y. For Belady's algorithm to be optimal we must show that for all k:

$$y > z \Rightarrow \Delta C_k = C_k(S + r_t - z, t) - C_k(S + r_t - y, t) \geq 0$$

since if this is the case then the y to choose to obtain minimal achievable cost is just the y with greatest forward distance. We will show that $\Delta C_k = 0 \vee 1$ by induction on k. The result is trivial for $k = 0$ since we are then considering processing the next *zero* page references and any algorithm is optimal. Now suppose that $y > z \Rightarrow \Delta C_j = 0 \vee 1$ for $j = 0, 1, \ldots, k-1$ we must show that the same statement is true when $j = k$. There are three cases to be considered:

Case 1: $r_{t+1} \in S - y - z$ In this case we have:

$$\begin{aligned} \Delta C_k &= C_k(S + r_t - z, t) - C_k(S + r_t - y, t) \\ &= C_{k-1}(S + r_t - z, t+1) - C_{k-1}(S + r_t - y, t+1) \end{aligned}$$

which is $0 \vee 1$ by the induction hypothesis.

Case 2: $r_{t+1} = z$ In this case we have:

$$\begin{aligned}\Delta C_k &= C_k(S + r_t - z, t) - C_k(S + r_t - y, t) \\ &= 1 + C_{k-1}(S + r_t - z + r_{t+1} - u, t + 1) - C_{k-1}(S + r_t - y, t + 1) \\ &= 1 - [C_{k-1}(S + r_t - y, t + 1) - C_{k-1}(S + r_t - u, t + 1)\end{aligned}$$

where by the induction hypothesis u has greatest forward distance in $S + r_t - z$. So $u \geq y$ in the linear ordering and the term in square brackets is either 0 if $u = y$ or $0 \vee 1$ if $u > y$ by the induction hypothesis. So again in this case ΔC_k is $0 \vee 1$.

Note: We need not consider the case $r_{t+1} = y$ since we assume that $y > z \geq r_{t+1}$

Case 3: $r_{t+1} \notin S + r_t$ In this case we have:

$$\begin{aligned}\Delta C_k &= C_k(S + r_t - z, t) - C_k(S + r_t - y, t) \\ &= [1 + C_{k-1}(S + r_t - z + r_{t+1} - u, t + 1)] - [1 + C_{k-1}(S + r_t - y + r_{t+1} - v, t + 1)] \\ &= C_{k-1}(S + r_t - z + r_{t+1} - u, t + 1) - C_{k-1}(S + r_t - y + r_{t+1} - v, t + 1)\end{aligned}$$

where u has greatest forward distance in $S + r_t - z$ and v has the greatest forward distance in $S + r_t - y$. Now let s be the element of $S + r_t - z - y$ with the greatest forward distance then there are three possibilities in the ordering of s, y and z.

a) $s > y > z$ In this case $u = v = s$ and ΔC_k reduces to:

$$C_{k-1}((S + r_t + r_{t+1} - s) - z, t + 1) - C_{k-1}((S + r_t + r_{t+1} - s) - y, t + 1)$$

which is $0 \vee 1$ by the induction hypothesis.

b) $y > s > z$ In this case $u = y$ and $v = s$ and ΔC_k reduces to:

$$C_{k-1}((S + r_t + r_{t+1} - y) - z, t + 1) - C_{k-1}((S + r_t + r_{t+1} - y) - s, t + 1)$$

which is $0 \vee 1$ by the induction hypothesis since $s > z$.

c) $y > z > s$ In this case $u = y$ and $v = z$ and ΔC_k reduces to:

$$C_{k-1}(S + r_t - z + r_{t+1} - y, t + 1) - C_{k-1}(S + r_t - y + r_{t+1} - z, t + 1)$$

which is 0.

Thus by induction ΔC_k is $0 \vee 1$ for all k and the optimality of Belady's algorithm is established.

2.6 Computer Security

2.6.1 Introduction

Computer Security has been the subject of intensive research since multi-user operating systems were first introduced. Its importance continues to grow as more sensitive information is stored, transmitted and processed by computers. Some applications include the military, banks, credit bureaus and hospitals. Security flaws of computer systems and approaches to penetration have been enumerated in the literature. Here are some of the more common flaws:

- *The system does not authenticate itself to the user.* A common way to steal passwords is for an intruder to leave a running process which masquerades as the standard system logon. After an unsuspecting user enters an identification and a password, the masquerader records the password, gives an error message (identical to the standard one provided by the logon process in the case of a mistyped password) and aborts. The true logon process is left to take care of any retry.
- *Improper handling of passwords.* Passwords may not be encrypted, or the table of encrypted passwords may be exposed to the general public, or a weak encryption algorithm may be used.
- *Improper implementation* A security mechanism may be well thought out but improperly implemented. For example, timely user abortion of a system process may leave a penetrator with system administrator access rights.
- *Trojan horse:* A borrowed program may surreptitiously access information that belongs to the borrower and deliver this information to the lender.
- *Clandestine code:* Under the guise of correcting an error or updating an operating system code can be embedded to allow subsequent unauthorized entry to a system

In this section we will study cryptographic methods for access control and message protection.

2.6.2 Encryption Systems

An *encryption system* is an encryption procedure executed by a *sender*, which takes a message (called the *plain-text*) and a small piece of information (called the *key*) and creates an encoded version of the message (called the *cipher-text*). The cipher-text is transmitted along an open line to a *receiver* who must then use a decrypting procedure together with the key to recover the plain-text. The key is arranged in advance between sender and receiver.

When we consider the quality of an encryption system, we assume that a third-party trying to decode the message knows the encryption and decryption procedures and has a copy of the cipher-text. The only thing missing is the key. We also assume that the sender does not spend time trying to contrive a difficult to read message but relies entirely on the encryption system to provide all the needed security.

A more demanding standard for measuring the quality of an encryption system is that it should be safe against a *chosen plain-text attack*. It is often possible for the third party to process a known message through the encryption procedure and thus obtain a plain-text cipher-text pair from which it may be possible to deduce the key.

2.6.3 Examples

Simple Substitution

This system involves a simple letter-for-letter substitution method. The *key* is a rearrangement of the 26 letters of the alphabet. For example if the *key* is given as:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
actqgwrzdevfbhinsymujxplok
```

and the *plain-text* message starts as follows

```
THE SECURITY OF THE RSA ENCODING SCHEME ...
```

then the *cipher-text* message will read:

```
uzg mgtjyduo iw uzg yma ghtiqdhr mtzgbd ...
```

Most messages can de decoded without the key by looking for frequently occuring pairs of letters. (TH and HE are by far the most common pairings to be found in most english messages). Once these letters have been itentified the rest usually fall into place easily. Of course this system is useless against a *known plain-text attack*.

The Vigenere Cipher

This cipher works by replacing each letter by another letter a specified number of positions further down the alphabet. For example, `J` is `5` positions further down from `E` and `D` is `5` positions on from `Y`. The *key* in this cipher is a sequence of shift ammounts. If the sequence is of length `10` the the first member of the key is used to process the letters in positions `1, 11, 21, ...` in the plain-text. The second member of the key is used to process the letters in positions `2, 12, 22, ...` and so on. For example if we use the key:

```
3 1 7 23 10 5 19 14 19 24
```

and the *plain-text* message starts as follows

```
THE SECURITY OF THE RSA ENCODING SCHEME ...
```

then the *cipher-text* message will read:

```
wil pohnfbrb pm qrj kgt cqdvassz gvfhnl ...
```

This type of cipher was considered very secdure up until 1600 AD but it is not very difficult to crack. If the length of the key is known then a guess at the first key element coupled with a table showing possible two letter combinations in positions `1,2` and `11,12` and `21,22` etc will usually reveal the second element of the key. The same technique can be used to get the rest of the key. Again this cipher is useless against a *known plain-text attack*.

One-Time Pads

In the previous example, if the key-sequence is long enough then the cipher becomes harder and harder to crack. In the extreen case when the key-sequence is as long as the plain-text itself the cipher is theoretically *unbreakable* (since for any possible plain-text there is a key for which the given cipher-text comes from that plain-text). This type of cipher has reportedly been used by spies, who were furnished with notebooks containing page after page of randomly generated key-sequences. Note that it is essential that each key-sequence be used only once, (hence the name *one-time pad*.

One-time pads seem practical when an agent is communicating with a central command. They become less attractive if several agents need to communicate with each other.

2.6.4 Introduction to Number Theory

To ensure that cipher systems are safe one usually resorts to *Number Theory*. Before presenting some number theoretic cipher systems we must revise our number theory background.

Congruences

The *congruence* $a \equiv b \bmod n$ says that when divided by n, a and b have the same remainder. For example:

$$100 \equiv 34 \bmod 11 \quad , \quad -6 \equiv 10 \bmod 8$$

In the second example we are using $-6 = 8(-1) + 2$. Note that we always have $a \equiv b \bmod n$ for some $0 \leq n \leq n-1$, and we are usually only concerned with that b.

If $a \equiv b \bmod n$ and $c \equiv d \bmod n$ then we can *add* or *multiply*:

$$a + c \equiv b + d \bmod n \quad , \quad ac \equiv bd \bmod n$$

.

Division however does not always work:

$$6 \equiv 18 \bmod 12 \quad , \quad 3 \not\equiv 9 \bmod 12$$

The Greatest Common Divisor

For any two numbers, a and b, the number (a, b) is the largest number which divides a and b evenly. For example:

$$(56, 98) = 14 \quad \text{and} \quad (76, 190) = 38$$

(a, b) is called the *greatest common divisor* of the integers a and b.

Theorem 1 *For any two non-zero integers a and b, there are two other integers x and y, with the property that (a, b) is the smallest positive integer that can be expressed as:*

$$(a, b) = ax + by$$

Proof: Consider the set S of all integers that can be written in the form $ax+by$, and let S^+ be the set of all positive integers in S. Now the set S contains the integers $a, -a, b$ and $-b$, so the set S^+ is not empty. Thus S^+ must have a least element, call this element d. We must show that $d = (a, b)$.

First by the division algorithm there are integers q and r such that $a = dq + r$, with $0 \leq r < d$. Thus $r = a - dq$ and since $d \in S^+$ we have $r = a - (ax + by)q$ and a little algebra gives $r = a(1 - xq) + b(-nq)$. Thus r is in S but since $0 \leq r < d$ and since d is the smallest element of S^+ we must have $r = 0$. So $d|a$.

Similarily it can be shown that $d|b$.

Now suppose that $c|a$ and $c|b$, then $a = cu$ and $b = cv$. Thus $d = ax + by = cux + cvy = c(ux + vy)$ which shows that $c|d$. Thus d is the *greatest common divisor* of a and b. So $d = (a, b)$.

Euclid's Algorithm

Given a and b the equation $ax + by = d$ can be solved by making a sequence of simplifying substitutions. Suppose $a < b$ then divide a into b getting a quotient q and remainder r so that $b = aq + r$. Now rewrite:

$$ax + by = d$$

as

$$ax' + ry = d \quad \text{where} \quad x' = x + qy$$

Now try to solve the equation $ax' + ry = d$ by employing the same technique, $r < a$ so divide r into a getting a quotient and remainder and rewrite the equation, etc etc. Eventually one ends up with an equation of the form $sx' + ty' = d$ where one of s and t is 0 while the other is d. Consider the following example where we are trying to compute $(30, 69)$.

$$\begin{array}{rcll} 30x + 69y & = & d & \\ 30x' + 9y & = & d & [x' = x + 2y] \\ 3x' + 9y' & = & d & [y' = y + 3x'] \\ 3x'' + 0y' & = & d & [x'' = x' + 3y'] \end{array}$$

from the last line of this reduction we can read off $x'' = 1$ and $y' = 0$ is a solution if $d = 3$. Note that back-substitution will give $x = 7$ and $y = -3$ and the solution to the original problem is:

$$(30, 69) = 3 = (30)(7) + (69)(-3)$$

It is important to realize that this process is feasable on a computer even if a and b are several hundred digits long. It is easy to show that the larger of the two coefficients decreases by $\frac{1}{2}$ every two equations. Thus in twenty iterations the larger coefficient will decrease by a factor of $2^{-10} < 10^{-3}$. The greatest common divisor of two 600 digit numbers could be computed in no more than 4000 iterations.

Corollary 2 *If p is prime and if $ar \equiv as \bmod p$ and if $a \not\equiv 0$ then $r \equiv s \bmod p$.*

Proof: Since p is prime we have $(a, p) = 1$ so there are integers x and y such that $ax + py = 1$. Hence $ax \equiv 1 \bmod p$ and $r \equiv (1)r \equiv axr \equiv xar \equiv xas \equiv s \bmod p$.

Corollary 3 *If p is prime and $a \not\equiv 0 \bmod p$ then for any b there is a y with $ay \equiv b \bmod p$.*

Proof: In the preceeding proof we found an x with $ax \equiv 1 \bmod p$. Now just take $y = bx$ and the result follows.

Powers modulo a Prime

The sequence $a, a^2, a^3, \ldots, \mathrm{mod} p$ has many applications in cryptography. Before looking at the theoritical properties of such a sequence you should convince yourself that it is feasable to compute $a^b \bmod p$ even if a and b are several hundred digits long. The trick is to compute $a^2, a^4, a^8, \ldots$ using $\bmod p$ arithmetic at each step until you are almost there.

For example to compute $432^{687} \bmod 987$ we note that:

$$687 = 512 + 128 + 32 + 4 + 2$$

so that in

$$\begin{aligned} 432^{687} &= (432^2)(432^4)(432^{32})(432^{128})(432^{512}) \\ &\equiv (81)(639)\ldots(858) \bmod 987 \\ &\equiv 204 \bmod 987 \end{aligned}$$

Note that even if the numbers are several hundred digits long then although special routines must be written to handle the *modulo multiplications*, these calculations with exponents will be feasable.

We will now develope a series of theorems involving powers of numbers in some modulo arithmetic field.

Theorem 4 *Suppose $b \not\equiv 0$ and let d be the smallest number such that $b^d \equiv 1$. Then for any $e > 0$, $b^e \equiv 1$ implies $d|e$.*

Proof: If $d \not| e$ then $e = dq + r$ for some $0 < r < d$ and $b^r \equiv b^{e-dq} \equiv b^e(b^d)^{-q} \equiv 1$ which contradicts the definition of d.

Theorem 5 *There are at most d solutions to a polynomial congruence of degree d:*

$$\alpha_0 x^d + \alpha_1 x^{d-1} + \ldots + \alpha_d \equiv 0 \bmod p$$

Proof: This theorem is proved in the same way as the corresponding theorem in ordinary algebra: If $x = \beta$ is a solution the the polynomial can be written as $(x - \beta)$ times a polynomial of degree $d - 1$ which by induction has at most $d - 1$ solutions.

Primitive Roots

In cryptography we often work in a field $\bmod p$ where p is some large prime number. We will be interested in elements of this field whose powers that take on all possible values in the field. Such an element is called a *primitive root*. Here is a more formal definition:

Definition: a is called a *primitive root* of p if for every b between 1 and $p - 1$ there is an x between 1 and $p - 1$ such that $a^x \equiv b \bmod p$.

Primitive roots are only useful if we know they exist. The following theorem which is quite hard to prove ensures the existance of a primitive root for appropriately chosen p.

Theorem 6 *If p is prime then a primitive root a exists for modulo p arithmetic.*

Proof: Choose any $a \not\equiv 0$ and let d be the smallest positive number for which $a^d \equiv 1$, (there must be such a number since $a^K \equiv a^L$ implies $a^{K-L} \equiv 1$). If $d = p-1$ then a is a primitive root. If $d < p - 1$, we will find a' and d' with $d' > d$ such that $(a')^{d'} \equiv 1$ and the process can be repeated until a primitive root is found.

We have $a^d \equiv 1$ so the sequence $a, a^2, a^3, \ldots, a^d \equiv 1$ consists of d different solutions to the polynomial equation $x^d \equiv 1$. If $d < p - 1$ then let b be any non-member of the sequence. Let e be the smallest positive number with $b^e \equiv 1$. If $e > d$ then we can take $a' = b$ and $d' = e$ and we are done so from now on assume that $e \leq d$. Also note that e does not divide d so that $\frac{e}{(d,e)} > 1$.

Now let $a' = a^{(d,e)}b$ and let $c = \frac{d}{(d,e)}$ and let $d' = ce > d$.

We must show that ce is the smallest number with the property that $(a')^{ce} \equiv 1$.

First note that $(a')^{ce} \equiv (a^d)^e(b^e)^c \equiv 1$

Also note that $(c, e) = (\frac{d}{(d,e)}, e) = 1$ so by Euclid there exist integers K and L such that $cK + eL = 1$.

Now assume that $(a')^x \equiv 1$, then we have $1 \equiv (a')^{cx} \equiv b^{cx}$. So $cx = eM$ for some integer M and $x = (cK + eL)x = e(KM + Lx)$. So $x = ey$ for some integer y We will show that y is divisable by c and we are done.

$$\begin{aligned}(a')^x \equiv 1 &\Rightarrow (a^{(d,e)}b)^{ey} \equiv 1 \\ &\Rightarrow a^{(d,e)ey} \equiv 1 \\ &\Rightarrow (d,e)ey = dN \quad \text{for some integer } N \\ &\Rightarrow ey = cN \\ &\Rightarrow y = (cK + eL)y = c(Ky + LN)\end{aligned}$$

Thus x is divisable by ce so ce is the smallest number with the property that $(a')^{ce} \equiv 1$ but as already mentioned $ce > d$ and hence a' is a better candidate than a for a primitive root. This completes the proof and some corollaries follow easily.

In the following three corollaries assume that p is prime:

Corollary 7 *If a is a primitive root of p then $a^{p-1} \equiv 1 \bmod p$.*

Proof: We know that $a^d \equiv 1$ for some d between 1 and $p-1$. If $d < p-1$ then the sequence of powers of a would start repeating before all the numbers between 1 and $p-1$ were obtained and then a would not be a primitive root.

Corollary 8 *For any $b \not\equiv 0$ we will always have $b^{p-1} \equiv 1 \bmod p$.*

Proof: Let a be a primitive root then using the previous corollary we have $b^{p-1} \equiv (a^x)^{p-1} \equiv (a^{p-1})^x \equiv 1$.

Corollary 9 *If $x \equiv y \bmod p-1$ then $b^x \equiv b^y \bmod p$.*

Proof: For some integer r we have $y = r(p-1)+x$ thus $b^y \equiv (b^{p-1})^r b^x \equiv b^x \bmod p$.

2.6.5 The Discrete Logarithm Problem

The existance of a primitive root a for any prime p shows that the equation

$$a^x \equiv b \bmod p$$

has a solution for any $b \not\equiv 0$. We have seen that given the left hand side of this equation it is usually feasable to compute the right hand side even when the integers involved are large. Going the other way however is much harder. Given a primitive root a and any element b the computation of x to satisfy the above equation is called the *discrete logarithm problem*. x is called the *discrete logarithm* of b with respect to the primitive root a modulo prime p. Many modern encryption systems are based on the fact that no efficient way of computing discrete logarithms is known.

To make use of the discrete logarithm problem to build an encryption system one must have a reliable method of finding at least one primitive root a given any prime

p. A little analysis shows that in most cases it will be sufficient to choose a at random and then test for primitivity. If a turns out to be not primitive then choose another a at random.

The analysis goes as follows: It is easy to show that if a is a primitive root then a^x is a primitive root if $(x, p-1) = 1$.

Firstly:

$$\begin{aligned}(a^x)^n \equiv 1 \bmod p &\rightarrow a^{nx} \equiv 1 \\ &\rightarrow (p-1)/nx \\ &\rightarrow nx = r(p-1)\end{aligned}$$

Thus:

$$\begin{aligned}(x, p-1) = 1 &\rightarrow Ax + B(p-1) = 1 \\ &\rightarrow Anx + Bn(p-1) = n \\ &\rightarrow Ar(p-1) + Bn(p-1) = n \\ &\rightarrow (Ar + Bn)(p-1) = n \\ &\rightarrow p - 1/n\end{aligned}$$

and so we have shown that $(a^x)^n \equiv 1 \rightarrow (p-1)/n$ which means a^x is a primative root.

Now suppose $p-1$ has a prime factor decomposition:

$$p - 1 = (p_1)^{\alpha_1}(p_2)^{\alpha_2} \dots (p_n)^{\alpha_n}$$

then the number of primitive roots would be given by the number of x's relatively prime to $p-1$:

$$\#(x's) = (p-1)(\frac{p_1 - 1}{p_1})(\frac{p_2 - 1}{p_2}) \dots (\frac{p_n - 1}{p_n})$$

For example if $p = 1223$ then $p - 1 = 1222 = (2)(13)(47)$ and

$$\#(x's) = 1222(\frac{1}{2})(\frac{12}{13})(\frac{46}{47}) \approx 0.45$$

so choosing a at random would succeed 45% of the time. This is an example of a *probabilistic algorithm* and problems in cryptography are often solved by means of them.

2.6.6 The Diffie-Hellman Key exchange procedure

As a first example of how the intractability of the discrete logarithm problem may be used in a cryptographic setting consider the problem of two people, A and B,

trying to agree on a secret key knowing that a third party, C, is listening to all communications between them.

The technique is as follows: A and B agree publically on a large prime p and a primitive root a. These numbers will also be known to C. Then A secretly chooses a large number α while B secretly chooses a large number β. Then $a^\alpha \bmod p$ and $a^\beta \bmod p$ are computed by A and B respectively and publically announced. The secret key which will be known only to A and B can then be computed by them as:

$$\text{secret key} \quad = (a^\beta)^\alpha \bmod p = (a^\alpha)^\beta \bmod p$$

Note that for C to compute the secret key he would have to determine either α or β from his knowledge of p, a, a^α and a^β. In other words he would have to solve the discrete logarithm problem for large modulo arithmetic which no one to this date has been able to do.

2.6.7 The Code Protection Problem

As another example of the use of the intractability of the discrete logarithm problem consider the following scheme for protecting code against piracy.

The *author* of the code selects a large prime p with primitive root a and stores these as constants in the code. The author also chooses a secret number c for that copy of the code and stores $a^c \bmod p$ as a constant in the code.

At startup the code computes a machine identity h. This identity could be a combination of the BIOS id and manufacturing date together with the hard disk id and formatting date.

The code then computes $a^h \bmod p$ and makes this number known to the *user* by displaying it on the screen. The user then phones the author and tells him over the phone the number displayed on the screen.

If that user is currently paid-up then the author then computes a password $(a^h)^c \bmod p$ and then phones the user back to inform him of his password.

The user enters the password from the keyboard and the code computes $(a^c)^h \bmod p$ to determine if access is granted.

Note that the user cannot compute the password before the code is run since neither c nor h is obtainable from non-executing code. Naturally tracing must be prohibited to prevent the password being detected at access determination time.

2.6.8 The Rivest-Shamir-Adleman public key system

The idea of *public key encryption* is to allow a *receiver* to set up a system so that anyone can send him an encoded message, but only the receiver will be able to decode it. The plan is as follows:

The receiver chooses two large primes p and q. He then computes a number e that is relatively prime to both $p-1$ and $q-1$. In other words:

$$(e, p-1) = (e, q-1) = 1$$

. He also computes another number d such that:

$$ed \equiv 1 \bmod (p-1) \quad \text{and} \quad ed \equiv 1 \bmod (q-1)$$

Finally the receiver computer the product of p and q:

$$n = pq$$

The receiver keeps p, q and d secret and publishes e and n. To send a message $M < n$ to this receiver, any member of the public can compute $M^e \bmod n$ and transmit M^e to the receiver safe in the knowledge that no evesdropper can recover M from M^e. Rivest, Shamir and Alderman showed that the receiver can recover M from M^e by computing $(M^e)^d \bmod n$. This public key encryption technique has become widely used and is known as RSA encryption.

To show that RSA encryption is feasable we must show that it is feasable to compute e and d from knowledge of p and q and we must also show that $(M^e)^d \equiv M \bmod n$. Lastly the reader must be convinced that it is extremly hard to compute p and q from n so that the secrecy of d is guarenteed.

Firstly to get e such that $(e, p-1) = (e, q-1) = 1$ just choose e to be prime and greater than $\frac{p}{2}$ and $\frac{q}{2}$.

Secondly to find d such that $ed \equiv 1 \bmod (p-1)$ and $ed \equiv 1 \bmod (q-1)$ we solve

$$ex + (p-1)(q-1)y = 1$$

for x any y via *Euclid's algorithm* with *back substitution* and let $d = x$.

Thirdly to show that $(M^e)^d \equiv M \bmod p$ we use the last corollary from the section on number theory which states that if $x \equiv y \bmod p-1$ then $b^x \equiv b^y \bmod p$. We have $ed \equiv 1 \bmod (p-1)$ so the corollary tells us that $M^{ed} \equiv M^1 \bmod p$. Similarily $M^{ed} \equiv M^1 \bmod q$ Thus $M^{ed} - M$ is divisable by both p and q so $(M^e)^d \equiv M \bmod (pq = n)$

Lastly to convince yourself that the factors p and q can remain secret even if n is known consider the fact that the crude approach of dividing n by all numbers up until $\sqrt{n}$ would take approximately 10^{50} steps for a 100 digit n and in the last 100 years many famous mathematicians have been unable to devise a significantly better factoring algorithm.

2.6.9 Authentication and Digital Signatures

A problem with public key encryption is that it is easy for a troublemaker C to send a message to A pretending to be B. This problem can be solved if both A and B have published encryption keys. The solution is as follows:

Suppose B wants to send a message M to A. He first encrypts the message using his own private decryption key d_B to get $M^{d_B} \bmod n_B$. He then prepends his name and encrypts the result using $A's$ public encryption key to get $(B + (M^{d_B} \bmod n_B))^{e_A} \bmod n_A$ This *mess* is sent to A via an open line. A decrypts the *mess* using his private decryption key d_A and discovers B's name at the begining of an encrypted message. A then decrypts the rest of the message using $B's$ public encryption key e_B. If the result makes sense A is secure in the knowledge that only someone knowing $B's$ private decryption key d_B could have sent the message.

2.6.10 Secure Shell Environment:

`ssh2` (Secure Shell) is a program for logging into a remote machine and executing commands in a remote machine. It is intended to replace rlogin and rsh, and provide secure, encrypted communications between two untrusted hosts over an insecure network. X11 connections and arbitrary TCP/IP ports can also be forwarded over the secure channel.

`ssh2` connects and logs into the specified hostname. The user must prove his identity to the remote machine using some authentication method.

Public key authentication is based on the use of digital signatures. Each user creates a public / private key pair for authentication purposes. The server knows the user's public key, and only the user has the private key. The filenames of private keys that are used in authentication are set in .ssh2/identification. When the user tries to authenticate himself, the server checks .ssh2/authorization for filenames of matching public keys and sends a challenge to the user end. The user is authenticated by signing the challenge using the private key.

If other authentication methods fail, ssh2 will prompt for a password. Since all communications are encrypted, the password will not be available for eavesdroppers.

When the user's identity has been accepted by the server, the server either executes the given command, or logs into the machine and gives the user a normal shell on the remote machine. All communication with the remote command or shell will be automatically encrypted.

If no pseudo tty has been allocated, the session is transparent and can be used to reliably transfer binary data.

The session terminates when the command or shell in on the remote machine exits and all X11 and TCP/IP connections have been closed. The exit status of the remote

program is returned as the exit status of ssh2.

Ssh2 automatically maintains and checks a database containing the host public keys. When logging on to a host for the first time, the host's public key is stored in a file .ssh2/hostkey-PORTNUMBER-HOSTNAME.pub in the user's home directory. If a host's identification changes, ssh2 issues a warning and disables the password authentication in order to prevent a Trojan horse from getting the user's password. Another purpose of this mechanism is to prevent man-in-the-middle attacks which could otherwise be used to circumvent the encryption.

`ssh2` exercise

`ssh2` has been installed on your unix box. Download `ssh2` for windows from `www.ssh.com` and try to establish a secure shell connection to your unix box. Remember that `ssh2` has replaced `ssh`, the original secure shell command. Use `man ssh2` to get information on `ssh2` options. Also make use of `keysgen` to generate a private/public pair of keys and set up ssh so that you can start a unix session without transmitting a password.

2.7 Further Reading

This set of notes is suppose to be self contained. The following books and articles are not required reading for this course but they may help you to understand some of the topics presented.

Paul Sheer, Rute Users Tutorial and Exposition, 2000.

Rute is a dependency consistant UNIX tutorial. This means that you can read it from beginnning to end in consecutive order. This book can be downloaded from: `http://hughm.cs.unp.ac.za/ murrellh/notes/rute.ps`

David Rusling, The Linux Kernel, 1999.

This book is for Linux enthusiasts who want to know how the Linux Kernel works. It describes the principles and mechanisims that Linux uses. This book can be downloaded from: `http://hughm.cs.unp.ac.za/ murrellh/notes/tlk.ps`

Silberschatz and Gavin, Operating Systems Concepts, Willey, 6th edition

A standard introduction to os concepts.

Coffman and Denning, Operating Systems Theory, Prentice Hall, 1973

This book contains proofs for many of the more difficult theorems discussed in this course.

Maekawa and Oldehoeft, Operating Systems, Benjamin-Cumings, 1987

Not as advanced as Coffman but complements Coffman nicely.

Nishinuma and Espesser, Unix First Contact, Macmillan, 1987

Exactly what it claims to be, a first contact introduction.

Pilavakis, Unix Workshop, Macmillan, 1989

Good introduction to UNIX with an excellent chapter on inter-process communication.

Filipski, Making Unix secure, Byte, pp. 113-128, April 1986

Describes security issues with respect to the UNIX operating system. Read the article and in particular the password encryption scheme. The UNIX password encryption is based on DES, the Data Encryption Standard.