Introduction to the UNIX Shell

by Mark Virtue

and The Virtual Training Company

Chapter 1

Introduction

About this Course

- This course contains instruction on the following topics:
  - The UNIX shell (specifically the *Bourne* family of shells)
  - Common text-based (command-line) UNIX programs, such as
    - ls
    - vi
    - grep
    - etc
- UNIX GUI (graphical user interface, such as X-Windows) usage will *not* be taught
- The commands taught in this course are compatible with *all* brands of UNIX (including Linux)

Audience and Prerequisites

- You might want to learn about the UNIX shell if:
  - You are using a version of UNIX at work/home/school (e.g. Linux) and want to know more about the commands available
  - You have a need to log into a computer remotely across the Internet and run commands there (e.g. your ISP’s computer)
  - You need to learn the basics of the shell so that you can then learn about shell script programming for the purposes of UNIX system administration or CGI programming
- No prior knowledge of UNIX or UNIX shell commands is assumed
- Familiarity with computers in general and command-line interfaces (such as DOS) in particular will be helpful, but is not necessary

Chapter 2

Understanding UNIX

What is UNIX?

- UNIX is the name given to a family of *operating systems*
  - An operating system is a software platform upon which programs may be run
  - Windows 2000 is an example of a (non-UNIX) operating system
  - Each operating system is usually tied to only one type of computer (such as Intel x86 or Sun SPARC)
- There are many flavours (or *variants*) of UNIX produced by a variety of technology companies
- Many are incompatible with each other, in the sense that they cannot run each others' programs
- In order to fully understand what UNIX is – and how the different versions are related to each other – it is necessary to know something of UNIX's history

UNIX History

- UNIX was originally developed in the early 1970s by AT&T’s Bell Laboratories
- It was so useful an operating system that other organisations (including universities) expressed interest in developing versions of their own, and were given the source code for free
- Soon all the large computer vendors were marketing their own (diverging) versions of UNIX optimised for their own computer architectures, boasting many different strengths and features (including Microsoft’s own effort: Xenix)

UNIX History (cont.)

- It quickly became apparent that, although UNIX systems were available everywhere, they seldom were able to interoperate without significant effort. The trademark UNIX was ubiquitous, but it was applied to a multitude of different, incompatible products
- In 1987, the two leading vendors of UNIX – AT&T (System V) and Sun Microsystems (BSD) combined their efforts to produce *System V Release 4* (SVR4), foisting what they hoped was a new standard upon the UNIX world

UNIX History (cont.)

- This only served to further divide the industry, and many other market players banded together to develop their *own* open UNIX variant, called OSF/1 (Open Software Foundation UNIX version 1)
- To introduce a sense of unity, an organisation called X/Open began putting in place a set of open standards that would allow greater interoperability between UNIXs
- In 1993, AT&T sold their UNIX business to Novell, who sold it to X/Open (now The Open Group)
- Linus Torvalds ported a version of UNIX to the PC and gave the source code to the community at large – Linux was born

Which UNIX?

- There are currently hundreds of UNIX variants running on computers around the world
- There are roughly 80 still being developed and supported
- The following table details some of the major variants

Which UNIX? (cont.)

UNIX Variant	Company
AIX	IBM
A/UX	Apple
FreeBSD (free)	
HP-UX	Hewlett-Packard
IRIX	Silicon Graphics
Linux (free, PC)	*Various*
NEXT	Next
SCO-UNIX (PC)	Santa-Cruz Organisation
Solaris	Sun
Ultrix	Digital
UnixWare (PC)	Novell (now SCO)
QNX (real-time)	Quantum Software

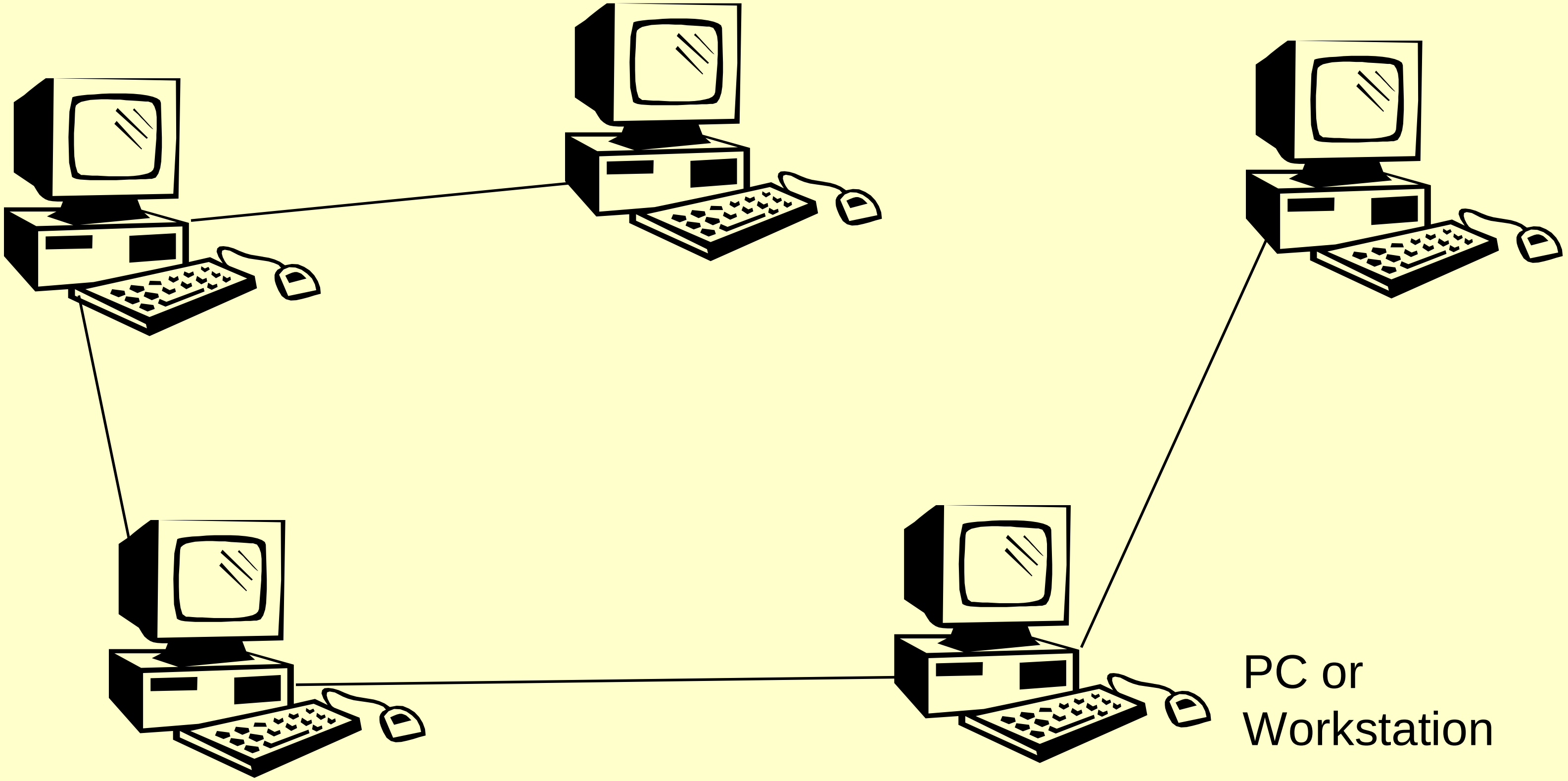
Which UNIX? (cont.)

- How compatible are these variants of UNIX? Specifically, if a person learns *one* UNIX, will those skills by useful when running another UNIX?
- The answer is complex:
  - For end users, the UNIX *shell* (the subject of this course) is virtually identical across all UNIX's
  - Some UNIX's offer GUIs (X-Windows, Motif, etc). These can differ widely, but having learned one, most end users will find the others relatively easy to learn
  - For advanced users, UNIX shell-scripting is virtually identical across all UNIX's
  - UNIX system administrators will find significant differences when administering a variety of UNIX systems

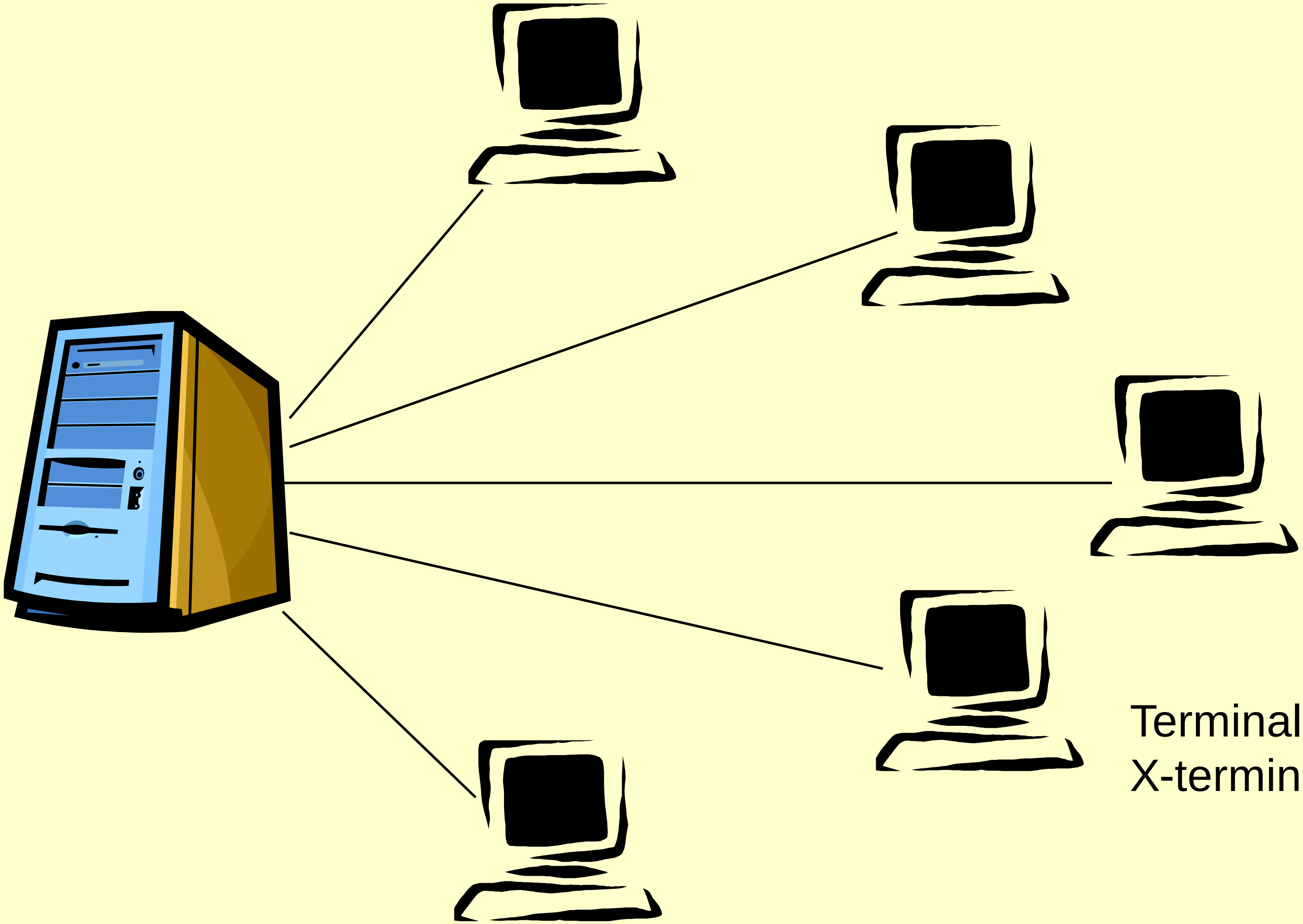
UNIX Architecture

- UNIX is a *terminal*-based operating system, meaning:
  - Many users may be simultaneously using the one computer, each with their own keyboard and monitor (and sometimes mouse) – known as a *terminal* (compare this with PCs, which have only a single keyboard and monitor for each computer)
  - The programs that each user is running are all competing for the computers CPU time and memory. This means that one user can potentially slow down the system for the rest of the users on the system
- Some UNIX terminals are capable of displaying a GUI, complete with icons and mouse support. Such terminals are known as *X-terminals*
- UNIX is also capable of existing within a *network*

Five Computers on a Network



One Computer with Five Terminals



Terminal or
X-terminal

Chapter 3

Understanding the UNIX Shell

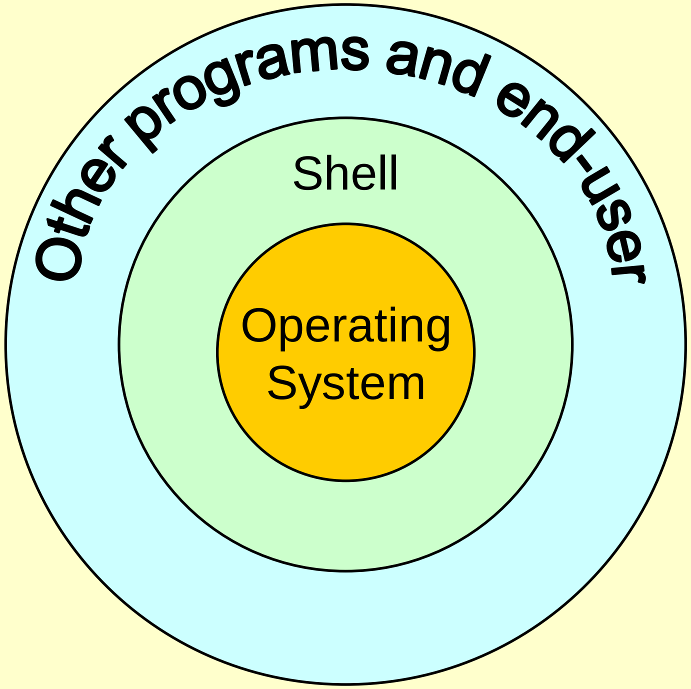
What is the UNIX Shell?

- What is a “shell”?
- A shell is simply “a program that is used to start other programs”
- All operating systems have shells

Operating System	**Shell**
DOS	`command.com`
Windows 3.1 / NT3.x	Program Manager
Windows 95 / 98 / ME / NT4 / 2000 / XP	Windows Explorer

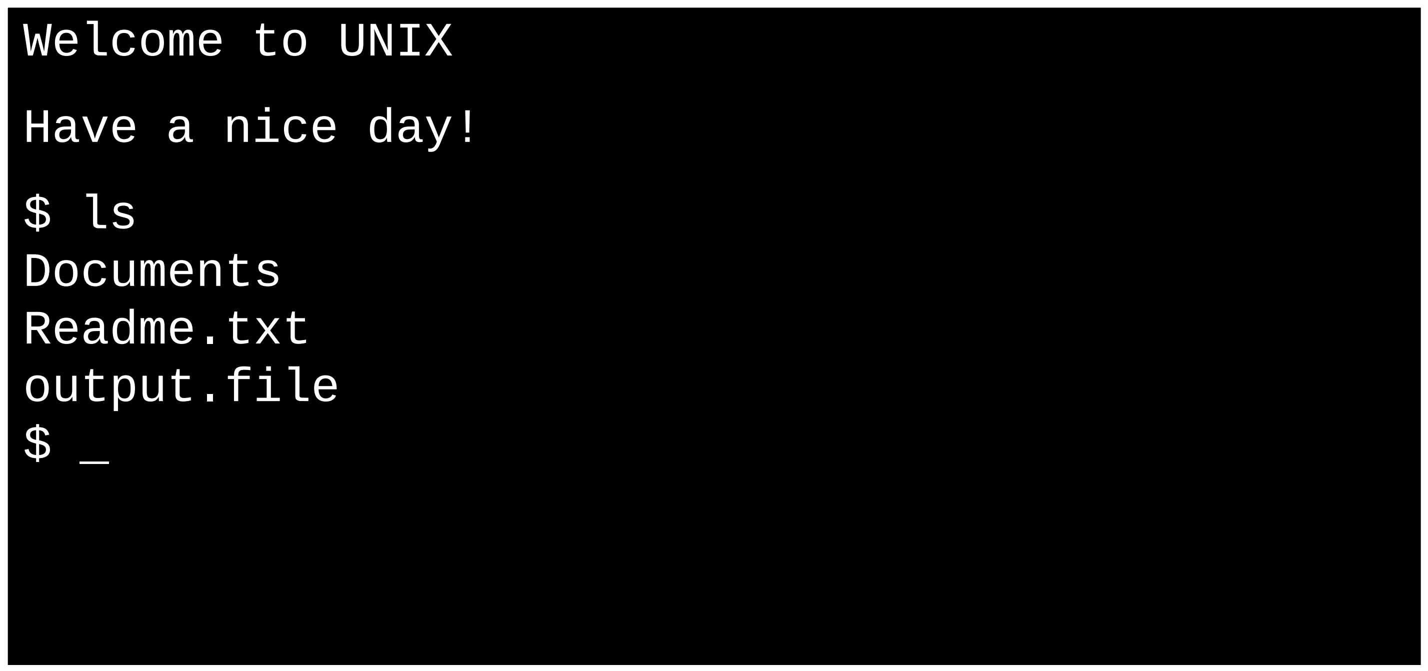
What is the UNIX Shell? (cont.)

- Another way to think of a shell is a layer of software between the operating system and the user (thus the term “shell”)



What is the UNIX Shell? (cont.)

- The UNIX shell is a text-based, command-line-driven program
- Each line of text that the user types is interpreted by the program (the shell) as one command (program) to run
- It looks like this:



```
Welcome to UNIX

Have a nice day!

$ ls
Documents
Readme.txt
output.file
$ _
```

Which Shell?

- There are a variety of (similar) UNIX shells to choose from
- The original and most widely supported shell is called the *Bourne shell* (after S.R. Bourne). These days it is considered the most basic of shells. Its program filename is ***sh***
- There are a number of Bourne shell derivatives, each offering a variety of extra features, including:
  - The Korn shell (after David Korn) (***ksh***) (not open/free)
  - The Bourne-again shell (***bash***) (open/free)
  - zsh
  - and many more

Which Shell? (cont.)

- It is possible to write scripts using the Bourne shell as an interpreter. Such scripts are known as *shell scripts*
  - The facilities that the Bourne shell makes available for this purpose allow the shell to be thought of as a programming language (a *scripting language*, to be precise)
  - Compare these with DOS *batch files*, interpreted by `command.com`
  - All Bourne-compatible shells share the same programming language (that's why they're called Bourne-compatible)
  - Much UNIX system administration and Web back-end processing (CGI) is done via Bourne shell scripts
  - The Virtual Training Company offers a training course on UNIX Shell Script Programming (Bourne-compatible shells)

Which Shell? (cont.)

- In an effort to offer more power to shell programmers, another shell was developed in which the programming language was more closely related to the powerful “C” programming language. This shell was called the *C-shell* (filename ***csh***)
- This in turn spawned its own family of compatible shells, most notably ***tcsh***
- The Bourne and C-shell families are completely incompatible from a programming standpoint, and they differ in their implementations of other certain features (such as command history), but otherwise they can be used interchangeably

Logging In

- Virtually all UNIX systems require users to log in (identify themselves to the computer with a username and password) before any programs can be run
- On some systems, as soon as a user has successfully logged in, a shell is automatically started for them to use to run programs
    - Note: at the end of this course we will look at how to choose which shell is automatically started for you
- Your username and password will have been created for you by the system administrator
- One username (root) has full system privileges, meaning they are never denied access to a resource

Logging In (cont.)

- On other systems (notably those with GUIs), once the user has logged in they have the option of starting any number of (concurrent) command-line shells by clicking the appropriate icon

Basic Commands

- In this module we will simply type in a few basic commands to get the feel for text-based commands
- For each of the following, type the name of the command, and then press the ***ENTER*** key (on some UNIX systems this key is labelled ***RETURN*** or ↵)

Command	Purpose
`uname`	Details about the machine you are logged into (including the version of UNIX). Try also: `uname -a`
`ps`	Find out which shell you're running
`cal`	Display a calendar for the current month Try also: `cal 2001 / cal 10 2001 / cal 9 1752`
`passwd`	Change your login password (follow the prompts and *be careful)*

Command Syntax

- The syntax (structure) of every command entered into the UNIX shell is as follows:

  ```
  $ prog-name [options] [arguments]
  ```

- Where:
  - *`prog-name`* is the name of the program you wish to run (for example, `ls`)
  - *`options`* are single letters prefixed by a “–” (dash, minus sign) that modify the behaviour of the program (for example, `-a`). These are always *optional*, meaning that the program will still do *something* if no options are specified
  - *`arguments`* are any other words (separated by spaces or tabs) that the program needs to perform the task you wish it to perform (such as a file or directory name). Some programs require a certain number of arguments, others (like `ls`) do not require any

Command Syntax (cont.)

- Notes:
  - The program name *must be* first. It is not possible to type in the name of a document and expect the shell to know how to open it (the UNIX shell has no concept of file “associations”). The program name must be the name of a binary program or an executable (text) script. Nothing else will run
  - Case is significant. In other words, `ls` is different from `LS`
  - It is possible to specify the location of the program as well (for example: `/bin/ls` `./myscript`)
  - There must be a space between every element of the command-line. Commands like `ls-la` or `cd/` will not run
  - The *`options`* may be specified in a variety of ways. All of the following are synonymous:

```
    ls -lax      ls -l -x -a      ls –ax -l
```

Getting Help

- Help text is available for every standard UNIX program
- This help is known as the “Manual Pages”, or *man* pages
- These are a reference tool, not a “how-to” guide
- They are used as follows: `man` ***`prog-name`***

  For example: `man cal`
- It is difficult to determine which program is best for a certain task. A program that offers partial help is

  `whatis`
- Many versions of the UNIX man pages can be found on the Internet

Logging Out

- There are two ways of exiting a (Bourne-compatible) shell:
  - Typing the `exit` command
  - Typing ***Ctrl-d***
    (in UNIX command-line programs, ***Ctrl-d*** always means:
    "*I have finished typing – I will be typing nothing further into this program*")
- Once the shell exits, UNIX automatically logs you out, and you are (typically) presented with a login screen again
- If you are using a GUI, exiting the shell will simply close the shell's window – the user will remain logged in

Chapter 4

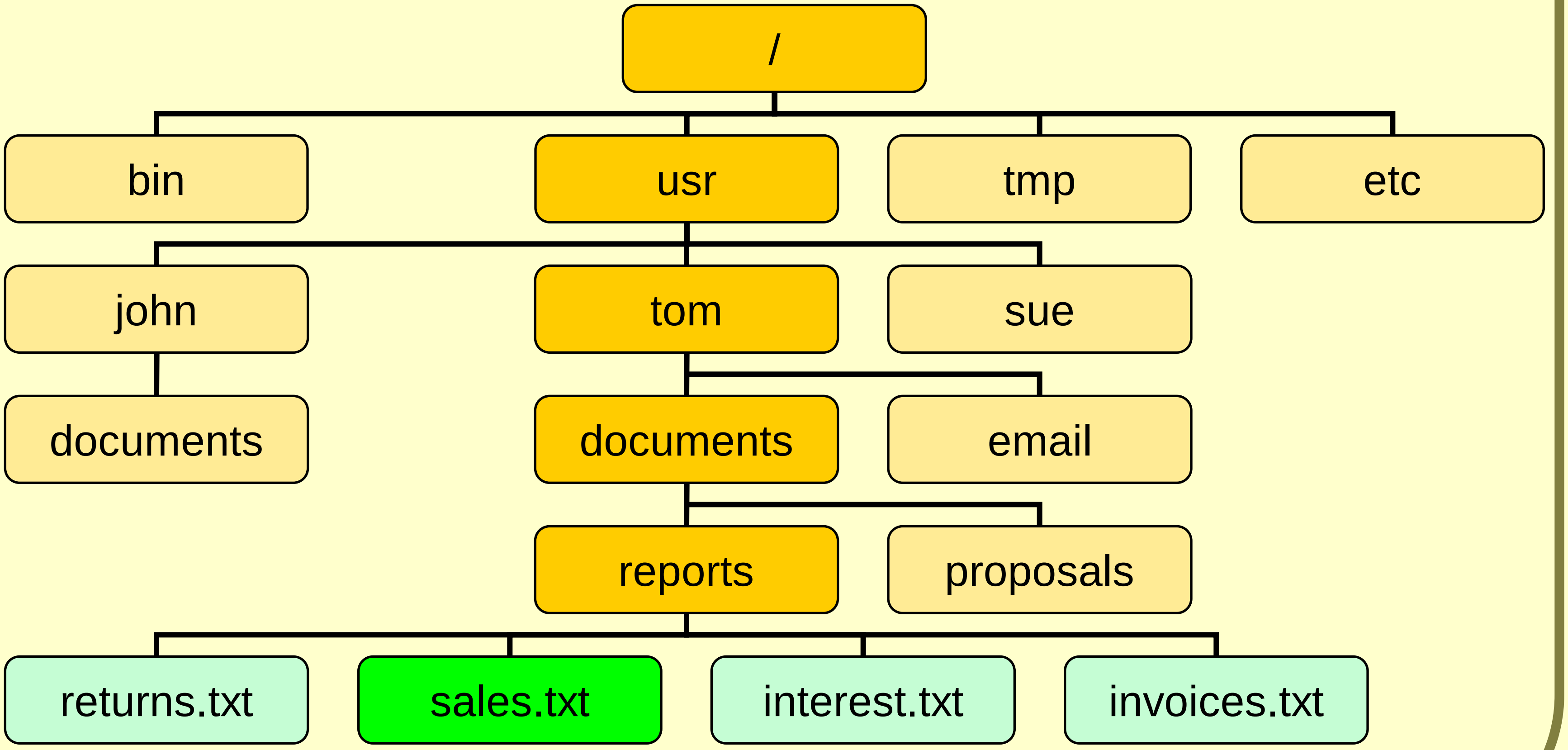
Files and Directories

Working with Files and Directories

- Similar to most operating systems, data in UNIX is stored in *files*. These files are organised hierarchically into *directories* (called *folders* in Windows)
- A useful analogy is to think of a UNIX filesystem as a tree (perhaps an upside-down one), with the directories being *branches*, and the files being the *leaves*
- The “top” of the tree is called the *root directory* (called “/”)
- Every file (and directory) on the tree is named by listing all the branches that lead back to the root, each separated by a “/”, as follows:

```
/usr/tom/documents/reports/sales.txt
```

Working with Files and Directories



Working with Files and Directories

- When working with files and directories, the first and most important information to be able to find out is: *What's available?*
- The program called `ls` is used to display file details (most importantly their *names*) and display the contents of directories
- Useful *options* for `ls` include:
  - `-l` display many file details, including size and security info
  - `-C` arrange list alphabetically in columns
  - `-r` (recursive) display contents of every sub-directory
- *Arguments* for `ls` (if there are any) are interpreted as names of either files or directories

Working with Files and Directories

- If we can identify a *directory*, we can use the program `cd` to “go there” (e.g. `cd Documents`)
- This means that if we want to manipulate files within that directory, we no longer have to prefix each filename with the directory’s name
- The command `cd` with no arguments will “take us” to our “home” directory
- To determine where you are at any time, use the pwd command

Filenames and File Types

- The following characters may be used in a filename with no problems:

```
 a-z  A-Z  0-9  .  ,  @  -  _  +  =  :
```

- The following characters *should* be avoided when naming files, because they have special use with most shells:

```
   space  ~  `  !  #  $  %  ^  &  *  ?
   (  )  '  "  [  ]  {  }  ;  <  >  \  |
```

- There is only one character that is not possible to use within a filename:

```
                       /
```

Filenames and File Types (cont.)

- Unlike Windows, UNIX has no concept of a file *type*
- Files do not have *associations* – at least, not using a command-line shell. This means that it is not possible to type in the name of a document and expect the file to be opened in the appropriate program
- Nor do UNIX filenames have any formal *extension* – a period (‘`.`’) can be placed anywhere within a filename, even at the beginning
- Some programs will work more readily with filenames that have certain extensions
- The program `file` is used to make a guess as to a file’s contents

Wildcards

- When using the shell (and many other programs), it is possible to specify *groups* of files in a simple manner, by using “wildcard” characters
- The shell’s wildcard characters are:
  - `*` match any characters in the filename(s)
    - For example: `*.txt   mark.*   a*b   *.*`
  - `?` match any *single* character
    - For example: `c?t   fred.????   ????x?`
  - `[]` match any *single* character that appears within the brackets
    - For example: `c[aou]t   [a-zA-Z]*.txt   [!d]*`
- What does the following match?

```
ls ?[!.]*.[ch]
```

Displaying File Contents

- If a file contains nothing but text (for example, an HTML file), the file can be displayed on the screen using a number of programs:
  - `cat` The simplest program for displaying file contents on the screen
  - `more` Allows output to be displayed a page (or a line) at a time
  - `head` Display only the first 10 lines of the file(s)
  - `tail` Display only the last 10 lines of the file(s)

Comparing files

- Files may be compared to check how similar their contents are
- There are two programs for this, depending upon what is in the file:
  1. `diff` Used to compare two *text* files. A complete list of differences is output to the screen
  2. `cmp` Used to compare two *binary* (non-text) files. The only output is a simple statement about the character location of the first difference between the files (if there is one)

Copying, Moving and Renaming Files

- Files may be duplicated (copied) using the `cp` command
- `cp` is used in two ways:

1. `cp file1 file2`
   Create a duplicate of `file1` called `file2` in the current directory. If `file2` exists it is overwritten (if you have permission)
2. `cp file1 [… fileN] dir1`
   All the files `file1` … `fileN` are copied to the specified directory `dir1`

Copying, Moving and Renaming Files

- Similarly, files may be moved and renamed using the `mv` command
- `mv` is used in two ways:

1. `mv file1 file2`
   Simply rename `file1` as `file2` in the current directory. If `file2` exists it is overwritten (if you have permission)
2. `mv file1 […fileN] dir1`
   All the files `file1` … `fileN` are moved to the specified directory `dir1`

Deleting Files

- The program called `rm` is used to remove (delete) files
- If a file is marked as “read-only” (more about this later), you will be asked for a `y/n` confirmation
- This program has several options:
  - `-i` (interactive) Ask for a `y/n` confirmation before deleting each file
  - `-f` (force) Do not ask for *any* confirmation, and display no error message if a file does not exist
  - `-r` (recursive) Use with extreme caution!! If any of the arguments is a directory, remove the directory and all its contents (including subdirectories)

Hidden Files

- All files whose names begin with a “`.`” are considered “hidden” files
- This means they are not displayed during a regular `ls` of that directory, and are not matched by wildcards (`*` or `?`). For example, they are not removed by the command `rm *`
- The `-a` option can be used with `ls` to list hidden files
- These files are not special in any particular sense. Any file can be renamed so that it starts with a “`.`”
- Hidden files are usually configuration files, such as `.profile` or `.exrc`

The "." and ".." Directories

- Two useful aliases are automatically created in each directory:
  - "`.`" This is a shorthand for "the current directory"

    For example: `cp /tmp/*.doc .`
  - "`..`" This is a shorthand for "the parent" directory"

    For example: `mv *.txt ..`
- These are treated as "hidden"

Relative vs Absolute Paths

- Consider the following file:

  `/usr/tom/business/reports/june/sales.txt`

- There are a number of ways of accessing this file, depending upon your “current directory”:
  - From `/usr/tom`

    `business/reports/june/sales.txt`
  - From `/usr/tom/business/reports/june`

    `sales.txt`
  - From `/`

    `usr/tom/business/reports/june/sales.txt`
  - From `/usr/tom/business/reports/june/drafts`

    `../sales.txt`

Relative vs Absolute Paths (cont.)

- If we had no concept of a “current directory,” we would always have to use the full filename (path) for any file (known as an *absolute path*)
- The notion of a “current directory” was created to allow shorter, simpler specification of filenames
- A filename that is specified relative to your “current directory” is known as a *relative path*
- Any filename (path) that begins with a “/” is an absolute path. All others are relative paths
- All UNIX programs can handle relative or absolute paths, or a mixture of both

Working with Directories

- Directories can be created using the `mkdir` program
    - For example: `mkdir newdir`
- Directories can be removed (deleted) using the `rmdir` program
    - For example: `rmdir newdir`
    - Directories can only be deleted using `rmdir` if they are empty (contain no files – not even hidden files)
    - To remove a directory that is not empty, use `rm -r` (*Note: this will remove all files and subdirectories as well*)
- Directories can be renamed and moved using `mv` in the usual manner

Finding Files

- The `find` program can be used to locate files (amongst other things). It’s usage is:

```
find top-directory [criteria and actions]
```

- For example:

```
find /usr/tom -name report.txt -print
```

- Notes:
  - The `-print` option is the default action in some implementations of find, and is thence not necessary
  - When using wildcards with the -name criterion, enclose the argument in single-quotes (`'`), as follows:

    ```
    find . -name '*.txt' -print
    ```

Finding Files (cont.)

- The `find` program can also be used
  - to find files based on other criteria
  - to perform actions on the files found (other than display their names)
- For example:

```
find . -type f -exec rm -i {} \;
```

- Files can be found on the basis of any file attribute

Archiving Files

- Consider the following tasks you may wish to perform on a collection of files and subdirectories:
  - Move them to another location on the hard disk
  - Back them up onto removable media (such as tape)
  - Send them electronically to another machine (perhaps over the Internet)
- For these purposes it is necessary to create an *archive* – a single file (or image on tape) that contains all the information about the files and directories (including the file names, ownerships, modification dates and contents)

Archiving Files (cont.)

- There are several programs available in UNIX that can archive files:

Command	Can write to offline media?	Provides compression?
`tar`	Y	Y
`cpio`	Y	N
`gzip`	N	Y
`compress`	n/a	Y

Chapter 5

Security

Users and Groups

- UNIX is a "multi-user" operating system
- This means:
  - More than one user may interact with (log on to) the system at any given moment
  - Each user has a separate set of access privileges for system resources (such as files)

Operating System	Simultaneous users?	Separate privileges?
UNIX	Y	Y
Windows 2000/XP	Y	Y
Windows NT	N	Y
Windows 95/98	N	N

Users and Groups (cont.)

- A *user* is a system ID that allows each user that logs in to identify themselves for resource-access purposes
- When a collection of users require similar access to a resource, each user in the collection may be made a member of a *group*, and the group given access to the resource
- The following table shows security-related commands:

Command	Purpose
`who am i`	Display login information, including user name
`id`	Display current user and group information
`su` *`username`*	Temporarily run a shell as another user
`newgrp` *`groupname`*	Switch current group

File Protection Overview

- The following security-related information is stored against each file/directory in a UNIX filesystem:
  - The *owner* of the file (a user name)
  - The *group* that the file belongs to (a group name)
  - The *permissions* that various parties have to access the file
- There are three sets of permissions:
  1. The access privileges of the *owner* of the file
  2. The access privileges of any members of the file's *group*
  3. The access privileges of *everyone else*
- Only the file’s *owner* is allowed to change any of these
- All the above details may be examined using `ls -l`

File Protection Overview (cont.)

- The three permission letters (`r`, `w`, and `x`) mean the following:

Permission	For *Files*	For *Directories*
`r`	read	view contents (e.g. `ls`)
`w`	modify (write)	create or delete files
`x`	execute	access (e.g. cd)

Changing File Permissions

- The program used to change file permissions is `chmod` (short for **ch**ange **mod**e)
- There are two distinct methods of using `chmod`:
  1. Symbolic mode
  2. Numeric mode
- Both methods have the following usage:

```
chmod permissions filename(s)
```

Only the *`permissions`* differs between methods

- *Symbolic mode* is used mainly by beginners. Once a user is comfortable with `chmod`, they almost always use *Numeric mode*

chmod Symbolic Mode

- In symbolic mode, permissions are specified by using letters, as follows:
  - `chmod u+w file1` Give the *owner write* permission
  - `chmod g-r file1` Remove *read* permission for the *group*
- The most common letters are:

Persons		Action		Permissions	
u	owner	+	add	r	read
g	group	-	remove	w	write
o	other	=	set	x	execute
a	all				

chmod Symbolic Mode (cont.)

- Multiple permissions may be specified as follows:
  - `chmod uo+w,u-rx file1`

    Give the *owner* and *others write* permission, and remove *read* and *execute* permission for the *user*
- As you can see, to set all 9 permissions could be time-consuming

chmod Numeric Mode

- In numeric mode, permissions are specified by using three numbers
- All permissions are specified in one command
- If we allow that `r`=4, `w`=2 and `x`=1, we can set permissions for each of the three types of persons (owner, group & others) by adding together the numbers corresponding to the permission we want
- For example:
  - `chmod 640 file1            rw-r-----`
    Give the *owner read* and *write* permission, the group read permission, and no permissions to anyone else
  - `chmod 070 file1            ---rwx---`
    Give full permissions to the *group* only

chmod Numeric Mode (cont.)

- While there are hundreds of legal combinations, only a few are used commonly:

	chmod	ls
Data files	`444`	`r--r--r--`
	`644`	`rw-r--r--`
	`664`	`rw-rw-r--`
	`666`	`rw-rw-rw-`
Programs	`750`	`rwxr-x---`
	`755`	`rwxr-xr-x`
	`777`	`rwxrwxrwx`
Directories	`755`	`rwxr-xr-x`
	`775`	`rwxrwxr-x`
	`777`	`rwxrwxrwx`

Changing File Ownership

- Changing the owner of a file is done using the `chown` program, as follows:

  ```
  chown owner filename(s)
  ```

- For example:

  ```
  chown fred *.doc
  ```

- Notes:
  - Ownership of a file has *nothing to do* with the location of the file. In other words, a file may be owned by `fred` but sitting in `tom`'s home directory
  - You cannot change *any* permissions on a file that you don't own, so if you're changing many things, change ownership *last*
  - If you change the ownership of a file, you cannot change it back

Changing File Group

- Changing the group ownership of a file is done using the `chgrp` program, as follows:

```
chgrp group filename(s)
```

- For example:

```
chgrp marketing data*
```

- Notes:
  - If you are a member of a group, and that group has permission over a file, and when you try to exercise that permission you get a "permission denied" error, use the `newgrp` command to switch your current group

A Dangerous Security Loophole

- Consider the following permission set:

```
drwxrwxrwx ... dir1
-r--r--r-- ... dir1/file1
```

- It is possible to *modify* `file1` by doing the following:
  1. `cp file1 file2`
  2. Modify `file2` (this is possible because all files created by you are able to be modified by you by default)
  3. `rm file1` (this is possible because `dir1` is writable)
  4. `mv file2 file1`
  5. Change any necessary ownership/permissions/etc to make `file2` look more like the original `file1`
- Moral: Always pay attention to *directory* permissions

Chapter 6

Combining Programs
- Pipes and Filters

Standard Output

- Most running UNIX programs produce output
  - Such output usually ends up on the screen (the user's monitor)
- This output can be "redirected" to one of two other "places":
  1. A file
  2. Another program
- To redirect output to a *file*, the ">" symbol is used
- For example:

```
ls -l > listing
```

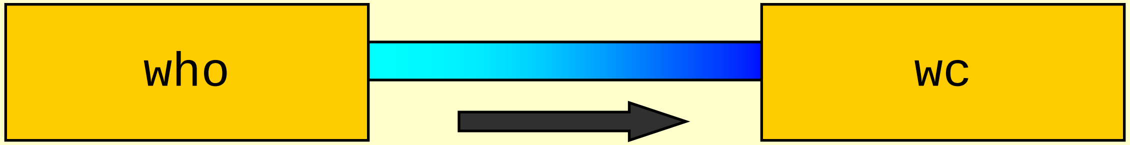
A file called `listing` is created in the current directory that contains the output of the `ls` program

Standard Output (cont.)

- To redirect output to another program, the "|" ("pipe") symbol is used
- For example:

```
who | wc
```

- When this command is typed on the command line, the shell does the following:
  - Starts the who program
  - Starts the wc program
  - Connects the two in such a way that the output of the first program is "piped" to the second, where it is used as input



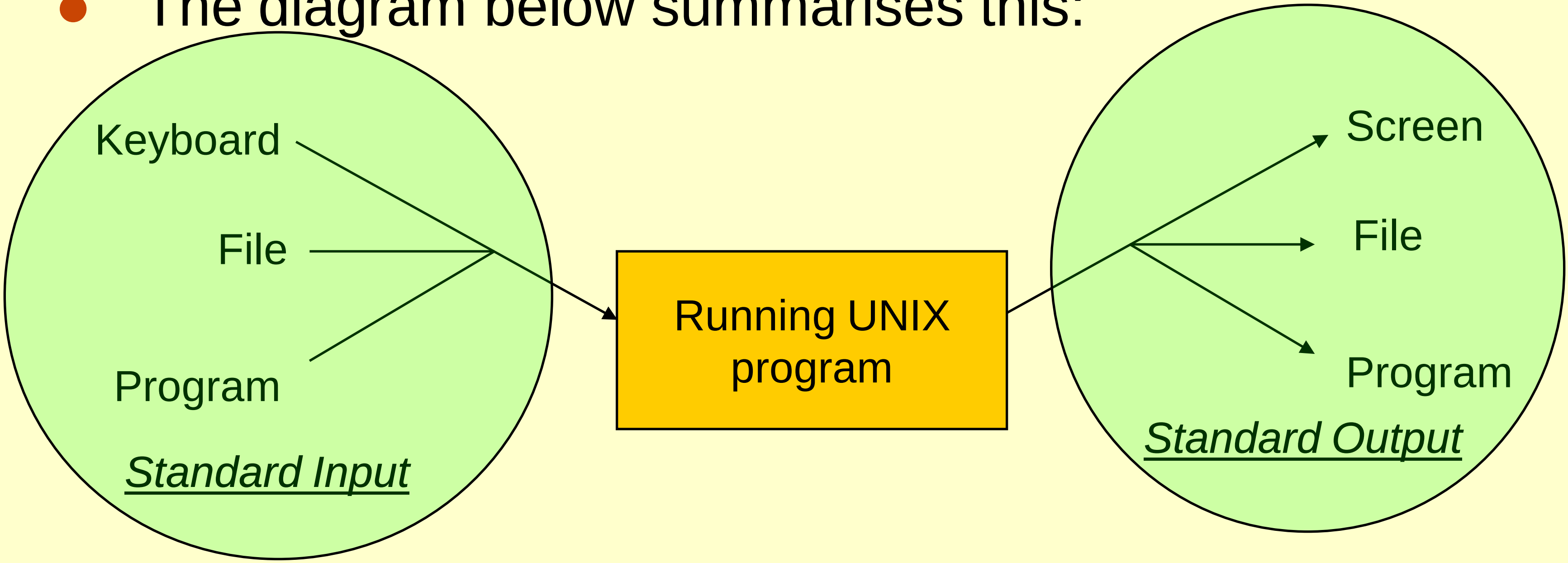
Standard Input

- The previous diagram implies (correctly) that programs can take *input*
- Not many programs take input
- By default, a program's input comes from the *keyboard*
- An appropriate use of the shell can cause a program to take input from one of two other places:
  1. A file
  2. Another program
- Input is read from a file by using the "<" symbol
- For example:

```
wc < file
```

Standard Input and Output

- The input and output described above are known as *Standard Input* and *Standard Output*
- The diagram below summarises this:



Standard Input and Output (cont.)

- Why would we want to do this? Why would we want to connect programs together, or read data from files?
- Most of the command-line utilities that come with UNIX adhere to the philosophy that "complex" tasks may be performed by combining simple programs
- This "roll-your-own" approach has made UNIX very successful, and is the foundation of shell-scripting

About Filters

- A *filter* is a UNIX command-line utility that has the following properties:
  1. It takes standard input
  2. It performs some processing on the data it reads
  3. It produces output based upon that input
- For example, `wc` is a filter. The processing it performs is counting lines, words and characters. `ls`, however, is *not* a filter (it takes no input)
- Filters are used to process the data produced by other programs and the data in files

Common Filters

- The following programs are filters that are used regularly in UNIX:

Filter	Processing done to Standard Input
`cat`	None
`more`	Pagination
`grep`	Removal of lines that do not contain certain text
`sort`	Sorting
`wc`	Counting of lines, words and/or characters
`tee`	Duplication – write to files *and* screen
`sed`	Basic editing
`awk`	Anything

Searching for Text in Files

- In the previous table, it was said that `grep`'s processing is "removal of lines that do not contain certain text"
- This can be put more meaningfully in two other ways:
  - `grep` is used to search for text in files (or Standard Input)
  - `grep` is a true *filter*, in every sense of the word
- grep has the following usage:

```
grep pattern filename(s)
```

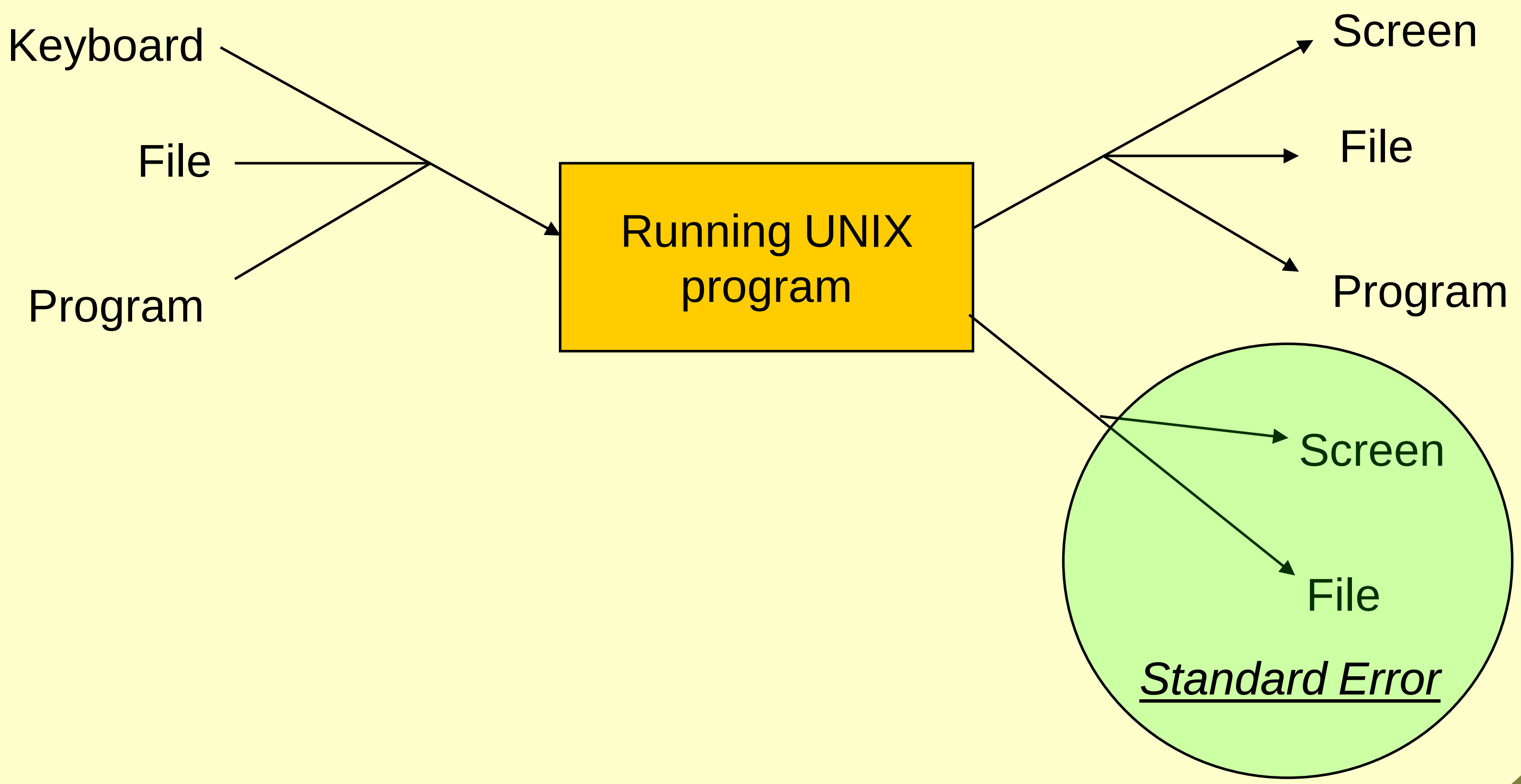
- For example:

```
grep Mark *.txt
```

- `grep` uses its own set of wildcards
- Use `grep` with `find` to locate files in all directories

Standard Error

- Each program actually produces *two* sets of output:



Standard Error (cont.)

- To redirect Standard Error to a file, use "2>"
  - (Standard Output can also be redirected using "1>")
- This means that a command's output can be redirected to two separate files, if need be, as follows:

  ```
  command1 > fileA 2> fileB
  ```
- It is possible to redirect all output to the same place, as follows:

  ```
  command1 > fileA 2>&1
  ```
- Any form of output may be redirected to `/dev/null` if it is not wanted at all

Chapter 7

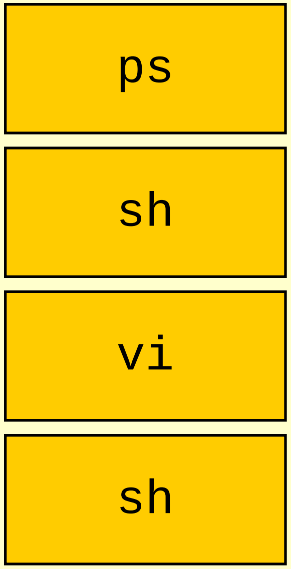
Process Control

About Processes

- A *process* is a running program
- Every process is assigned a unique *process ID* (usually in the range 1-30000)
- Running processes for the current login session can be examined by using the `ps` command
- `ps` also has the following options:
  - `-u user` Display all processes for the given user
  - `-f` or `-l` Display listings with more details
  - `-e` Display all processes on the system

About Processes (cont.)

- It is sometimes useful to think of a process *stack*:



ps

sh

vi

sh

Running Commands Asynchronously

- If a process is run *asynchronously* ("in the background"), the shell does not wait for the process to finish before presenting another prompt and allowing you to start further processes
- To run a process asynchronously, append a "&" to the command line
- Notes:
  - Commands that take *Standard Input* should never be run in the background
  - Commands that produce *Standard Output* (or *Error*) *can* be run in the background, but it is a good idea to redirect their output to a file (or `/dev/null`)
  - Asynchronous processes are terminated when you log out

Killing Processes

- Any running process may be stopped by using the `kill` program, as follows:

  ```
  kill process-ID
  ```

- For example:

  ```
  kill 12548
  ```

- Notes:
  - Some processes can leave the system in an unexpected state when they are killed – try to avoid using `kill` if possible
  - kill will only succeed if you have permission to kill a process
  - Some processes are programmed to ignore attempts to kill them. If you really want the program to end, and you've already tried `kill`, then try:

    ```
    kill -9 process-ID
    ```

Jobs

- If you are using `ksh` or `bash`, it is possible to suspend (or *stop*) running processes, in order that further processes may be started. A stopped process is called a *job*
- While a process is running, use `Ctrl-z` to stop it
- The command `fg` will resume the most recently stopped program
- The command `jobs` is used to see a list of all stopped jobs. Each is prefixed by a number that can be used by `fg` to resume that particular process
- Note that programs running asynchronously cannot be stopped

More Process Control

- All running processes are assigned a *priority* by the operating system. This priority determines what share of the CPU time the process is given
- It is possible to adjust this priority using the `nice` command
- A higher priority will result in the process completing sooner, at the expense of other processes. A lower priority will prevent your process from impinging too much upon the speed of other processes
- Priorities are numbers in the range `-20` (highest) to `19` (lowest). The default priority is `10`

More Process Control (cont.)

- `nice` is used as follows:

```
nice -priority command [arguments]
```

- For example:

```
nice --20 find / -name output.file
```

or

```
nice -19 backup /home
```

- Note that on most UNIX's, no user except `root` is permitted to increase a process's priority

More Process Control (cont.)

- Recall that processes running asynchronously are terminated when the user logs out
- It is possible to prevent this, by using the nohup command (short for "no hang-up"), as follows:

```
nohup command [arguments] &
```

- For example

```
nohup backup /home &
```

- Any *Standard Output* the process produces will be written to a file called `nohup.out`, unless it is redirected elsewhere

Scheduling Commands

- It is possible to schedule commands to be run at a particular time in the future (usually within the next 24 hours)
- The command `at` is used to schedule commands, as follows:

```
$ at 2315
> cd /home/fred
> tar cf /dev/fd0 *
> rm -rf *
> Ctrl-d
$
```

- Any *Standard Output* produced is emailed to the user

Chapter 8

`vi` – A UNIX Text Editor

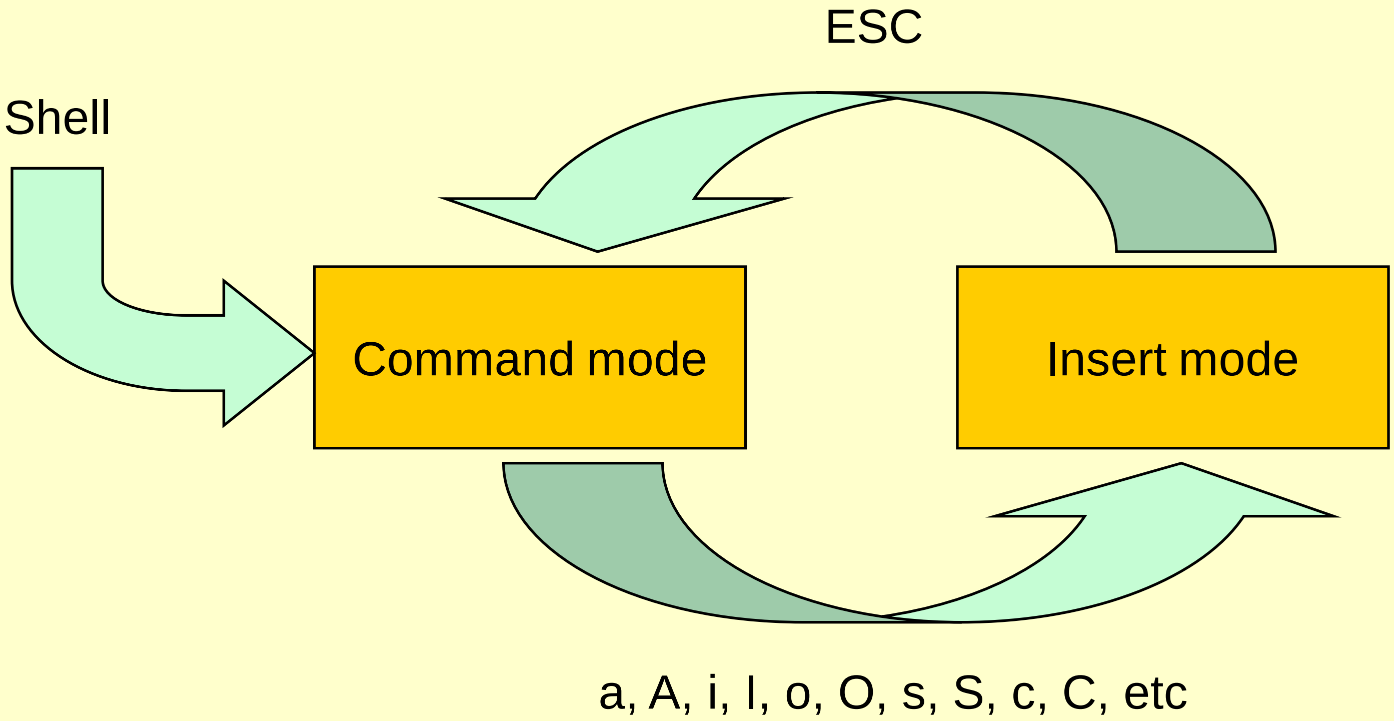
Understanding `vi`

- `vi` (short for **vi**sual editor) is a UNIX text editor – a full-screen program used to edit text files, including:
  - HTML documents
  - Shell scripts
  - Configuration files
  - Source code (C, C++, etc)
- It evolved from the line-based editing programs `ed` and `ex`, and shares many of the same editing commands
- `vi` is notoriously unintuitive and difficult to learn, but conversely, once learnt, it is one of the most powerful, and feature-rich editors in the world (on *any* platform)

Understanding `vi` (cont.)

- Owing to the limitations of primitive early keyboards, and unlike most other text editors, when using `vi`, you will always find yourself in one of two *modes*:
  1. *Command mode*, where each key typed represents an editing command
  2. *Insert mode*, where each key typed (except ESC) represents text that you wish to insert into the document
- The diagram on the next page summarises this

Understanding `vi` (cont.)



ESC

Shell

Command mode

Insert mode

a, A, i, I, o, O, s, S, c, C, etc

Manipulating Files

- Like most text editors and word processors, `vi` can be started with a document to edit, or without – as an "empty canvass", as follows:
- To use `vi` to edit a file:

```
vi filename(s)
```

- For example:

```
vi script1
vi *.txt
```

- Or to start `vi` with no file:

```
vi
```

Manipulating Files (cont.)

- Once `vi` has been started, you will find yourself in *Command Mode*
- To perform some simple editing:
    - Use (for example) a to "append" text after the current cursor position (and enter *Insert Mode*)
    - Enter some text, for example:

      ```
      The quick brown fox
      Jumps over the Lazy Dog
      ```
    - Press ESC to return to *Command Mode*
- We will now look at the many ways in which we may quit `vi` and save the changes

Manipulating Files (cont.)

:w	write (save) the file (only if a name has been specified)
:w *file*	write to the specified file (save as)
:q	quit (only if no changes have been made)
:wq	save and then quit
:x or ZZ	save the file (if changes have been made), then quit
:q!	abandon any changes and quit
:w!	write to a read-only file (that you own)
:e *file*	open the specified file (if no changes have been made)
:e! *file*	open the specified file (abandon any changes)
:e# or ^6	open the last file edited
:n and :n!	open the next file specified on the command line
:rew	rewind to the first file specified on the command line
:f or ^g	display current file details

Moving Around

- PC keyboard special keys (arrow keys, `Page Up`, `End`, etc) *sometimes* work in `vi`
- Any command listed below with a ☑ can be prefixed with a number ***n*** to move ***n*** intervals
- Moving on the current line

`SPACE` or `l`	☑ Move ahead one character
`BACKSPACE` or h	☑ Move back one character
`$`	Move to the last character on the line
`^` or `0`	Move to the first character on the line
`f`***`X`***	☑ Move to (**f**ind) the next instance of character ***X***
`;`	☑ Move to the *next* instance of character ***X***

Moving Around (cont.)

- Moving between lines
  - `ENTER` or `j` or `+` ☑ Move to the next line
  - `k` or `-` ☑ Move to the previous line
  - `^f` ☑ Move **f**orward one page (page down)
  - `^b` ☑ Move **b**ack one page (page up)
  - `^d` ☑ Move **d**own half a page
  - `^u` ☑ Move **u**p half a page
  - `G` **G**o to last line in file
  - `1G` **G**o to first line in file
  - ***n***`G` **G**o to line ***n*** in file

Moving Around (cont.)

- Other move commands
  - `/`***`pattern`*** Move to the next occurrence of ***`pattern`***
  - n (N) ☑ Move to the **n**ext (previous) occurrence
  - `w` (b) ☑ Move forward (back) one **w**ord
  - `%` Find the matching bracket: `( ) [ ]` and `{ }`
  - `]]` (`[[`) ☑ Move to the next (previous) C function
  - `m`***`x`*** **M**ark a line with label ***x***
  - `'`***`x`*** Go to line labelled ***x***

Basic Editing

- The following commands take you into *Insert Mode*, where text is then entered. Press ESC to return to *Command Mode*:
  - a (i) **A**ppend (**i**nsert) text after (before) the current cursor
  - A (I) **A**ppend text to the end (beginning) of the line
  - o (O) Start (**o**pen) a new line after (before) the current line
  - s ☑ **S**ubstitute the current character with text
  - cw ☑ **C**hange the remainder of the word to new text

Basic Editing (cont.)

- The following commands are also used for basic editing, but do not take you to *Insert Mode*:
  - x (X) ☑ Delete (cut) the char under (before) the cursor
  - dd ☑ **D**elete (cut) the current line
  - p ☑ **P**ut (paste) the recently deleted text
  - yy ☑ **Y**ank (copy) the current line
  - `r`***x*** ☑ **R**eplace the current character with ***x***
- The following commands special commands are very useful:
  - u **U**ndo the last command
  - U Restore the current line to how it was when you arrived on it
  - **.** ☑ Repeat the last command

Advanced Editing

- The following commands are simply useful:
  - `>>` (`<<`) ☑ Shift the current line to the right (left)
  - `J` **J**oin the next line to the current line
  - `^l` Redraw the screen
  - `:`***`m`***`,`***`n`***`s/`***`abc`***`/`***`xyz`***`/g`
    Substitute all occurrences of ***abc*** with ***xyz*** on
    lines ***m*** to ***n*** (using `grep`-like regular expressions)
    (use `1,$` for all lines in file, `.` for current line)
  - `:!command` Run a shell command (e.g. `ls`)
    (use % for current filename, # for alternate file)
  - `:sh` Obtain a temporary shell

Configuring `vi`

- It is possible to specify options that modify the general behaviour of `vi`
- Use `:set` to list currently set options (or `:set all` to list *every* option)
- "On/off" options (for example, `ai`) can be specified as follows:

`:set ai`	(turn option on)
`:set noai`	(turn option off)

- "Value" options (for example, `ts`) can be specified as follows:

```
:set ts=4
```

Configuring `vi` (cont.)

- Useful options include:

Option	Name	Type	Purpose
`ai`	autoindent	on/off	Cause new lines to inherit the indentation of the previous line
`ic`	ignorecase	on/off	Searches will be case-insensitive
`nu`	numbers	on/off	display line numbers
`sw`	shiftwidth	value	The number of spaces the shift with >> and <<
`ts`	tabstop	value	The number of spaces to use when displaying TAB characters

Configuring `vi` (cont.)

- If you would prefer these options to be in effect every time you start `vi`, put them into a file in your home directory called `.exrc`
- Ensure every line starts with `set` …

Chapter 9

The UNIX File System

How Files are Stored

- What does it mean to say that a file is "in" a directory?
- Everything on a UNIX filesystem (hard disk) is a file, even directories
- A UNIX directory "file" simply contains a list of filenames and *i-node numbers*

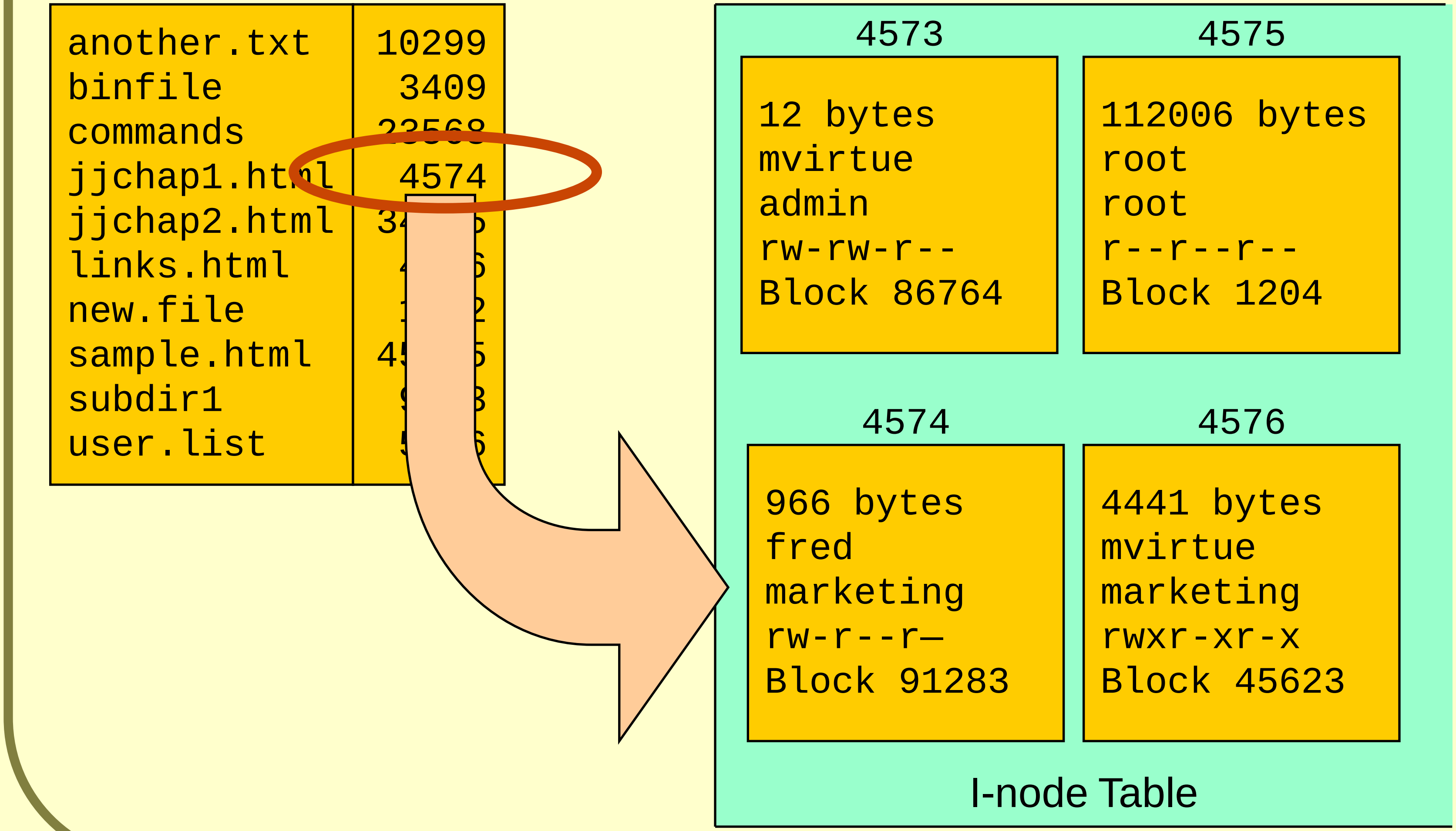
Directory
"File"

another.txt	10299
binfile	3409
commands	23568
jjchap1.html	4574
jjchap2.html	34785
links.html	4366
new.file	1002
sample.html	45845
subdir1	9933
user.list	5646

How Files are Stored (cont.)

- An *i-node number* is simply a reference to an i-node in the i-node table (c.f. FAT in DOS/Windows)
- An *i-node* ("information node") is a set of details about a single file on disk, including:
  - File size
  - Creation, access and modification times
  - Owner and group
  - Permissions
  - File type (file, directory, device, etc)
  - Link count (typically 1)
  - Starting block number
- Use `ls -i` to display i-node numbers for any file

How Files are Stored (cont.)



another.txt	10299
binfile	3409
commands	23568
jjchap1.html	4574
jjchap2.html	34[illegible]5
links.html	4[illegible]6
new.file	1[illegible]2
sample.html	45[illegible]5
subdir1	9[illegible]3
user.list	5[illegible]6

4573

12 bytes
mvirtue
admin
rw-rw-r--
Block 86764

4575

112006 bytes
root
root
r--r--r--
Block 1204

4574

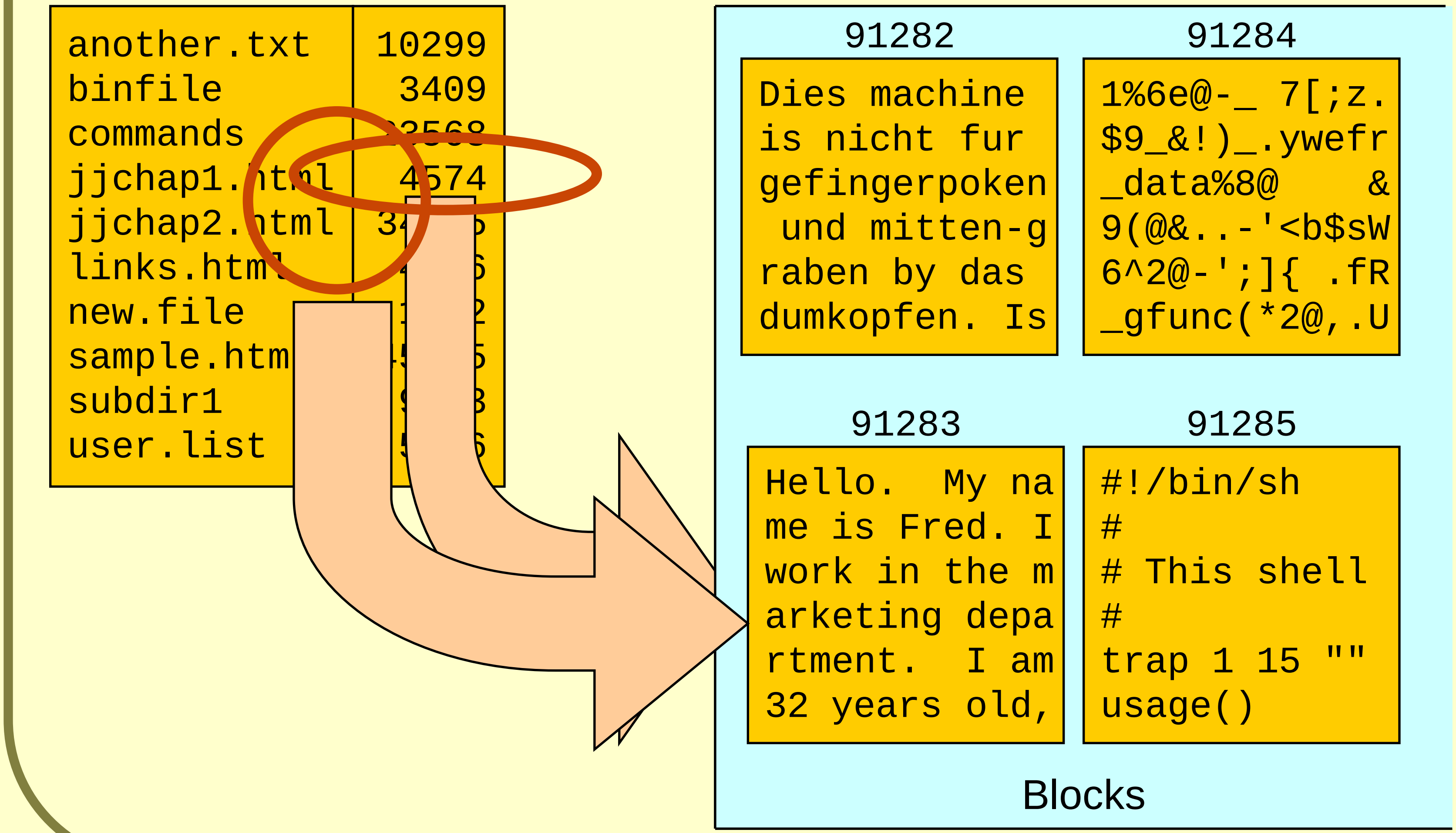
966 bytes
fred
marketing
rw-r--r—
Block 91283

4576

4441 bytes
mvirtue
marketing
rwxr-xr-x
Block 45623

I-node Table

How Files are Stored (cont.)



another.txt	10299
binfile	3409
commands	23568
jjchap1.html	4574
jjchap2.html	[illegible]
links.html	[illegible]
new.file	[illegible]
sample.html	[illegible]
subdir1	[illegible]
user.list	[illegible]

91282

```
Dies machine
is nicht fur
gefingerpoken
 und mitten-g
raben by das
dumkopfen. Is
```

91284

```
1%6e@-_ 7[;z.
$9_&!)_.ywefr
_data%8@    &
9(@&..-'<b$sW
6^2@-';]{ .fR
_gfunc(*2@,.U
```

91283

```
Hello.  My na
me is Fred. I
work in the m
arketing depa
rtment.  I am
32 years old,
```

91285

```
#!/bin/sh
#
# This shell
#
trap 1 15 ""
usage()
```

Blocks

Understanding Links

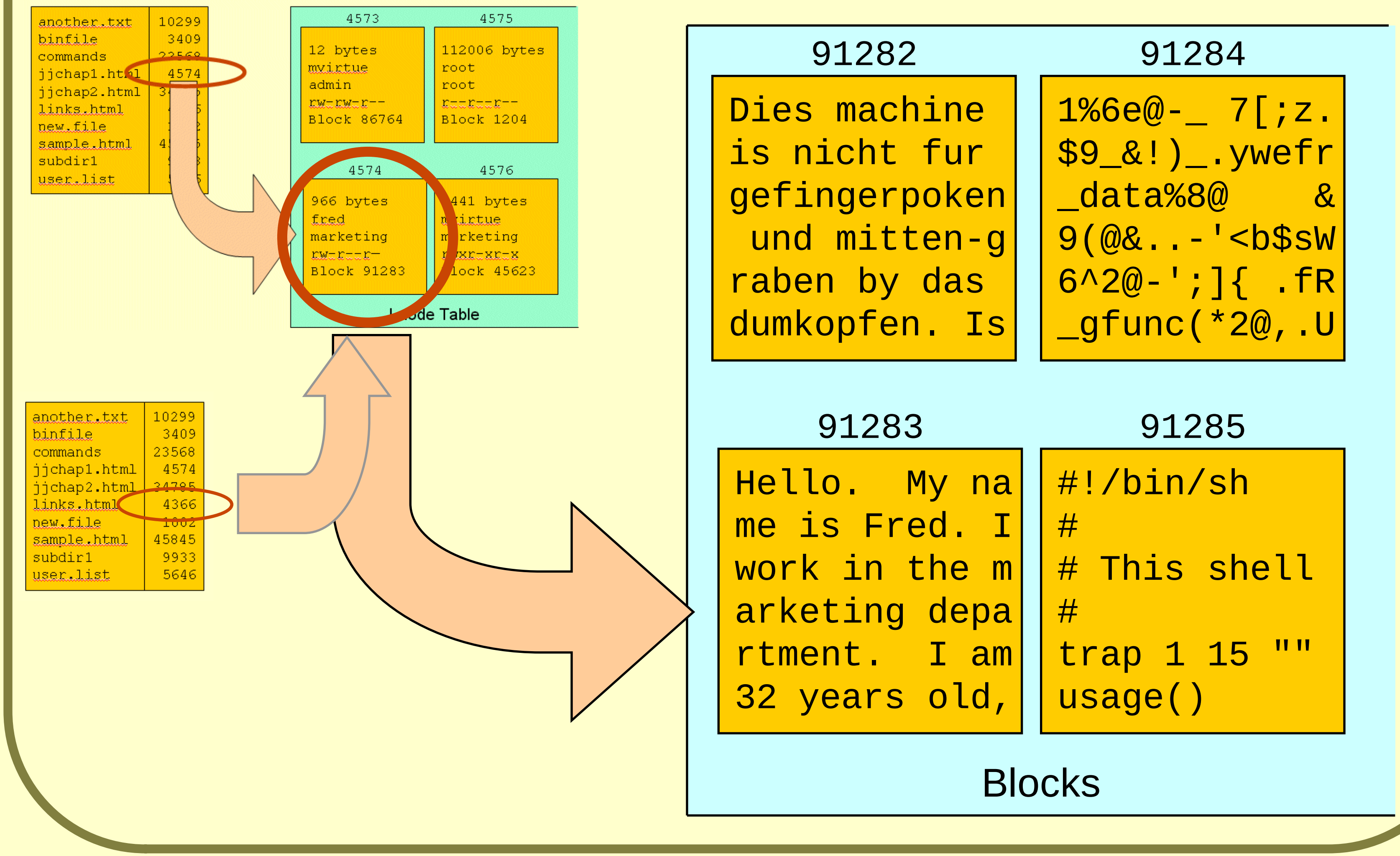
- In UNIX, unlike some other operating systems, it is possible for a file to have more than one name, in more than one directory
- Each name is called a *link* (including the original name)
- All links to a file share the same i-node, meaning that every link shares the same permissions, etc
- No link is considered "more important" than any other link. In particular, if there is more than one link, there is no concept of an "original" and "duplicates"

Understanding Links (cont.)

- The `rm` program simply "unlinks" the link that you specify
- When all links to a file have been unlinked, the data that the file contained is "deleted" (marked as available)
- `ls -l` can be used to examine how many links any given file has (including the one in the listing)
- There is no simple program that can show all links to a given file. Instead, use `find` and search for an i-node number, as follows:

```
find dir –inum 12345
```

Understanding Links (cont.)



another.txt	10299
binfile	3409
commands	23568
jjchap1.html	4574
jjchap2.html	[illegible]
links.html	[illegible]
new.file	[illegible]
sample.html	[illegible]
subdir1	[illegible]
user.list	[illegible]

4573

12 bytes
mvirtue
admin
rw-rw-r--
Block 86764

4575

112006 bytes
root
root
r--r--r--
Block 1204

4574

966 bytes
fred
marketing
rw-r--r–
Block 91283

4576

441 bytes
mvirtue
marketing
rwxr-xr-x
Block 45623

Inode Table

another.txt	10299
binfile	3409
commands	23568
jjchap1.html	4574
jjchap2.html	34785
links.html	4366
new.file	1002
sample.html	45845
subdir1	9933
user.list	5646

91282

```
Dies machine
is nicht fur
gefingerpoken
 und mitten-g
raben by das
dumkopfen. Is
```

91284

```
1%6e@-_ 7[;z.
$9_&!)_.ywefr
_data%8@     &
9(@&..-'<b$sW
6^2@-';]{ .fR
_gfunc(*2@,.U
```

91283

```
Hello.  My na
me is Fred. I
work in the m
arketing depa
rtment.  I am
32 years old,
```

91285

```
#!/bin/sh
#
# This shell
#
trap 1 15 ""
usage()
```

Blocks

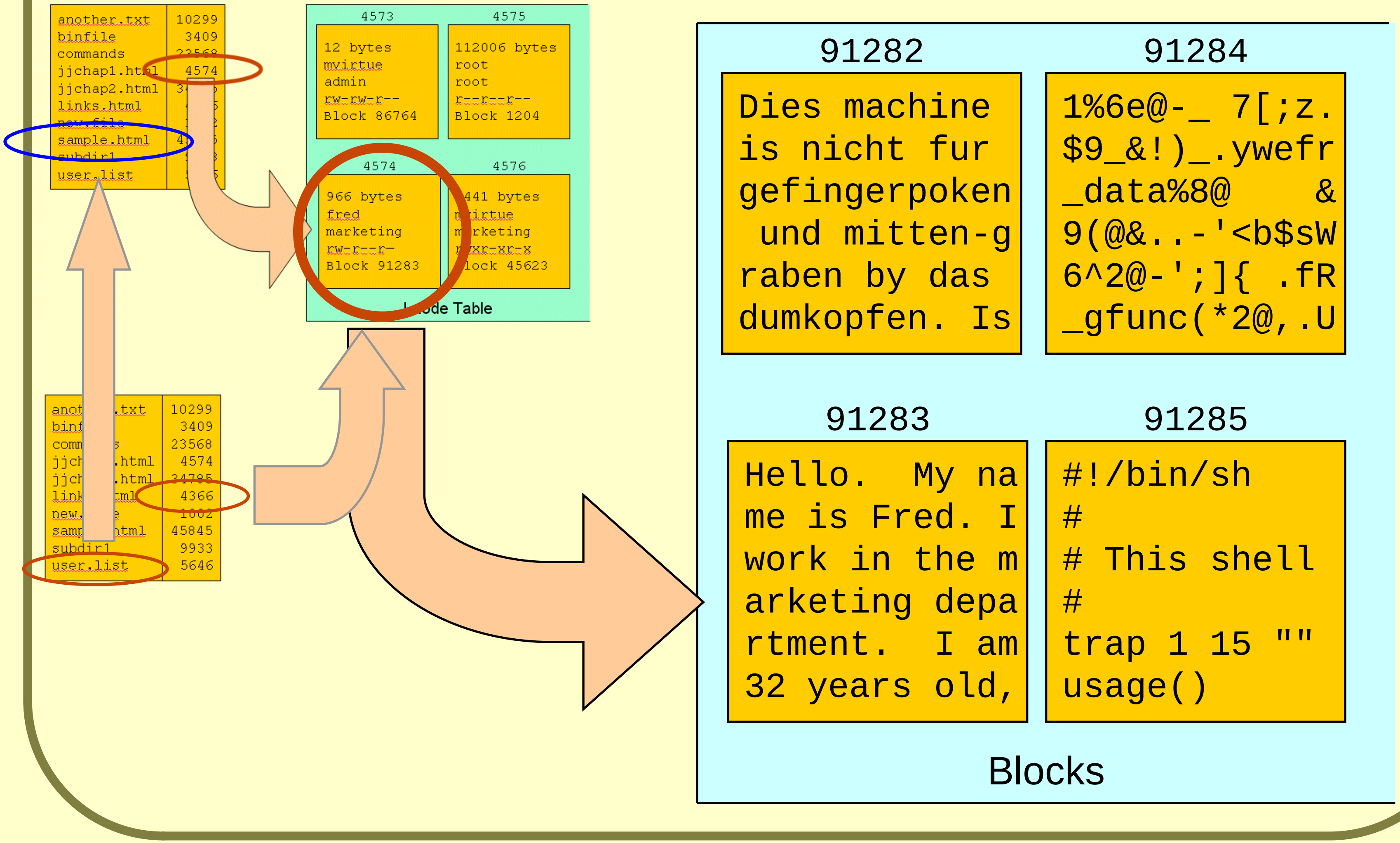
Linking Files

- The program `ln` is used to create links to existing files
- Its usage is identical to that of `cp`, except the file is not copied (duplicated), another reference (link) to the file is simply created

Symbolic Links

- All the links mentioned so far are also known as *hard* links, to distinguish them from *symbolic* links
- To create a symbolic link to a file means to create a small file (a separate file) that contains nothing but a reference to the original's *filename*
- This means that, when referring to symbolic links, the terms "original" and "link to original" make sense
- Microsoft Windows also offers a form of symbolic link, known as a *shortcut* (Windows does not support hard links)
- `ln -s` is used to create symbolic links
- Using `ls -l`, symbolic links are indicated by a `"l"` file type

Symbolic Links (cont.)



another.txt	10299
binfile	3409
commands	23568
jjchap1.html	4574
jjchap2.html	[illegible]
links.html	[illegible]
new.file	[illegible]
sample.html	[illegible]
subdir1	[illegible]
user.list	[illegible]

I-node Table

4573	4575
12 bytes mvirtue admin rw-rw-r-- Block 86764	112006 bytes root root r--r--r-- Block 1204

4574	4576
966 bytes fred marketing rw-r--r– Block 91283	441 bytes mvirtue marketing rwxr-xr-x Block 45623

anot[illegible].txt	10299
binf[illegible]	3409
comm[illegible]s	23568
jjch[illegible].html	4574
jjch[illegible].html	34785
link[illegible]tml	4366
new.[illegible]e	1002
samp[illegible]html	45845
subdir1	9933
user.list	5646

Blocks

91282

```
Dies machine
is nicht fur
gefingerpoken
 und mitten-g
raben by das
dumkopfen. Is
```

91284

```
1%6e@-_ 7[;z.
$9_&!)_.ywefr
_data%8@    &
9(@&..-'<b$sW
6^2@-';]{ .fR
_gfunc(*2@,.U
```

91283

```
Hello.  My na
me is Fred. I
work in the m
arketing depa
rtment.  I am
32 years old,
```

91285

```
#!/bin/sh
#
# This shell
#
trap 1 15 ""
usage()
```

UNIX File Types

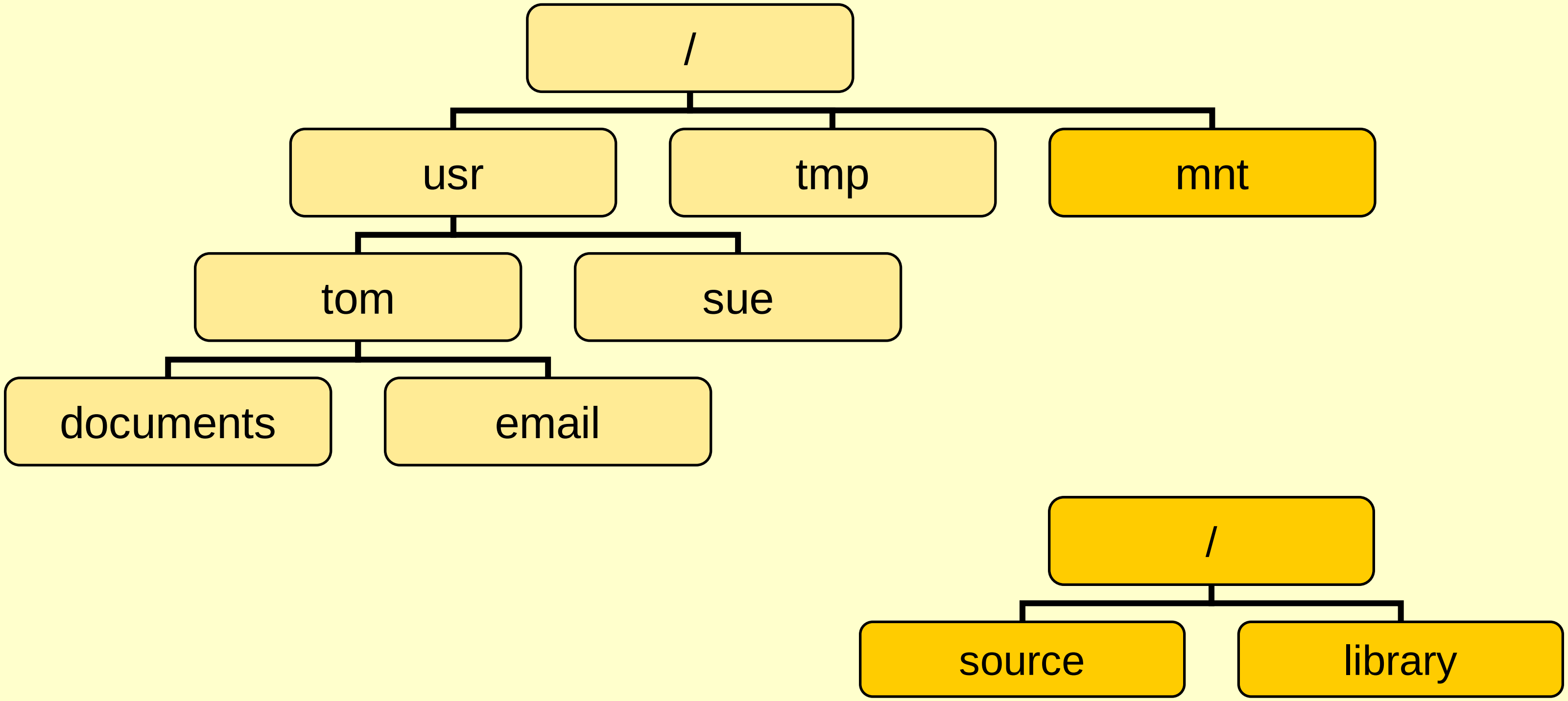
- Most files in UNIX represent actual files (in the regular sense). `ls -l` represents these with a `"-"`
- The table below shows the other types of file that exist:

`ls -l`	Type	Description
d	directory	Container for other files
`l`	symbolic link	Reference to another file
c	character device	Represents character-based hardware (e.g. serial port)
b	block device	Represents block-based hardware (e.g. floppy disk)
p	pipe (FIFO)	Communications file

Mounting

- Microsoft DOS and Windows use designators like `a:`, `c:` and `k:` to represent each accessible filesystem
- In UNIX, all files that users can access are found in a single directory tree (with its root called "`/`")
- This is achieved by *mounting* each available filesystem onto a subdirectory of the "root" filesystem, including:
  - Other hard disks
  - Floppy disks
  - Network disks
- Each filesystem is typically mounted onto an *empty* subdirectory

Mounting (cont.)



Mounting (cont.)

- It is easy to determine used and available disk space on any filesystem, by using the `df` (**d**isk **f**ree) command
- The disk space used by any directory can by displayed with the `du` (**d**isk **u**sage) command

Chapter 10

Communication

Using `telnet`

- `telnet` is a program used to obtain a login prompt (and/or a shell) on another UNIX machine
- The remote machine must be connected to the local machine by a TCP/IP network (including the Internet)
- In order to log on to a remote system, you will need:
  - A `telnet` program (such programs exist for many platforms, including Windows)
  - The *name* or *IP address* of the remote machine (for example, `shell.abc.com` or `123.234.12.99`)
  - (usually) A username and password (your local username and password will not necessarily work on the remote machine)

Using `mail`

- There is a simple and rudimentary email program available on all UNIX systems, called `mail`
- It is used in two modes:
  - To send an individual message to one or more recipients:

    ```
    mail fred@abc.com
    ```

    or

    ```
    mail root
    ```

  - To read and reply to messages and send new messages:

    ```
    mail
    ```

Chapter 11

Customising Your Shell Environment

Changing Your Login Shell

- Most users are unable to change their login shell. This is usually the job of the system administrator
- If you have superuser access, you can modify your login shell by trying one of the following:
  - Some UNIX systems offer a program called `usermod`
  - Start whatever program is used to create and modify user accounts and look for an option to change your shell
  - Edit the file called `/etc/passwd` and modify the last part of the line that corresponds to your user
- Edit the `.profile` file in your home directory and add the line `exec` *`new-shell`* to the end (for example: `exec /bin/csh`)

Environment Variables

- A shell *environment variable* is a shell setting that can be viewed and adjusted at any time
- Each variable is of the form *name=value*
- Shell variables may be set on the command-line as follows:

```
name=value
```

- For example:

```
PS1=Hello
```

- Existing variables may be modified in exactly the same manner

Environment Variables (cont.)

- The values of existing variables may be examined by typing:

```
echo $variable
```

- For example

```
echo $PS1
```

- The program `set` is used to show all environment variable that have been set
- Shell variable *names* cannot contain spaces
- Shell variable *values can* contain spaces. If this is required, enclose the value with double-quotes:

```
PS1="hello there"
```

Environment Variables (cont.)

- Many programs (including the shell itself) use shell environment variable as configuration options
- For example:

```
TEMP=/home/fred/tmp
```

- All variables that are to be used as configuration options should be *exported*, as follows:

```
export variable
```

- For example:

```
export PS1
```

Common Environment Variables

Variable	Purpose
`LINES`, `COLUMS`	Specify the dimensions of the screen for full-screen programs
`HOME`	Directory name. Various programs will create config files here. Where `cd` will go with no arguments
`LOGNAME`	The name of the currently logged-in user
`MAIL`	Location of the mailbox that the shell checks periodically to notify of new mail
`MAILCHECK`	Number of seconds between new mail checks
`PATH`	Program search directories – *See later module*
`PS1`	Shell prompt – *See later module*
`SHELL`	Your preferred shell, for programs like `vi` and `telnet`
`TERM`	Terminal type

Your PATH

- PATH is an environment variable used by the shell to determine where to look for executable programs
- PATH is interpreted as a list of directory names separated by "`:`". When a command is typed on the command-line, the shell will look in each PATH directory in the list in turn until an executable file with that name is encountered
- Users can add their own directories to the list (or even completely replace the list):

```
PATH=$PATH:/home/fred/bin
```

- or

```
PATH=.:$PATH
```

Your Prompt

- PS1 contains the text that represents your shell prompt
- As with any shell variable, this can be changed to anything you like
- Certain shells, such as `ksh` and `bash`, offer *dynamic* prompts, i.e. prompts that change with time. Dynamic prompts can include:
  - The current working directory
  - The current date and time
  - The current machine name
  - The current user
  - The command-history command number
  - The results of any program

Your `.profile`

- Changing your `PATH`, your prompt, or any other shell variable will only have an effect during the current shell session. In other words, as soon as the shell ends (on logout, etc), the changes are lost
- To make the same changes every time you log in, put the commands into the file called `.profile`
- `.profile` is a file containing commands that are executed automatically every time the user logs in with a Bourne compatible shell
- It is possible to add to `.profile` *any* command that you would like to have automatically run when you log in

Your `.profile` (cont.)

- `.profile` is to logging into UNIX what `autoexec.bat` is to booting into DOS
- `.profile` is an example of a *shell script*
- If you edit your .profile, it is possible to apply the changes without having to log out and log in again. Use the command:

```
 . .profile
```

Command-line Editing

- A *command-line history* is a shell feature that allows users to re-enter commands without having to retype them
- *Command-line editing* allows the current command or any previous command to be modified before ENTER is pressed
- If you use bash, you can use the PC-keyboard arrow keys t perform simple and intuitive command-line editing and history
- If you use sh, you have no command-line editing or history available

Korn Shell Command-line Editing

- If you use `ksh`, you have command-line editing and history available by way of `vi` commands
- If you imagine that your shell command-line is a little, one-line `vi` session, starting in *Insert Mode* each time `ENTER` is pressed, then you should have no trouble browsing your command history or editing command-lines
- For example, to re-enter the last command, type the following:
  - ESC (to enter Command Mode)
  - k (to go up one line)
  - `ENTER` (to enter the command)

Other Shell Customisation Options

- The program `umask` is used to specify the permissions that all files created by the current shell should have
- It usage is as follows:

  ```
  umask permissions
  ```

  where *`permissions`* are similar to those specified in chmod's numeric mode, except that *`permissions`* specifies the permissions that should be turned *off*
- For example, if the command

  ```
  umask 022
  ```

  is given then all subsequent files created will have a permission "mask" of **666 - 022 = 644** (`rw-r--r--`) (Note: "execute" permission cannot be specified)

Other Shell Customisation Options

- It is necessary for some shell accounts to exhibit the following behaviour:
  1. The user logs in
  2. A particular program is automatically started, such as an accounts package or some third-party software
  3. When that program exits, the user is immediately presented with another login prompt
- To cause this behaviour, add the following line to the end of the `.profile`:

```
exec program-name
```

- The only way `.profile` can be subsequently adjusted is to log in with a different username

Other Shell Customisation Options

- Other alternatives to achieve the same result include:
  - Adding the following lines to the end of the `.profile`:

    ```
    program-name
    exit
    ```

    (Note: this method uses slightly more memory)
  - Making the program your shell

    (Note: with this method, no shell variables can be set or other commands run)

The End