Department of Computer Science

Introduction to Unix

Kate Lance
March 1996

Contents

1 Unix Overview

1.1 The Computer Science network

- We run a system of powerful multi-user workstations which support both staff and students. Most people access the computers via X-terminals, which provide window-system graphical displays. From the user's point-of-view, it is just like working at the computer itself.
- The computers are mainly Sun SPARCstations, which run a version of the Unix operating system called Solaris.
- All users of the computers are allocated their own area of disk space, their *home directory*, often called just ~ (tilde).
- When you start a work session, you are logged on to your home directory, which contains a number of files to set up and define your working environment. All of the names of these files start with the . (dot) character.
- When you type in a command, you are actually operating a program called a *shell*, which **interprets** and **executes** commands from users to the *kernel*, the software interface to the machine hardware.
- All of the space on disks is arranged in a hierarchy, or tree. Each level of the tree is symbolised by a / character. The top level is just called /; the next level down, for instance, might have executable programs in **/bin**; files of source code in **/src**; and directories for users in further levels under **/home**.

1.2 Your home directory and initialization files

- On our computers we use a shell called *bash*. It is defined by several files:
  - ~**/.bash_profile**: this contains commands to set up shell and environment **variables**. This file is owned by the system and you may not change it.
  - ~**/.bashrc**: this contains your $PATH and a number of abbreviations (**aliases**) for long commands. It is also owned by the system.
  - ~**/.user_profile**: the file for you to set your own variables.
  - ~**/.userrc**: the file for your $PATHs and abbreviations.

The X11 window system provides your graphical interface. It's appearance is defined by three files installed in your home directory. They are:

- **.Xresources**: this mainly sets up the size and font of the windows, called **xterm**s.
- **.twmrc**: this is a setup file for the **twm** window manager, the program that runs your X11 session. Menus are defined, such as one for the machines you can log into, another for functions such as **Logout** or **refresh**, plus default variables and mouse button settings.
- **.xsession**: this actually starts up your **xterm** windows, clocks, calendars, email indicators (**xbiff**), and other programs.

1.3 Passwords and security

- When you log on, you enter your login name and a password (which does not appear on the screen).
- Your password must be at least 6 characters long, up to 8 characters, with at least one digit or special character.
- Your password must be changed every few months or so, with the command **yppasswd**.
- Do **not** use *any* dictionary words, personal names, usernames, work-related names, nicknames, etc. for passwords. Also do not just append or prepend a digit to any of the above, or change "o" to "0" or "l" to "1", and so on. These variations are all very easily deduced by widely-available password cracking programs.

1.4 Getting assistance

- **man** *command* – on-line manual pages in full on the particular *command*.
- **whatis** *command* – one-line description of the command.
- **man -k** *keyword* – to see a quick reference list of **all** mentions of the keyword (or command) in the manual listings.
- **apropos** *keyword* – same as **man -k** *keyword*.

1.5 Commands (in general)

- Commands are shorthand instructions to tell the system what to do. They are interpreted by the shell program. Their format is usually
  command *argument_list*
  where *argument_list* consists of options and/or filenames.
- Options modify commands, and are usually single letters or combinations of letters preceded by a dash (eg **man -k ls**).

- Use a semicolon (;) to separate multiple commands on the same line.
- Use a backslash (\) to continue a long command onto the following line.
- **Control/c** will stop the activity of a command and return control to you.

1.6 Files vs processes

- **Files store information:**
  - Data, text, directories, program code, devices
- **Processes do the work:**
  - Shells, commands, executing programs, daemons (system processes)

2 Directories

2.1 Moving around

- Directories organize collections of files.
- **cd** or **cd ~** puts you into your home directory.
  - **.** refers to whichever directory you are in at that moment.
  - **cd ..** puts you into the directory just above the present one, the **parent** directory.
  - **cd** *directory_name* puts you into the named directory.
- Pathnames are the full names of files or directories. Each level is signified by a /, e.g. **cd /internet/username/dir1/subdir2** means to change level to directory **subdir2**. This is an absolute pathname. If you were already in directory **dir1**, then **cd subdir2**, without a leading /, would be the relative pathname to that directory.
- **$PATH** is a variable defined in your startup files. It lists the directories, both your own and system ones, to be searched whenever you ask for a program to be run, e.g.
  ```
  > echo $PATH
  /bin:/usr/bin:/usr/ucb:/usr/openwin/bin:/usr/bin/X11:/local/bin
  ```
  (**echo** *string* writes *string* to the terminal.)
- **pwd** tells you which directory you are currently working in (print working directory).

2.2 Housekeeping

- **mkdir** *directory_name* is the command to create a directory.
- **rmdir** *directory_name* removes the directory. It must not contain any files, otherwise **rmdir** will not work.
- **rm -r** *directory_name* removes a directory and all its files (and other subdirectories) **recursively** from within the specified directory, then deletes the directory itself.
- **mv dir1 dir2** moves, or renames, **dir1** to **dir2**. If the directory **dir2** already exists, **dir1** is moved *inside* **dir2**.

2.3 Names

- In Unix, *all* commands, directory names and file names, are **case-sensitive**: files **Rubbish** and **rubbish** are quite different.
- Names may be comprised of letters, numbers, underscore (_) and dot (.). Do not use special characters such as $, >, <, and |.
- There are **no version numbers** in Unix. Make a copy of a file to a different filename before editing it (for instance) otherwise it will be overwritten by the new version.
- File type extensions such as .exe, .dat, etc. are not used. Some program source files have extensions such as .f for Fortran code, .c for C code, but these are actually a matter of convention rather than being necessary for the operating system.

2.4 Directory listings

- **ls** gives you a listing of all the files in a directory by name.
- **ls -F** gives you a listing with filetypes indicated by a special character: '/' for a directory, '*' for an executable file, and no character for a data file.
- **ls -l** gives you a long listing of the directory contents, including permissions, owner, size, and date created.
- Some other useful options to **ls** are **-a** show all (including files that start with '.'), and **-t** sort in time order (most recently modified).
- The options to **ls** are different in SunOS4 and SunOS5. **-g** in SunOS4 shows group ownership, while in SunOS5 it shows a long listing without the owner column.
- In your **.bashrc** file some of the options to **ls** have been aliased to the correct format for whatever machine you're working on, so have a look at the different formats and the aliases you can use.

- An example of a directory listing (using **ls -lagF**) is

```
-rwxr-xr-- 1 c9399999   cs2     693 Apr 17 11:58 .bashrc
-rwxrwxr-x 1 c9399999   cs2    1024 May 11 10:14 prog1*
drwxr--r-- 2 c9399999   cs2     512 May  6 11:22 texdir/
-rw-rw-r-- 1 c9399999   cs2    9850 May 14 13:24 unix1.tex
```

  The items in the example above, from left to right, are:

  - the **permissions**, which are the first 10 characters. 'd' signifies a directory, then there are three fields of three characters each. The three fields are permissions for **owner**, **group** and **world**. The three characters in each field are `r` (read), `w`(write), and `x` (execute). A dash (–) indicates that the permission is turned off.
  - The **links** to files. Here `1` means that the file has one link, to the present directory, while `2` for the subdirectory means that it links both to its own files as well as the present directory.
  - The **owner** of the files.
  - The **group** to which the owner belongs. Other members of the owner's group have more access permissions than people outside the group, ie. the world. In this case the `computer` group has `w` (write) access to prog1 and unix1.tex, and `x` (execute) permission for prog1.
  - The **size** of the file in bytes.
  - The **date** and time the file was created. This refers to the most recent 6 months; previous times are in the format of date and year.
  - The **name** of the file, and the filetype (/ directory, * executable).

3 Files

3.1 Name completion

If you type part of a filename or a command and press **Tab**, the shell will complete the name for you *if the part you typed is unique*. If it is not unique, the terminal will beep. Another **Tab** will show you a list of commands or filenames which would complete the non-unique entry. Type additional characters until the name is unique, then press **Tab** again.

3.2 Reading text files

- **cat file1** types the entire contents of **file1** to the screen.
- **more file1** also types **file1** but only a screenful at a time (press spacebar for the next screen).
- **less file1** also types **file1** a screenful at a time (it has other useful options).
- **head file1** types the first 10 lines of **file1**.
- **head -25 file1** types the first 25 lines of **file1**.
- **tail file1** types the last 10 lines of **file1**.
- **tail -34 file1** types the last 34 lines of **file1**.

3.3 Moving and removing

- **mv file1 file2** moves or renames **file1** to **file2**. **file1** will replace **file2** if **file2** already exists.
- **cp file1 file2** copies or overwrites **file1** to **file2**.
- **rm file1** deletes **file1**.
- **rm -i file1** asks for confirmation before it deletes **file1**. This is useful when deleting a number of files, e.g. **rm -i file*** (see below, **Wildcards**).
- At this site **mv**, **cp** and **rm** use the **-i** option by default. You can remove the option by typing, for instance, **\rm** or **'rm'**, or putting the line **unalias rm** in your **.userrc** file.

3.4 Comparing, counting, searching, and sorting

- **cmp file1 file2** compares **file1** to **file2** and states the first line that differs. This may be used for any file type.
- **diff file1 file2** compares **file1** to **file2** and shows which particular lines differ. It is only for text files.
- **wc file1** counts the lines, words, and characters in a file.
- **grep string1 file1** prints every line in **file1** that contains **string1**.
- **sort file1** sorts the lines of **file1** into alphabetical order.
- **touch file1** creates the empty file **file1**.

3.5 Wildcards

- **?** matches any single character.
- ***** matches any pattern of characters, eg. **ls file*** would show **file1, filerubbish** and **file34**.
- [**list**], eg. **cat file[123]** means type **file1**, **file2**, and **file3**.
- [**lower-upper**] (inclusive of **lower** and **upper**) eg **cat file[a-cx-z]** means to type **filea, fileb, filec, filex, filey**, and **filez**.

3.6 Finding files

- **find** is a command to search for and act upon files.
- **find . -name '*.o' -print** means to start from the current directory (.), look for the pathnames of any files with the extension .o, and print them on the screen.

- **find /dir1 -name 'rub*' -exec rm {} \;** means to find files starting with the letters 'rub' in the directory **dir1**, **-exec** means to execute the following command (**rm**), and **{}** means to substitute the pathname here. \ terminates the execute option.
- See the manual pages for more on **find**.

3.7 Permissions and filetypes

- **chmod** changes permissions on any files. It is used in two ways:
  - the three permissions, **rwx** are indicated by octal numbers, ie. **r** is 4, **w** is 2, and **x** is 1, for a total of 7 for all permissions, separately for each field of user (the owner), group, and other (the world). The format is
    chmod permission_numbers filename
    - **chmod 777 file1** changes permissions to `-rwxrwxrwx`
    - **chmod 444 file1** changes permissions to `-r--r--r--`
    - **chmod 750 file1** changes permissions to `-rwxr-x---`
  - the other way is to use letters and operators for each field. The format is **chmod [who operator permission] filename**. **Who** is **u** (user), **g** (group), **o** (other), **a** (all). **Operator** is + (add), – (remove), = (assign). **Permission** is **r** (read), **w** (write), **x** (execute).
    - **chmod a=rwx file1** changes permissions to `-rwxrwxrwx`
    - **chmod a–wx file1** changes permissions to `-r--r--r--`
    - **chmod u+x,ug+w file1** changes permissions to `-rwxrw----`
- **umask** is set in your **.bashrc** file. All files you create will have permissions set so that access is *denied* according to the **umask**, e.g. **umask 027** means that **user** (owner) is not denied access at all, **group** does not have write permission, and **other** are denied all access. **umask** by itself shows your current setting in octal, and **umask -S** shows you the setting in symbols.

4 File Manipulation

4.1 Standard input, output, error, and redirection

- **Standard input** is what you type into the terminal from the keyboard.
- **Standard output** is what appears on the terminal as output from commands, programs, etc. (excluding error messages).
- **Standard error** is the diagnostic output which also appears on the terminal.
- < is a redirection command. It means to **take input from** somewhere other than the keyboard, eg. **sort < file1** would get the input for the **sort** program from **file1**.
- > also redirects, and means to **direct output to** somewhere other than the terminal, eg. **sort file1 > file2** would sort **file1** and write the output to **file2**.

- **&>** or **>&** redirects error messages to a file. Errors normally appear on the terminal screen, even if the standard output has been redirected to a file.
  - To redirect error messages to the **same** file as the standard output, use **command &>** *outerrfile*.
  - To redirect output to one file and diagnostics to a **different** file, type **command >** *outfile* **&>** *errorfile*.
  - To throw the error messages away, send them to the 'bit bucket', **/dev/null**, eg. **command >** *outfile* **&> /dev/null**.
- Redirection applies only to **one command group** at a time: if you want to do a series of commands then redirect the output, put the series of commands in parentheses, eg, **(command ; command ; command) > outfile**.
- Parentheses define a **command environment** in which a series of commands may operate within a subshell, without changing anything in the current environment, eg. **(cd dir1; sort file1 > file2)** would return to the current directory after executing the command without having to **cd** back to it.
- The command **typeset -x noclobber** in your **.bash_profile** file prevents overwriting of a file if you accidently redirect output to a file that already exists.
- Override **noclobber** by putting | after the redirection command if you really do want to overwrite a file, eg.
  sort file1> | **file2**
  will sort **file1** and overwrite any existing contents of **file2**.

4.2 Append to, read from here

- **>>** is the **append** command, eg.
  cat file1 >> file2 appends the contents of **file1** to **file2**
  date >> datefile creates the file **datefile** with the system date and time in it.

  typeset -x noclobber in your **.bashrc** file **prevents** the creation of a file with **>>**, so use **>>** | to override it.
- **<<** creates a **here document**. This takes multiple lines of keyboard input as standard input for a particular command. The format is
  command << *marker*
  input
  ....
  marker
  Where *marker* may be a single character or a word which does not appear in the text. This is a quick way to write a small text file, for instance.

4.3 Pipes, filters, and tees

- **Pipes (|)** direct the standard output of one **process** into the standard input of another **process**, eg. **ps | sort | more** would sort a list of processes and type them a screenful at a time.

- **Filters** are sequences of pipes and commands that operate on files, e.g.
  ps aux | grep username > file1
  would filter all the processes being run by **username** and write them to a file.
- **tee** takes standard input and sends a copy to a file (or files) and a copy to the standard output (usually the screen), eg. **ls | tee file1** would put a list of filenames on the screen and also write the list to **file1**.
- **tee -a** appends to an existing file.

4.4 tar and compress

tar (tape archive) saves to and restores files from archive files or tapes. Format is
tar *option archive_file file(s)*
(This is one of the few Unix commands for which the – is unecessary.) Some options are:

- **c** initialize a new archive
- **v** verbose, display information during archiving
- **r** add file to end of files in archive
- **f** force to use the following archive
- **p** preserve original modes of extracted file
- **t** list files in archive
- **x** extract file from archive

compress and **uncompress** may be used to reduce file sizes, to free up your disk space. Examples:

- **tar cf dir2.tar dir2** copies the contents of **dir2** to a new archive file, **dir2.tar**.
- **tar rf dir2.tar file1** appends **file1** to **dir2.tar**.
- **tar tvf dir2.tar** shows you the contents of **dir2.tar**.
- Extract the files with **tar xfp dir2.tar**.
- To extract **file1** only, **tar xfp dir2.tar file1**.
- **compress dir2.tar** reduces the file in size (roughly by about half). The output is a file called **dir2.tar.Z**. Use **zcat** to examine the contents of a compressed file.
- **uncompress dir2.tar.Z** restores **dir2.tar**.
- **gzip** and **gunzip** work the same way as **compress** and **uncompress** but they are even more efficient. The suffix of a file compressed with **gzip** is **.gz**.

5 What's Going On

5.1 Time and space

- **date** gives the system date and time.
- **cal** [**month**] **year** gives a calendar for a specified year and (optional) month. Month is numerical, 1–12, and year must be in format yyyy.
- **du** [**directory**] shows your disk usage in kbytes. **du -s** gives the total only, and **du -a** gives the size of each file.
- **df** [**filesystem**] gives the total filesystem usage in kbytes, for a particular filesystem, or all if no filesystem is specified.
- **quota -v** shows your disk quota. In order the fields are: filesystem, amount used (in kilobytes), filesystem quota, maximum excess quota available for a short period, time left to reduce usage if you are over quota; plus the same fields for inodes. (Every file on the system has a unique inode (index node). It describes the actual disk layout of the file data, plus ownership, permissions, access date, size etc.)

5.2 Who am I

- **whoami** gives your login name.
- **finger** shows who is logged on, their terminal line, how long their terminal has been idle when they logged on and from where they logged on.
- **w** and **who** show users, terminal lines, time logged on, idle time, the processes being executed or which machine or terminal servers are being used.

5.3 Process control

- **ps** shows the process (or job) status, that is, which processes you are currently using, such as programs, shells, system commands, etc.
- **ps -aux** (SunOS4) or **ps -ef** (SunOS5) gives a long, user-orientated list of all processes occurring on the system.
- **psg** is a local alias that shows all your current processes, and **psg***string* shows any processes with *string* in the listing.
- & at the end of a command line puts the process in the **background**, ie. it runs while you use the terminal for other activities. You may put more than one job in the background.
- **jobs** tells you which jobs you have running in the background and their numbers.
- **fg** brings a job to the **foreground**. **fg %n** brings a specified job to the foreground, where **n** is the appropriate number from the **jobs** command. **%n** may be replaced by the process number (PID) from the **ps** command.

- **ctrl/z** temporarily stops a foreground job from executing so that you may do something else from the terminal.
- **fg** or **fg %n** restarts stopped job **n** in the foreground.
- **bg** or **bg %n** restarts stopped job **n** in the background.
- **kill %n** terminates a specified job, **kill %** terminates the current job.
- If you have stopped jobs in the background then you will get a message to this effect when you try to logout. You can then deal with the job, or logout again, and the job will be terminated.
- If the job has been put in the background with **&** at the end of the command it will continue to execute after you have logged out.

5.4 History (a sense of deja vu)

- The **history** facility is one of the major attractions of the **bash** shell. It keeps a list of all the commands you have typed in a session, usually up to 1000 commands.
  - Type **history** to see the list of previous commands.
  - Repeat the most recent command with **!!**
  - Repeat command 25 (for example) with **!25**
  - Repeat a command beginning with **man** (for example) with **!man**
  - You are not allowed to edit or delete your **.bash_history** file.

To **re-use and edit previous commands** (under **bash**) press **Esc** −. Then use − and + to move backwards and forwards respectively through the commands. They can be changed with **vi** editing commands. **Control/C** lets you escape this mode. You will need the setup file **.inputrc** in your top directory for this to work (it is installed in all new accounts).

5.5 Aliases

- **alias** allows you to abbreviate commands or substitute your own terms for commands. Format is **alias substitute=command**. Note there are no spaces either side of "=".
- An essential alias to have is **alias rm='rm -i'** which means to enquire if deletion is really desired before acting. Note the single quotes which force **rm -i** to be treated as a unit, a compound command.
- To disable an alias temporarily, for instance to delete without checking first, type **\rm files**, or **'rm' files** at the keyboard so that the **-i** is excluded.
- To remove the aliased option altogether, put the line **unalias rm** in your **.userrc** file.
- Aliases are usually defined in the **.bashrc** file. Some system aliases may be already set up for you. Type **alias** with no arguments to see which aliases you may have.

5.6 Special characters

- **Quoting** (or **escaping**) is the way to use special characters or variable names with their literal or ordinary meaning.
- \ is used to escape a single character, eg. at the end of a line it escapes the newline character, so it acts as a command continuation.
- The following characters have special meaning to the shell (they are metacharacters) and need to be escaped to be used literally in a command, eg **echo *** prints * to the screen, while **echo *** shows the list of filenames in the directory.
  - `*[]?{}~-` filename expansion
  - `><&!` redirection
  - `\'"` quoting
  - `!^` history
  - `$` variable identifier
  - `()` command group
  - ‘ command substitution
  - `;` command delimiter
  - `&` background jobs
  - `|` pipes
  - **return** newline
  - **space, tab** argument separator
- Different type of quotation mark vary in their activity:
  - single quotes (’) quote all characters literally.
  - double quotes (”) quote most characters literally, but expand shell variables.
  - backwards single quotes (‘) expand command substitution variables (i.e. show the output of the quoted command), e.g.

```
> typeset d=date
> echo $d
date
> echo '$d'
$d
> echo "$d"
date
> echo ‘$d‘
Mon Feb 26 09:59:23 EST 1996
```

6 Variables and Commands

6.1 Builtins

Not all of the commands that you issue are Unix commands. Some are actually built into the shell itself, and they are usually the more efficient to use. For instance when you run the Unix command **pwd** the parent shell has to create a subshell to run the program **/bin/pwd**. There is a shell *builtin* called **dirs** (directory stack) which gives the same information (your home directory is indicated by ~) but which is faster and easier for the system to implement than **pwd**.

You have already been introduced to many of the shell builtins, with commands such as: alias, bg, cd, dirs, exit, fg, history, jobs, kill, login, logout, set, umask.

Another useful one is **source**, which is used to run shell scripts or special files such as **.bashrc** and **.bash_profile** in the current environment, without starting up a new subshell. For example, if you had changed your **.userrc** file and wanted the new version to apply to the present session, type **source .userrc**.

6.2 Variables

Your computing environment is organized and maintained by means of defined variables. Some are predefined when you first log on, getting information from system files. Some you define for yourself. These variables are further classified as either environment variables or shell variables. **man bash** describes all of the variable options in great detail.

Environment variables are maintained by Unix rather than the shell. They store information about your user environment, your login directory, your username, your shell, and your terminal type, plus there are other optional definitions. These variables are available within the shell and are also exported to programs you run from the shell, such as editors, mail, and shell scripts.

Environment variables are usually defined in your **.bash_profile** and **.user_profile** files, with the commands **typeset** and **declare**, for example:
typeset -x noclobber, or
declare -x EDITOR=vi
The **-x** exports the definition to subsequently opened shells, or you may explicitly type **export EDITOR**. The command **printenv** shows a list of your environment variables.

Shell variables are maintained by the shell, and are available only within the shell, or within any new shells that you may open. They are used to define your path, your umask, your ulimits (resource usage limits), and any aliases you may wish to set up. Shell variables are usually defined in your **.bashrc** and **.userrc** files. Use **typeset** rather than **declare** in these files. The command **set** shows a list of your currently-defined shell variables.

Note: on this system, **.bash_profile** and **.bashrc** are not changeable by users, they are the same for everyone. However, you *are* able to edit your **.user_profile** and **.userrc** files.

6.3 Customizing your own variables

Shell variables aren't always system ones, you can create your own. Define them with the **typeset** command in the format
typeset -x variable=*string*

If you type this at the terminal the definition will exist only for that login session. If you want the definition to be always available, put it in your **.userrc** file. However if you have lots of these definitions you may slow yourself down as the **.userrc** file is run every time a shell script is executed. If they aren't going to be used in any scripts, it may be better in this case to put them in your **.user_profile** file instead, as it is only run once.

When actually using shell variables you invoke their meaning by putting $ in front of them, e.g., echo **$variable** would return *string*.

6.4 Command substitution

Command substitution replaces the command itself with the command output, using shell variables and the backwards single quote (grave accent) character (‘) e.g.,
typeset d=‘date‘
assigns the current date and time to variable **d**. Use command substitution by putting $ in front of the variable (as in variable expansion), e.g., **echo $d** types the actual date and time to the screen.

Command substitutions may also be embedded in strings e.g.,
echo "there are ‘who | wc -l‘ users logged on"
would return
there are 6 users logged on

6.5 Process creation

A process consists of

- a unique systemwide identification number (PID)
- a current directory and a table of open files
- a program area with executable instructions
- data areas with variable and environment definitions
- the operating system structures required by Unix
- and a life cycle: it is created by a program, it lives while it works, and it dies when the work is completed.

When the parent process begins to execute a command, it first calls a fork. The fork is a Unix system routine which duplicates all the process characteristics of the parent, creating the child process (identical to its parent apart from having a different PID).

The parent process calls a wait routine, which suspends the parent's activity, while the child process operates within a subshell created by the parent. The child process calls a Unix routine (**exec**) to actually execute the command program.

exec overlays the child process with the command program instructions and data, but the child environment is not altered (the overlaid program for a shell script is /bin/bash). The child process finishes executing, calls the exit routine, dies, then the parent process wakes up, to display again the prompt for interactive commands.

6.6 Command execution

There are three levels of command activity. Each is successively more demanding of system resources:

- If the command is one of the shell's own builtins it runs it directly inside the parent shell. This is the most efficient type of command procedure.
- If the command is a Unix command or your own executable, a subshell is created by the parent process for a child process to run the program.
- If the command is a shell script, the child process calls /bin/bash, which creates a new shell to replace the child process, in which it re-runs .bashrc (but not .bash_profile).

6.7 Efficiency miscellany

- Use builtins rather than Unix commands if possible to avoid excessive creation of processes and subshells.
- Use single Unix commands that operate on lines of text, like **grep** and **sort** rather than shell scripts to avoid the overhead of a new shell and a re-run of **.bashrc**.
- Use shell scripts rather than C programs for sequences of Unix commands. No compile, link, load, or object files to maintain.
- Use C programs rather than shell scripts for numeric or character data because for these C is the more efficient.
- If you don't need your aliases etc. you can run a shell script without running .bashrc by typing **bash -norc** <**filename**>, for a fast startup.
- In general, always **cd** to a directory before operating on files in it, because full pathname commands are less efficient than local commands.