

Preface

We are very glad to see you in this class of “Introduction to UNIX” at UITEC this year. UNIX becomes one of the most popular and powerful operating systems which can work on many of the engineering work stations or the personal computers. Since UNIX is originally not designed to be user-friendly system, at first you may feel somewhat irritating to use. However, it is very useful afterwards to learn and be accustomed to UNIX, because the chances that you encounter the computers on which UNIX is running is rapidly increasing, in almost every sorts of computers in every kinds of countries in the world. In this class, we will start from the introduction to UNIX system, progress to the fundamental operations, basic commands and file systems, brief explanation about the background jobs and other advanced commands, pipe and redirection. Next we will learn about the text editor (vi) to write text files. One of the final object of this class is to become familiar with the electric mail (E-mail) which has become one of the de-facto-standard tools of the communication recently, as long as the telephone or the facimilie. The schedules of the class are planned for the total beginner of UNIX, so someone who already has some knowledge about UNIX may feel somewhat boring. If you feel so, please help yourself play with the machines as you like, because in the learning process of the computers it is more important for you to make your hands in motion and type many commands by yourself, than to only read the books or listen the class passively. We prepared the two textbooks in aid of the class; one is “*Introduction to UNIX*” (Que corporation 1994, hereafter we call it the black textbook) which is rather formal and covers almost all the important things about UNIX. The other is “*UNIX for Dummies*” (IDG books, the yellow paperback one, hereafter we call it the yellow textbook) which is rather informal and easy to read with a lot of jokes and pans in it. We will sometimes consult these texts in the class (mainly the black one).

As an exercise environment, we offer you a number of units of the disk-based SPARC Station ELC installed in the computer room. A printer is available. These work stations support an operating system called SunOS which is based on UNIX. Exercises hereafter are conducted in the X window system environment.

Then let’s enjoy learning UNIX with us!

November 10, 1994
Takashi Ito and Yukiko Yokoyama

Contents

1	What is UNIX, and Why UNIX?	5
1.1	Before starting	5
1.2	What is UNIX?	5
1.2.1	Characteristics of UNIX system	5
1.2.2	UNIX and other operating systems	7
1.2.3	Components of a UNIX system	7
2	Login & Logout	10
2.1	Starting up a system	10
2.2	Entering your login information and password	10
2.2.1	login information	10
2.2.2	Entering your password	10
2.2.3	logout procedure	11
2.3	Fundamental operations of X-Window System	12
2.3.1	Starting up	12
2.3.2	Start the window manager	12
2.3.3	Start the xterm	12
2.3.4	Window operations	14
2.3.5	Closing the X-Window system	14
3	Files & Directories	16
3.1	Everything in UNIX is a file!	16
3.2	Directory tree	16
3.3	Directory referencing	16
4	Permissions	19
4.1	Concept of permissions	19
4.2	Using <code>ls -l</code> command	19
4.3	Directory permissions	19
4.4	Change permissions	21
4.5	Some other topics	22
4.5.1	Options of <code>ls</code>	22
4.5.2	UID and GID bit programs	22
4.5.3	Symbolic links	22
5	Other topics on UNIX filesystems	25
5.1	Restrictions of file name	25
5.2	Hidden files	25

5.3	Wildcards	25
5.4	Important commands	25
5.5	Redirection in UNIX	29
5.5.1	Redirecting output (using >)	29
5.5.2	Redirecting input (using <)	29
5.6	Piping in UNIX	30
5.7	alias (csh command)	32
5.8	history (csh command)	33
5.9	finger and talk	36
5.10	Background jobs	38
5.10.1	background jobs and foreground jobs	38
5.10.2	Job control: foreground, stopped, background	38
5.11	Showing processes	40
6	E-mail	43
6.1	Sending mails	43
6.1.1	The simplest way	43
6.1.2	Recipient	44
6.1.3	Send a file	44
6.2	Reading mails	44
6.3	Structure of a E-mail message	46
6.4	xbiff	48
7	vi editor	50
7.1	Major modes of vi	50
7.2	At least you should know	50
7.2.1	To start vi	50
7.2.2	To quit vi, saving the text	51
7.2.3	Most basic cursor movement	51
8	Environment files	53
8.1	.login	53
8.2	.cshrc	53
8.2.1	Backquotes in C-shell	54
8.3	.logout	55
8.4	.xinitrc	55
9	Using compilers	57
9.1	The name of output	57
9.2	Using mathematical functions	58
10	Emacs editor	60
10.1	At least you should know about emacs	60
10.1.1	Start up	60
10.1.2	To quit emacs, saving the text	60
10.1.3	Basic cursor movement	61
10.2	On-line tutorial	61
10.3	Important keybinds	61
10.4	Undo function	61

11 C shell	66
12 Miscellaneous commands	99
12.1 Compressing and uncompressing files	99
12.1.1 compressing	99
12.1.2 uncompressing	100
12.1.3 zcat	100
12.2 Encoding/decoding files	100
12.2.1 encoding	100
12.2.2 decoding	101

Chapter 1

What is UNIX, and Why UNIX?

1.1 Before starting

Here we will briefly explain some points to notice for the preparation of this classes.

- Please don't forget to take off your outer shoes and put on a mule in the room.
- Smoking, eating and drinking are prohibited in the room.
- Computers are very delicate, so don't handle it violently. If you have hit some unknown keys accidentally and the machine has stopped, don't be in panic and call the instructor.

1.2 What is UNIX?

Reasons why we will learn UNIX system, and the answers to the question “what is UNIX?” is explained in detail from p.11 of the black text and from p.17 of the yellow text. UNIX is one of the operating systems (such like MS-DOS), which is developed from over 20 years ago and licensed by the UNIX System Laboratories Inc, and the word “UNIX” is a registered trademark of the AT&T. Twenty-year survival is fairly long as an operating system of the computers, which means there are many good points in UNIX system than the other ones.

1.2.1 Characteristics of UNIX system

Multitasking

UNIX can perform more than one task at a time for each user, which is a capability called *multitasking*. UNIX does this through a process known as *time sharing system*, or TSS. Although the computer seems to be devoting its full time to each task, it spends only fractions of a second with any job, switching its attention from one job to another. As one of the many tasks UNIX performs, UNIX continuously decides which job to run next, and how long to spend on each job. This process occurs so quickly (about an order of 10^{-6} seconds) that it is usually not even visible to the user. This is the biggest difference with a sigletasking operating system such like MS-DOS.

Multiuser capability

Besides being able to run more than one task at a time, UNIX can serve more than one *user* at the same time. UNIX does this through time sharing method, similar to case of the multitasking.

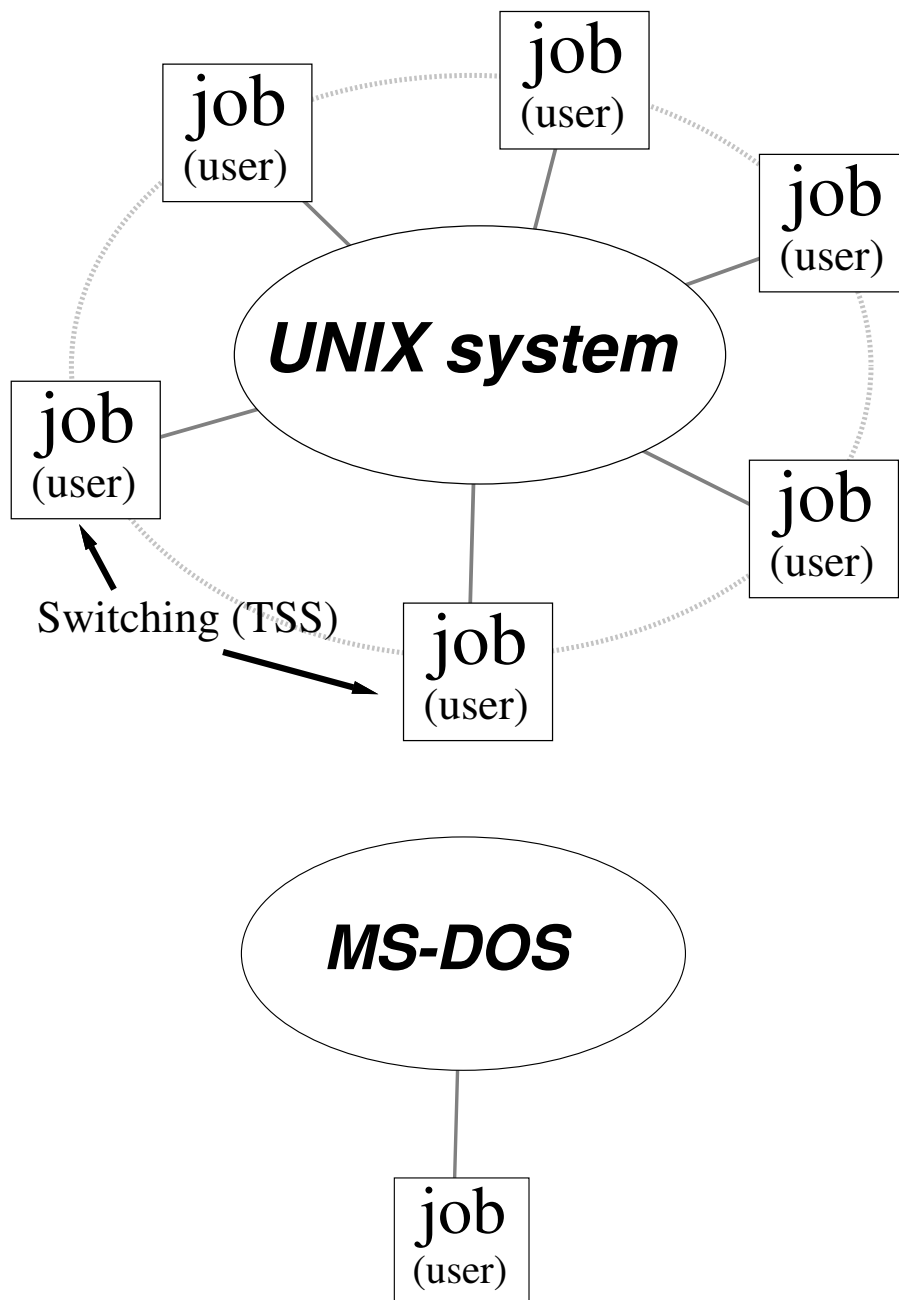


Figure 1.1. Multitasking of UNIX system vs. singletasking of MS-DOS

High portability

Portability of UNIX system is generally very high; that is, if you intend to make UNIX system work on a computer on which none of UNIXs are working on ever, it is fairly easier to do that (rewriting the source code of the UNIX system, compiling and implementation) than the other operating systems. It is mostly because source code (source program) of UNIX is written in C language which is independent of each computer hardware. Therefore UNIX has been ported every kinds of hardwares from the small personal computers, workstations, mainframes, and to the huge supercomputers. This is one of the biggest reason why we will learn UNIX here.

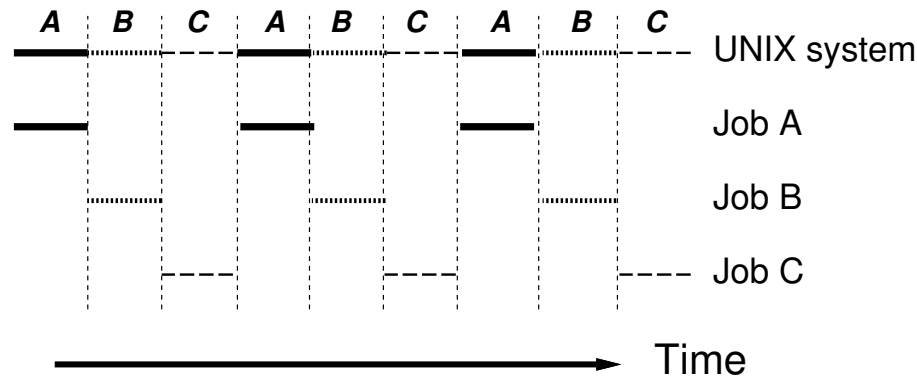


Figure 1.2. Schematic illustration of the time sharing system (TSS) of UNIX

1.2.2 UNIX and other operating systems

Since MS-DOS is currently the most widely known operating systems, how does UNIX compare? Why would you want to use UNIX as the operating system for your PC rather than MS-DOS (or OS/2, Windows NT, etc...)?

Briefly, UNIX is the only operating system that offers multitasking, graphics, and cross-platform compatibility (mentioned above). File sharing, network printer service, remote application execution, mutiuser access, and a graphical user interface are available for the UNIX users, not for MS-DOS users. If you need to work in a networked computer system¹, UNIX is currently the only operating system that combines all of these elements. In Table 1.1 we compares the features of several operating systems.

1.2.3 Components of a UNIX system

This section describes some of the components of the UNIX system. It will give you the background you need when working with computer systems. Figure 1.3 illustrates the relationship between the hardware, operating system (UNIX) and users schematically.

Hardware

The hardware is the computer itself. But the hardwares do not serve as anything without the softwares, including the operating system. UNIX is of course one of these softwares, and its

¹About the network systems we will mention in the later classes. However since network system is highly complicated and huge, it will be impossible to explain the whole figure of it, but only the introductory operation to it.

Table 1.1. UNIX compared with other operating systems.

Operating System	Different Manufacturers?	Hardware Required	Primary Multiuser?	Multitasking?	Market
MS-DOS	Yes	IBM PC or compatible	No	No	PCs
OS/2	Yes	IBM PC or compatible	No	Yes	PCs
Windows NT	No	Various	No	Yes	PCs
PICK	Yes	Various systems	Yes	Yes	Small business
VMS	Only DEC architecture	DEC VAX	Yes	Yes	Minis and superminis
MVS	Yes, mostly IBM	IBM mainframe	Yes	Yes	Mainframes
UNIX	Yes	Various	Yes	Yes	PCs to mainframes

foundation can be classified into two parts: kernel and shell.

Kernel

The kernel is the central core of UNIX and is named for the inner seed of a nut. The kernel is where the computer's activities are coordinated and controlled. After you boot² (start up) a UNIX computer, its first major job is to load the kernel on the main memory. The kernel remains in the machine's main memory until you shutdown (turn off) the computer. The kernel controls every facet of the hardware's operation and acts as a protective layer surrounding the hardware. User's programs can communicate with the hardware only by using the kernel as intermediary.

Shell

The shell is a program that acts as the user interface to the kernel. The shell is the part that you actually see and use. Usually shells are command line interfaces which means you can execute every commands only by typing its name from the command line.

There are many varieties in shells, but basically it consists of two kinds; sh (Bourne shell) and csh (C shell). There is a comprehensible explanation about shells from p.21 in the yellow textbook, so if you have more interest, please consult it. In the classes here, we utilized the most commonly used shell, C shell. "C" is named because the grammar of the shell program is quite similar to the C language.

²"boot" is a technical term used in the computer industry which means to *start up*.

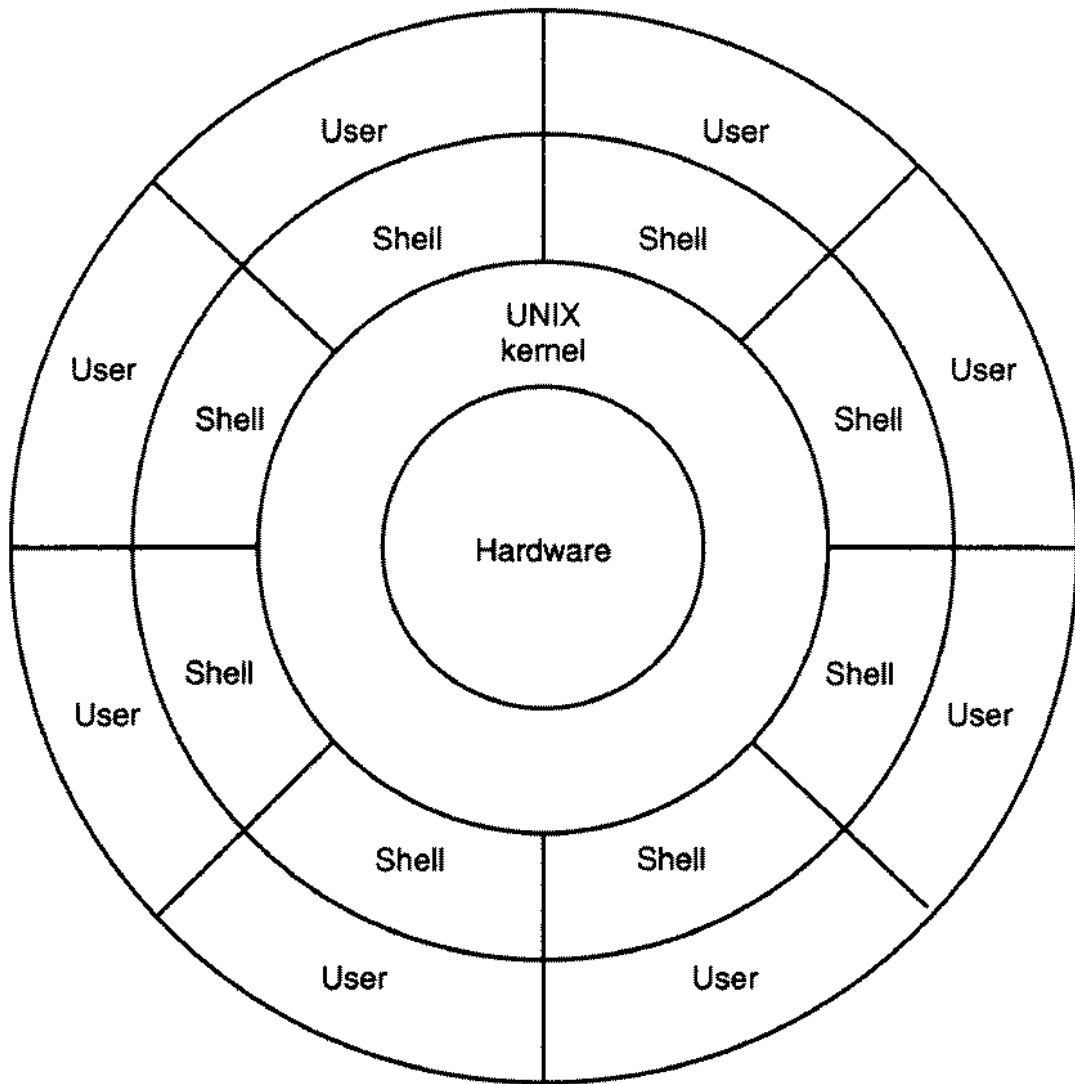


Figure 1.3. Schematic illustration of the relationship between the hardware, operating system (UNIX) and users.

Chapter 2

Login & Logout

2.1 Starting up a system

Unlike PCs, usual UNIX workstations such as SPARC Station are almost always working; 24 hours per day, 365 days per year. Of course sometimes they should be shutdown (turned off), but basically you should be aware that **you don't have to turn on or off the machine** unless you are a system manager. Machines in the exercise room are also working every day and night. Anytime you can come here and login the machines as long as the room is open.

2.2 Entering your login information and password

2.2.1 login information

Every UNIX user has a user name (or login name) and a password. Login name is the name of you in the machines. System administrators has set up an account just for you on the UNIX system, using your login name. UNIX keeps a record of each time you use your account. Unless you are starting up (*booting*) the systems, the only thing you see on your display will be

```
login:
```

The cursor is now located where you type in your login name. After typing your login name, press Return. For example

```
login: fkokus03 ↵
```

Here symbol ↵ means that you pressed the Return key.

2.2.2 Entering your password

Then, the UNIX system responds by asking you for your password, as in this example,

```
login: fkokus03 ↵
Password:
```

Password is very important in using UNIX system. It is a only key for the machine to recognize a user (who entered a login name) as you! If your password is known to someone with malice, he poses as you, login, and might do some malicious things like deleting files, changing password.

All of your passwords is **uitec94** with which you can login the machines. Essentially it is necessary for you to change your initial password immediately, but because of the local reasons in this university here, it is impossible to change it (sorry!). So you should use the password “**uitec94**” during the whole term.

Then, type your password. Notice that as you type your password, the screen does not display what you type. This prevents unauthorized access to your account by others who may be watching your screen. If you make a mistake as you type in your password, press **Control + u** and retype the password from the start. Backspace or delete key is no more available in this case.

When you have typed your password correctly, press Return. Perhaps the following display will appear;

```
SunOS Release 4.1.2-JLE1.1.2 (SM0) #6: Fri Sep 17 19:42:52 JST 1993
%
```

The percent sign (%) is the shell prompt and indicates that UNIX has accepted your access information and is ready for a command.

If, instead of the shell prompt, you are presented with this message:

```
Login incorrect
```

You may have simply made a mistake typing in your login name or password. Try the access procedure again. System administrator or instructor should be notified if repeated attempts prove unsuccessful.

2.2.3 logout procedure

When you finished your work on the machines, you have to logout. It is especially important to log off the UNIX system properly. Most systems require a single command at the shell prompt to log off, such as

```
% logout ↵
```

or

```
% exit ↵
```

Another alternative is pressing **Control + d**. Once you have successfully logged out the system, again you see the UNIX login message on-screen:

```
login:
```

Notice: Don't turn off the machines!

2.3 Fundamental operations of X-Window System

2.3.1 Starting up

To utilize the UNIX system more efficiently, window systems are necessary. We can use “X-Window System” (made in MIT, USA) on the workstations in the exercise room. The ability of multitasking is fully demonstrated when we use the window system. Anyway, please try;



Here, `xinit` is an abbreviation of INITializing X-window system.



Figure 2.1. login window (1)

2.3.2 Start the window manager

When X-Window system starts, a small box will appear on the upper-left corner of the display as in Figure 2.1. This is called the “login window”. Then, start the “window manager” `twm` to ease the window operation. If you don’t start the window manager, you cannot move, resize, or iconize the window (try later). To start the window manager, move the mouse pointer into the login window and type as in Figure 2.2.

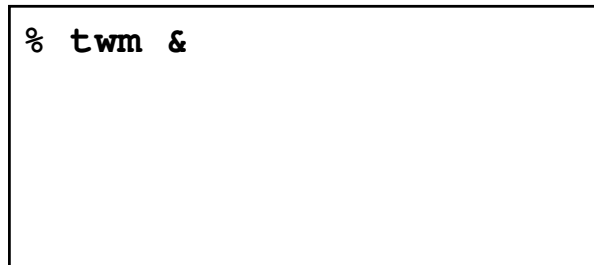


Figure 2.2. login window (2)

Do not forget to input `&` after the command name `twm`. `&` means that the command `twm` works as a background job (mentioned later). Then, a bar is displayed above the login window as in Figure 2.3.

2.3.3 Start the xterm

Then, open a working window “`xterm`” which is the abbreviation of X-TERMinal. `xterm` is one of the standard command which offers you the working environment on X-Window system (Figure 2.4).



Figure 2.3. login window (3)

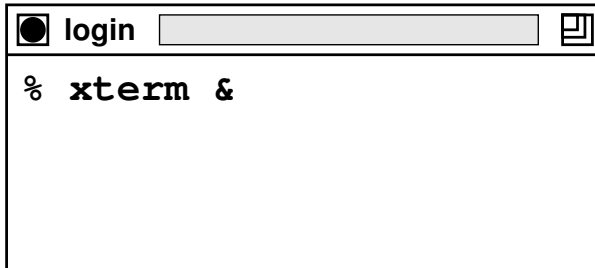


Figure 2.4. login window (4)

When a new window is displayed, only its frame is shown by the dotted lines as in Figure 2.5. You should move it to a desired position using the mouse, and click the left button to fix the window. The window, when fixed, will fully appear as shown below (Figure 2.6). You can then perform various operations in such windows, To enter a command, for example, move the mouse pointer into the window to use, and type the command from the keyboard. Notice that the commands cannot be entered if the pointer is out of the window!

Click the left mouse button to fix the window location

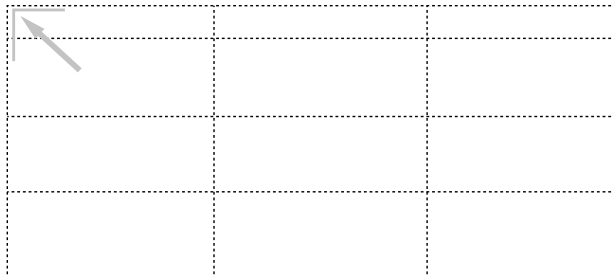


Figure 2.5. When a new window is displayed, only its frame is shown by the dotted lines.

If you start `xterm` with the option `-sb` *i.e.*



a window with an scroll bar leftside is produced. Scroll bar is very useful in such a case when you see long text files (mentioned later).

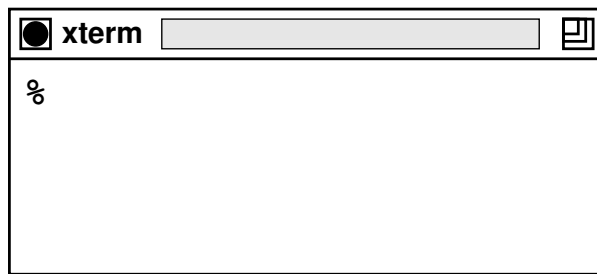


Figure 2.6. Whole figure of xterm

2.3.4 Window operations

Window operations, such as moving, resizing, iconizing, can be done mainly by using the mouse. The operational procedures are outlined below. See also Figure 2.7.

Move a window

To move a window, move the mouse pointer into the bar in the upper part of the window, and draw the mouse while holding down the left button. Release the left button when the window is at the desired position.

Resize a window

To resize a window, move the mouse pointer into the symbol at the upper right corner of the window and draw the mouse while holding down the left button. Release the left button when the window has the desired size.

Iconize a window

To iconize a window, click on the symbol at the upper left corner of the window once. Then the window is reduce to an icon. To restore the icon to the original window, click on the icon again.

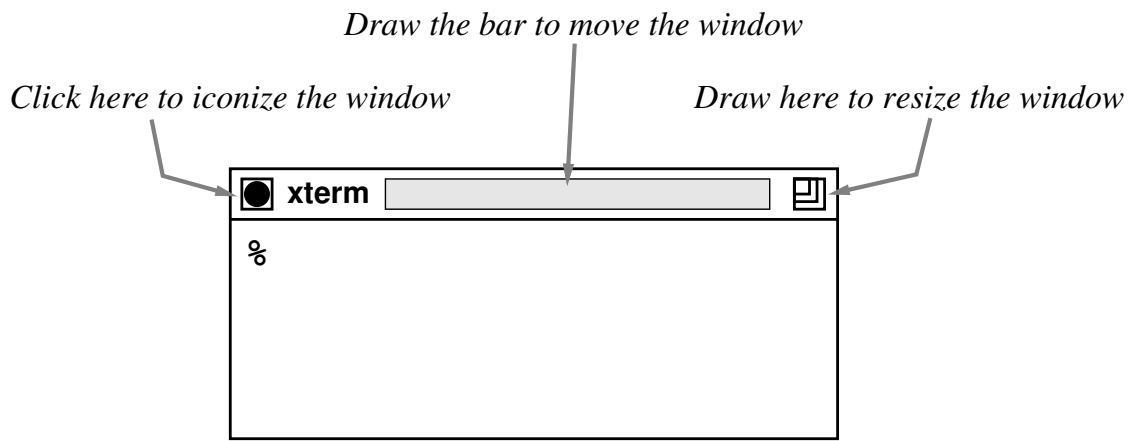
2.3.5 Closing the X-Window system

To close a window, type **exit** on the windows. And to close the whole X-Window system, type **exit** on the login window **after you have closed all the windows you have opened.**

% exit

Then you will logout from the UNIX system automatically, and the only thing you see on your display again will be

login:



xterm *This is the icon: Click on the icon to restore the window*

Figure 2.7. Window operation. Moving, resizing and iconizing.

Today's Check List

1. Login. Type your login name and password correctly.
2. Logout.
3. Login again, and start the X-window system with `xinit`.
4. Start up the terminal window manager `twm`.
5. Open a xterm window `xterm` (window (a)).
6. Open a xterm window from the window (a) with scroll bar `xterm -sb` (window (b)).
7. Iconize the window (b).
8. Restore the window (b).
9. Move and superpose the window (b) over the window (a).
10. Popup the window (a) on the window (b).
11. Popup the window (b) on the window (a).
12. Enlarging the window (b).
13. Reducing the window (b) (This may be somewhat puzzling).
14. Close all the windows. Note the turn of closing.

Chapter 3

Files & Directories

3.1 Everything in UNIX is a file!

- Ordinary file
- Directory file
- Special file

3.2 Directory tree

The UNIX file system is organized in what is called an upside-down tree structure.

- Home directory
- Current directory
- Parent directory

3.3 Directory referencing

- Absolute referencing
- Relative referencing

Today's Check List (1)

1. Login. Type your login name and password correctly.
2. Start the X-window system with `xinit`.
3. Start up the terminal window manager `twm`. Don't forget to append `&` after `twm`.
4. Open a xterm window from the window with scroll bar `xterm -sb`. Don't forget to append `&` after `twm`.
5. Confirm the name of your home directory by `echo ~`.
6. Check the name of your current directory by `pwd`.
7. Check your current position in Figure 3.1, and fill the directory names expressed by blank ovals.
8. While consulting Figure 3.1, and using *absolute* path,
 - (a) Change current directory to root (`/`).
 - (b) Change current directory to `/usr`.
 - (c) Change current directory to `/usr/lib`.
 - (d) Change current directory to your home directory.
9. While consulting Figure 3.1, and using *relative* path,
 - (a) Change current directory to the parent (`..`) directory.
 - (b) Change current directory to the parent directory again.
 - (c) Confirm where you are now, and `cd` to `/home`.
 - (d) Change current directory to `/etc`.
 - (e) Change current directory to `/root`.
 - (f) Change current directory to `/usr`.
 - (g) Change current directory to `/usr/lib`.
 - (h) Change current directory to root again.
 - (i) Be back to your home directory.

Then, explore in the directory tree as you like! Please don't forget to check where you are now by `pwd` command. If you would like to know the content of the directory, use the command `ls` (about `ls`, we will study in detail later).

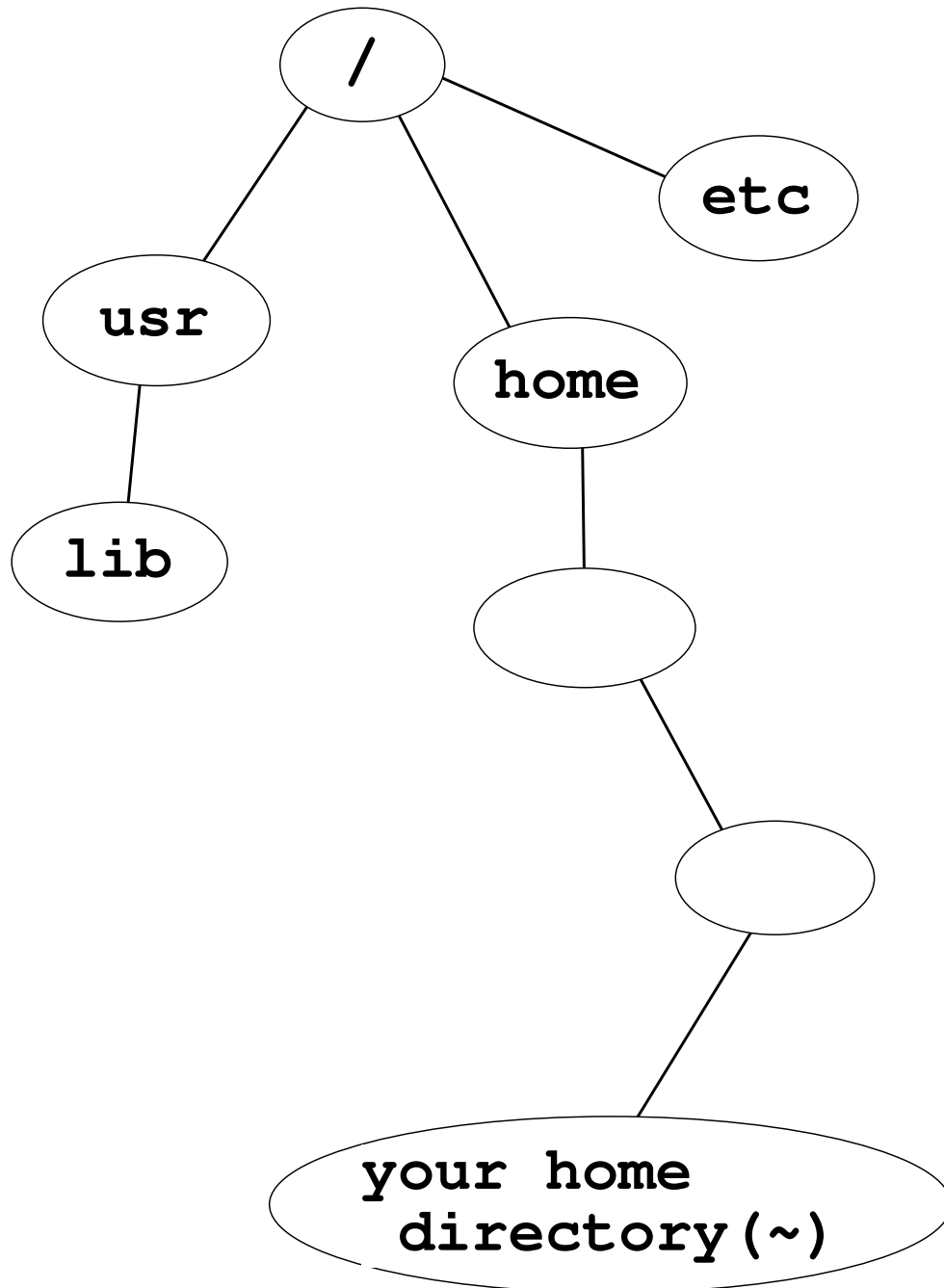


Figure 3.1. Directory tree in this system. Check your current position in this figure, and fill the directory names expressed by blank ovals.

Chapter 4

Permissions

In an office there are files which certain people don't have to see or should not see. To protect these from being seen by someone who should not, the drawers of the filing cabinet are locked. UNIX must provide the same kind of security to the files on its file system. The locks in UNIX are called *permissions*. Permissions are used to grant or refuse access privileges to files for particular users.

4.1 Concept of permissions

- Read (**r**)
- Write (**w**)
- Execute (**x**)

4.2 Using `ls -l` command

4.3 Directory permissions

- Read (**r**)
- Write (**w**)
- Execute (**x**)

Today's Check List (2)

1. Change directory to `~tito/Nov17`.
2. Confirm the permissions of the files there by `ls -l`.
3. Try to execute the executables.

Notice that `~tito` expresses the home directory of user `tito`. Similarly, `~fkokus01` expresses the home directory of user `fkokus01`.

Then, explore in the directory tree as you like, and confirm the permissions of the files there by `ls -l`. Please don't forget to check where you are now by `pwd` command.

4.4 Change permissions

Absolute (numeric) method

(ex.)

```
chmod 644 filename
```

Relative (symbolic) method

(ex.)

```
chmod g+w filename
```

Check List about Permissions

1. Change directory to your home directory.
2. Make a directory Nov17 by `mkdir Nov17`
3. Copy all files in `~tito/Nov17` into the above directory Nov17 by
`cp ~tito/Nov17/* ~/Nov17`
4. Change the permission of `hello` to 644.
5. Try to execute `hello`.
6. Add the execute permission of owner to `hello`.
7. Change the permission of `hello.c` to 000.
8. Try to read `hello` by using `cat` command (`cat hello.c`).
9. What should we do to read `hello.c` again?
10. Read the online manual of `chmod` (`man chmod`).

Notice that `cp` is a command which corresponds to MS-DOS `copy`, which copies file. Similarly, `cat` corresponds to MS-DOS `type`, which show the content of file. `*` (asterisk) is called wild-card, which means “all matched files”, and this is also a same concept as in MS-DOS. So the command “`cp ~tito/Nov17/* ~/Nov17`” means “**Copy all the files in `~tito/Nov17` to `~/Nov17`**”.

4.5 Some other topics

4.5.1 Options of ls

- **-l**
Show in the long format.
- **-lg**
Show in the long format with **group**.
- **-F**
Make directory a trailing **/**, executable files with a trailing asterisk *****, symbolic links with a trailing at-sign **@**. This option is very much useful.
- **-t**
Sort by time modified, instead of by name.
- **-a**
List all entry; in the absence of this option, files whose names begin with dot (**.**; hidden files) are not listed.

Several options can be executed at a time.

```
% ls -l -g -F
```

is equal to below.

```
% ls -lgF
```

4.5.2 UID and GID bit programs

(This is an advanced concept and unimportant for now)
Some files have **s** instead of **x** in the permission column.

4.5.3 Symbolic links

Symbolic linking is to give another name, or another interface to the same file. The enigmatic phenomena in the last week are all due to this symbolic links.

- To share the same file with another person.
- To save the disk space to do link instead of copying.

Check List about `ls` and `chmod`

1. Change directory to root (/).
2. Execute `ls` to show the modes of the files there.
3. Execute `ls -l` to show the modes of the files there.
4. Execute `ls -lg` to show the modes of the files there.
5. Execute `ls -F` to show the modes of the files there.
6. Execute `ls -t` to show the modes of the files there.
7. Execute `ls -lt` to show the modes of the files there. What is the difference between `ls -l` and `ls -lt`?
8. Execute `ls -a` to show the modes of the files there.
9. Execute `ls -lgFta` to show the modes of the files there.
10. Execute `ls -l -l -F -t -a` to show the modes of the files there.
11. Try to change the permission of the file `vmunix`.
12. Try to find the `UID` or `GID` programs. Who is the owner of them, and what is the group of them?
13. Read the online manual of `ln` carefully (`man ls`). What kinds of options of `ls` are there?

Check List about symbolic links

1. Change directory to you home (~).
2. Make a directory Nov24 by `mkdir Nov24`.
3. Copy the files `~tito/Nov24/testfileA` into the above directory Nov24 by
`cp ~tito/Nov24/testfileA ~/Nov24`
4. Change directory to `./Nov24`
5. Make a symbolic link of `testfileA` as the name of `testfileA.link` by
`ln -s testfileA testfileA.link`
6. Execute `ls`, `ls -lg` and `ls -F` to show the mode of the file.
7. Typeout the content of `testfileA` by
`cat testfileA`
8. Typeout the content of `testfileA.link`. What is the difference between `testfileA` and `testfileA.link`?
9. How are the permissions of symbolic link file?
10. Then, explore in the directory tree as you like! Please don't forget to check where you are now by `pwd` command.

Notice that the permission column of a symbolic file is always `lrwxrwxrwx`. However, it does not mean that the file is accessible (readable, writable or executable) by any user. The actual permission of a symbolic link file is determined by the original one (real body), so the expression of `lrwxrwxrwx` has no meaning in practice.

Chapter 5

Other topics on UNIX filesystems

5.1 Restrictions of file name

There are hardly any restrictions on UNIX file name!

- *Maximum length*: 255 characters (in SunOS)
- *Extensions*: arbitrary
- *Special characters*: OK

Even a blank can be used in a file name.

Notice that the UNIX filenames are case-sensitive. On the other hand, MS-DOS file names are case-insensitive.

5.2 Hidden files

Filenames of hidden files start with dot (.). These are used by applications to store user specific setup information. For example,

- `.cshrc` (by `cs`h)
- `.xinitc` (by `xinit`)

5.3 Wildcards

Wildcards in UNIX are almost the same as those of in MS-DOS.

- `?` matches any single character
- `*` matches anything at all

5.4 Important commands

(See also *p.327* in Yellow textbook)

- `cat` (from “conCATenate”)
- `cp` (from “CoPy”)

- mv (from “MoVe”)
- rm (from “ReMove”)
- wc (from “Word Count”)
- lpr (from “Line PRinter”)
- echo
- more
- head
- tail
- date

Check List on various commands


1. Change to your home directory.
2. Confirm what kind of hidden files are there in your home directory by `ls -a`.
3. Try to execute `ls` command with various options. Compare the results.
4. Copy all files in `~tito/Nov24` into the above directory `Nov24` by
`cp -r ~tito/Nov24/* ~/Nov24`
This takes somewhat long time. The option `-r` means *recursively* (copy all the subdirectories too).
5. Consult the on-line manuals of commands `cat` by `man cat`
6. Consult the on-line manuals of commands `cp` by `man cp`
7. Consult the on-line manuals of commands `rm` by `man rm`
8. Consult the on-line manuals of commands `mv` by `man mv`
9. Consult the on-line manuals of commands `more` by `man more`
10. Consult the on-line manuals of commands `head` by `man head`
11. Consult the on-line manuals of commands `tail` by `man tail`
12. Consult the on-line manuals of commands `date` by `man date`

There are various kinds of junk files and junk directories in `~tito/Nov24`. Try to execute each of commands. Example of execution are follows:

```
cat testfileA
cp hello testfile
mv testfile testfile2
rm testfile2
cat inet.services.txt
sort inet.services.txt
more inet.services.txt
head -40 inet.services.txt
tail -40 inet.services.txt
date
lpr -p listserve.refcard
wc -l listserve.refcard
```

Then, explore in the directory tree as you like, and try to execute above commands in each directories. Notice that in the directories in which you don't have the write permission, writing command such as `cp` or `rm` does not work.

NOTICE There are several Japanese text files in the directory `~tito/Nov24` or in the system directories. You cannot see the content of them because `xterm` cannot be designed to handle the Japanese characters. If you would like to see them, use `kterm` (Kanji TERMinal) instead of `xterm` as

```
% kterm -sb 
```

and execute the commands such as `cat` or `more`.

5.5 Redirection in UNIX

The results of UNIX commands are usually displayed on-screen. This is called *standard output*. Similarly, inputs of UNIX commands are usually typed in from keyboard. This is called *standard input*. “Redirection” means, as long as in the case of MS-DOS, as if you say to UNIX

“Don’t display this output on-screen — instead, put it somewhere else.”

or

“The input for this command is not coming from the keyboard this time, so look for it somewhere else.”

Of course, *somewhere else* means the files.

5.5.1 Redirecting output (using >)

In some directory, type

```
% ls -l > file
```

Then the output will not be displayed on-screen. To confirm the results, type

```
% cat file
```

You can see the results of the command `ls`.

Overwriting and appending

If *file* already exists, `> file` overwrites the content of it completely. If you would like to append to the end of *file*, use `>>` instead of `>`.

```
% ls -l >> file
```

5.5.2 Redirecting input (using <)

Redirecting input is often useful than redirecting output. One of the typical examples is the case when you send E-mail to someone;

```
% mail someone < file
```

Then the content of the file is sent to *someone* by E-mail. About E-mail we will learn in the later classes.

echo and redirect

The C-shell command `echo` is often used with the redirecting operation. `echo` writes its arguments on the standard output, so we put the output into file using `>`. For instance,

```
% echo 'This is the output of echo.' > file
```

5.6 Piping in UNIX

It can be really useful to redirect the output of one program (command) so that it becomes the input of another program. This process is called *piping* as same as in the case of MS-DOS. The syntax of piping is

```
commandA | commandB
```

Here, the output of `commandA` is used as the input of `commandB`. Piping is very useful when you use several UNIX commands in combination.

```
% ls -l | more
```

```
% ls -l | sort
```

```
% echo 'How are you?' | rev
```

Combination of redirection and pipe

Piping and redirection are of course able to be used in combination.

```
% ls -l | cat -n > file
```

Check List on redirecting and piping

1. Change to ~/Nov24
2. Try
`ls -lg > ls.result`
3. Confirm the content by `cat ls.result`
4. Append to the file by `ls -lg >> ls.result`
5. Confirm the content by `cat ls.result`
6. Overwrite to the file by `ls -lg > ls.result`
Then what will happen?
7. Try
`echo 'How are you?' | rev`
8. Then, check the on-line manual for the information about command `rev`. What does it work for?
9. Try
`ls -l | cat -n > ls.cat.result`
and then
`cat ls.cat.result`
10. Check the on-line manual of `cat` and consult the meaning of `-n` option.
11. Explore in the directory tree as you like, and try to execute above commands in each directories. Notice that in the directories in which you don't have the write permission, writing command such as using `>` or `>>` does not work.

When you are in the directory which contain a lot of files, the output of `ls` command can go whizzing by too fast to read, which makes impossible to see the files at the begging of the list before they disappear off the top of the screen. In such a case, the combination of `ls` and `more` can help you. Compare the two results following.

1. Change to `/etc`
2. List the content of the directory by
`ls`
3. List the content of the directory in the long format by
`ls -lg`
4. List the content of the directory by using `more`
`ls -lg | more`

The combination of `ls` and `more` may be the most frequently used in the UNIX world.

5.7 alias (csh command)

Make an alias

You can invent a short name for a long name command by **alias** command. An operation

```
% alias shortname originalname
```

enables you to utilize *shortname* instead of *originalname*. For example,

```
% alias dir ls -a  
% alias ll ls -lgF
```

alias with no argument lists all the alias entries at present.

```
% alias
```

Remove an alias

To remove aliases, use **unalias** command.

```
% unalias dir  
% unalias ls
```

unalias * deletes all alias entries.

5.8 history (csh command)

History listing

A command `history` is utilized to list several commands you have typed before.

```
% history
```

Then the result may be following form:

```
1 re
2 refile cur +ymnet ; next
3 refile cur +ten ; next
4 refile cur +planet ; next
5 re
6 refile cur +ymnet ; next
7 rmm ; next

( ... omitting ... )

63 next
64 refile cur +ls/ast ; next
65 vm
66 st +ls/ast
67 show 98
68 rmm
69 ll
70 vm
71 st +panda
72 show 3186
73 show 2931
74 vm
75 ll
76 ll
77 ll
78 which eriot
79 eriot m-vos
80 vm
```

Re-execution of the previous command

```
% !!
```

This command re-executes the previous command. For example, when you have done the succession of commands as

```
% cp testfile1 test3
% date
% ls -lgFt
```

and then enter `!!`. Then, the previous command

```
% ls -lgFt
```

is re-executed.

Re-execution of the n -th command

History list which is shown by `history` has the sequence number for each command. You can specify the old command by this number, for example,

```
% !78
```

is completely equivalent to the command

```
% which eriot
```

in the list above.

Re-execution of the command containing *string*

The command

```
% !string
```

re-executes the **latest** command which contains *string* in it. For example,

```
% !show
```

matches the 73-th command

```
% show 2931
```

, not 72-th command `show 3186` because the 73-th command is the latest one which contains the string *show*.

Check List on alias and history

1. Confirm what kind of aliases are defined now on your shell by `alias`
2. Try to define some aliases. Recommended ones are:
`alias ls ls -CF`
`cp cp -i`
`h history`
`home cd ~`
`rm rm -i`
3. Confirm the present list of history by `history`
4. Try to execute the command `!!`.
5. Try to execute some kinds of `!number` or `!string` as you like.

5.9 finger and talk

finger

finger are one of the most major commands by which you can find out who is using the local or remote machines. The simplest way is just to type

```
% finger
```

By default, finger displays information about each logged-in user, including his or her: login name, full name, terminal name, idle time, login time, and location if known. You can get an information such like below

Login	Name	TTY	Idle	When	Where
ai1	???	co		Wed 16:21	
ai1	???	p0	13	Wed 16:21	:0.0
ai1	???	p1		Wed 16:22	:0.0
s2muraka	???	p3	1d	Tue 14:55	csx0:0.0
s2muraka	???	p4	23:	Tue 14:56	csx0:0.0
tito	???	p5	43	Wed 15:51	133.40.8.42

Users who is now logging in the local (you are using now) are shown in the leftside column. If your machine is on a network, following command will work.

```
% finger @machine
```

This command tells you who is logging in *machine*.

talk

talk is a visual communication program which copies lines from your terminal to that of another user. If you wish to talk to someone on your own machine, then username is just the person's login name (which you have to search by **finger**). If you wish to talk to a user on another host, then username must be following form:

```
% talk user@machine
```

When first called, talk sends the message such like:

```
Message from TalkDaemon@your_machine at time...
talk: connection requested by his_name@his_machine.
talk: respond with: talk his_name@his_machine
```

to the user you wish to talk to. At this point, the recipient of the message should reply by typing:

```
% talk his_name@his_machine
```

Once communication is established, the two parties may type simultaneously, with their output appearing in separate windows. Typing CTRL-L redraws the screen, while your erase, kill, and

word kill characters will work in talk as normal. To exit, just type your interrupt character (CTRL-C); talk then moves the cursor to the bottom of the screen and restores the terminal.

Check List on finger and talk

1. Confirm who is logging in your local machine by
finger
2. Confirm who is logging in remote machines by
finger @csa0
finger @csa1
finger @csa2
finger @csa3
finger @csa4
finger @csa5
Or, at a sitting,
finger @cse0 @cse1 @cse2 @cse3 @cse4 @cse5
3. Try to talk to one of the users who is logging in now. For example,
talk fkokus10@cse5
talk tito@csa0

5.10 Background jobs

5.10.1 background jobs and foreground jobs

Background execution means to run a UNIX program that doesn't interact with the screen or keyboard. You use background execution to perform tasks that require significant time but that don't interact with you, such as sorting a file or formatting a file for printing, or huge numerical calculation for scientific purposes. Also, programs which should be always working when you are logging in, such as window manager (`twm`) or window terminal emulator itself (`xterm` or `kterm`), should be executed as background jobs.

On the other hand, foreground jobs are the characteristic of a program that can accept input from the keyboard and display data on-screen. By default, UNIX programs are run in the foreground. Programs that run in the foreground are also said to be *interactive*.

To run a job in background, put `&` at the end of command:

```
% command &
```

5.10.2 Job control: foreground, stopped, background

You may notice that, many times, you run a job, and realize that it is going to take longer than you thought, and decide that you want to switch it to a background job. In such cases, you can change the status of you job by following operations. Relation between the job status and control commands are shown in Figure 5.1.

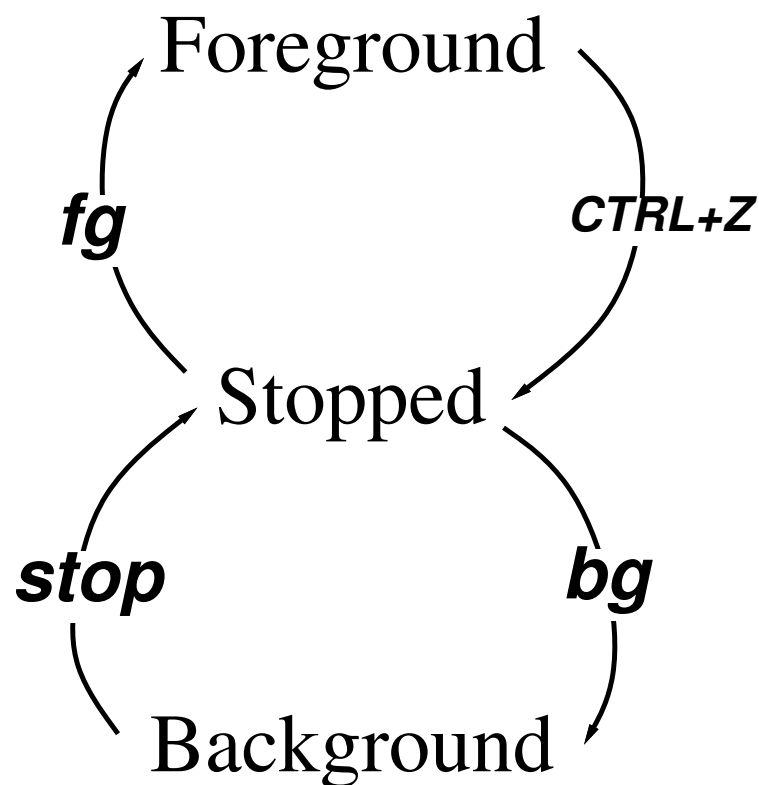


Figure 5.1. Schematic illustration of job status and controlling commands

To make foreground job stop: CTRL + Z

Suppose you run a program in foreground, and intend to bring it into background. In this case, you have to first stop the foreground job by pressing `control + Z`.

To list the status of jobs: jobs

To list the status of jobs on your present shell, `cs`h command `jobs` works. Following results will appear.

```
% jobs
[1]  + Stopped          vi text.tex
[2]  - Stopped          xterm -sb
[3]  - Running          kterm -km euc
```

Here, [1] denotes the job number, +/- sign denotes the job priority (not important here), Stopped/Running shows the job status. Name of the jobs are listed in the rightside column.

To make stopped job run in a background: bg

When you want to make a stopped job run in foreground, the `cs`h command `bg` (abbreviation of BackGround) can work. For example, when you would like to make job 2 run in background, specify the job number by `%job_number`.

```
% bg %2
```

To stop a background job: stop

When you want to stop a background job, simple `cs`h command `stop` will work. For example,

```
% stop %3
```

To make stopped job run in a foreground: fg

When you want to make a stopped job run in foreground, the `cs`h command `fg` (abbreviation of FackGround) can work. For example, when you would like to make job 3 run in background,

```
% fg %3
```

To kill (terminate) a background job: kill

When you want to kill (or terminate; not stop) a background job, `cs`h command `kill` will work. For example,

```
% kill %2
```

5.11 Showing processes

Process is a general term for all of users jobs (background and foreground) and system jobs. We can see what processes are running on the machine by **ps** (Process) command. **ps** command has a lot of options, but many of them are useless. Only we should know is **-aux** or **-auxw**. About the results of **ps** and the meaning of option **a**, **u**, **x** or **w**, consult the on-line manual of **ps**. A sample of typical results of **ps** is shown below.

```
% ps -aux
```

USER	PID	%CPU	%MEM	SZ	RSS	TT	STAT	START	TIME	COMMAND
root	0	0.0	0.0	0	0	?	D	Nov 21	0:32	swapper
root	1	0.0	0.0	52	0	?	IW	Nov 21	0:03	/sbin/init -
root	2	0.0	0.0	0	0	?	D	Nov 21	0:01	pagedaemon
root	165	0.0	0.0	1584	0	?	IW	Nov 21	44:33	/home/LOCAL/X11R5/bin/X
root	54	0.0	0.5	72	108	?	S	Nov 21	1:44	portmap
root	107	0.0	0.0	52	0	?	IW	Nov 21	0:00	rpc.bootparamd
root	59	0.0	0.0	40	0	?	IW	Nov 21	0:00	keyserv
root	70	0.0	0.0	668	0	?	IW	Nov 21	1:53	in.named
root	87	0.0	0.0	68	0	?	IW	Nov 21	0:28	syslogd
root	73	0.0	0.0	16	0	?	I	Nov 21	0:05	(biод)
root	95	0.0	0.0	136	0	?	IW	Nov 21	0:05	-accepting connections (
root	113	0.0	0.0	28	0	?	I	Nov 21	0:10	(nfsd)
root	101	0.0	0.0	60	0	?	IW	Nov 21	0:01	rpc.mountd -n
root	102	0.0	0.0	28	0	?	I	Nov 21	0:10	(nfsd)
root	104	0.0	0.0	36	0	?	IW	Nov 21	0:00	rarpd -a
root	105	0.0	0.0	24	0	?	IW	Nov 21	0:00	rarpd -a
root	109	0.0	0.0	52	0	?	IW	Nov 21	0:00	rpc.statd
root	114	0.0	0.0	28	0	?	I	Nov 21	0:11	(nfsd)
root	111	0.0	0.0	96	0	?	IW	Nov 21	0:00	rpc.lockd
root	134	0.0	0.0	12	4	?	I	Nov 21	19:01	update
tito	2590	0.0	1.5	60	344	p2	R	01:05	0:00	mail tito
root	129	0.0	0.0	80	0	?	IW	Nov 21	0:00	/usr/etc/snmpd -c /etc/s
root	137	0.0	0.0	176	0	?	IW	Nov 21	1:09	cron
root	150	0.0	0.0	56	0	?	IW	Nov 21	0:13	inetd
root	153	0.0	0.0	52	0	?	IW	Nov 21	0:00	/usr/lib/lpd
root	155	0.0	0.1	16	28	?	S	Nov 21	0:11	screenblank -d 120
kajino	1068	0.0	0.0	80	0	p7	IW	17:03	0:00	-sh (csh)
root	163	0.0	0.2	228	56	?	S	Nov 21	0:07	/usr/bin/X11/xdm
root	166	0.0	0.0	40	0	co	IW	Nov 21	0:00	- std.9600 console (gett
root	27831	0.0	0.0	248	0	?	IW	Nov 29	0:07	-pius:0 (xdm)
kajino	296	0.0	0.0	420	0	?	IW	10:33	1:14	kterm -sb -iconic -d ggt
greg	28206	0.0	0.0	72	0	p6	IW	Nov 29	0:00	-csh (csh)
root	302	0.0	0.0	60	0	?	IW	Nov 21	0:11	rpc.rquotad
tsuchi	7523	0.0	0.0	0	0	?	Z	Nov 21	0:00	<defunct>
wnn	27895	0.0	0.0	2088	0	?	IW	Nov 29	0:17	jserver
kajino	1067	0.0	0.0	420	0	?	IW	17:03	0:38	kterm -sb -iconic -d ggt
kajino	1062	0.0	0.0	56	0	?	IW	17:03	0:00	in.rshd
sano	2200	0.0	0.0	288	0	p1	IW	22:09	0:02	-csh (tcsh)

tsuchi	7519	0.0	0.0	44	0 ?	IW	Nov 23	0:00	rsh milano fvwm -display
root	26769	0.0	0.0	208	0 ?	IW	Nov 29	0:00	xconsole -geometry 480x1
root	28205	0.0	0.0	44	0 ?	IW	Nov 29	0:00	in.rlogind
kajino	1063	0.0	0.0	76	0 ?	IW	17:03	0:00	ssh -c kterm -sb -iconic
tito	2564	0.0	3.0	224	672 p2	S	01:04	0:00	-usr/local/bin/tcsh (tcs
sano	2196	0.0	0.0	408	0 ?	IW	22:09	0:05	kterm -n ferio
tito	2589	0.0	1.9	196	436 p2	R	01:05	0:00	ps -aux
kajino	292	0.0	0.0	76	0 ?	IW	10:33	0:00	ssh -c kterm -sb -iconic
root	2316	0.0	0.0	248	0 ?	IW	22:46	0:00	-isolde:0 (xdm)
kajino	297	0.0	0.0	100	0 p3	IW	10:33	0:02	-sh (ssh)
root	26763	0.0	0.0	248	0 ?	IW	Nov 29	0:00	:-0 (xdm)
kajino	291	0.0	0.0	56	0 ?	IW	10:33	0:00	in.rshd
root	2563	0.0	1.2	44	268 ?	S	01:04	0:00	in.rlogind

Check List on background/foreground jobs

1. Open a **kterm** window as a foreground job (without **&**)
`% kterm`
2. Stop the job of **kterm** by pressing **CTRL + z**
3. See the job status by **jobs**
4. Make the job run in background. Use **bg**
5. See the job status by **jobs**
6. Stop the job of **kterm** again by **stop** command.
7. See the job status by **jobs**
8. Make the job run again in foreground. Use **fg**
9. Stop the foreground job of **kterm** again by pressing **CTRL + z**, and see the job status by **jobs** .
10. Kill (terminate) the job of **kterm**. Use **kill**
11. Consult the on-line manuals of **bg**, **fg**, **stop** and **kill**. They are contained in the manual of **cs****h**, so simply
`% man cs`

Check List on ps

1. Execute the command **ps** with option **-aux**.
`% ps -aux`
2. Execute the command **ps** with option **-auxw**. What is the difference?
3. Execute the command **ps** with no option (simply **ps**). What is the difference?
4. List only your own processes by
`% ps -aux | grep your_login_name`
Then consult the on-line manual of **grep** to confirm its information.
5. Consult the on-line manuals of **ps** and confirm the meaning of each option.

Chapter 6

E-mail

Electronic mail (E-mail) is very useful way to send and receive messages by computers, and now widely used in science and business industry.

We use the UCB (University of California Berkeley) mail command in this class. UCB mail is a simple, but comfortable, flexible, interactive program for composing, sending and receiving electronic messages. While reading messages, mail provides you with commands to browse, display, save, delete, and respond to messages. While sending mail, mail allows editing and reviewing of messages being composed, and the inclusion of text from files or other messages.

Incoming mail is stored in the system mailbox for each user. This is a file named after the user in `/var/spool/mail`. When you read a message, it is marked to be moved to a secondary file for storage. This secondary file, called the `mbox`, is normally the file in your home directory (`~/mbox`).

6.1 Sending mails

6.1.1 The simplest way

The simplest way to send a mail to *someone* is as follows. First, you enter a command

```
% mail someone
```

where *someone* of course means the loginname of recipient. Then, you will be urged to input the subject of the mail.

```
Subject: █
```

Here, █ means the location of the cursor. You should input the subject, and enter. For example,

```
Subject: TEST 1
```

```
█
```

Next you should type in the main body of your mail. For example here,

```
Subject:  TEST 1
This is a test at Polytechnic University.
It was called 'uitec' formerly.
Bye.
□
```

When finished typing the mail body, enter CTRL + D to mark the end of input.

```
^D
```

That's all for sending.

6.1.2 Recipient

There are two ways about the expression of the recipient name. First, when you intend to send a mail to the local user, simple

```
% mail loginname
% mail fkokus08
% mail yokoyama
```

can work. Second, when you intend to send a mail to the remote user, name of the machine is also needed. For example,

```
% mail loginname@machine
% mail fkokus03@cse2
% mail fkokus07@csa1
```

6.1.3 Send a file

You can also send a text file by E-mail. In this case, you should use redirection of input we have studied before. First, you have to make a file by some text editors such as **vi** or **emacs**. Then, a simple way to use redirection of input

```
% mail someone < file
```

will work to deliver your messages (written in *file*) to recipient. In this case, subject is not added by default. To add a subject to your mail, use **-s** option of **mail** command. For example,

```
% mail -s Hello someone < file
% mail -s 'This is a simple subject.' someone < file
```

Don't forget to put the subject in quotations ' ' if the subject contains some special characters, such as space, tab, comma, or period.

6.2 Reading mails

If new mails are arriving, a message

You have new mail.

appears when you login the machine. In that case, you can read the messages by using mail command. Only you have to do is

% mail

Then, the information about sender, date, subject, status of mails are listed. For example

```
% mail
Mail version SMI 4.0 Mon Dec  2 20:17:48 JST 1991  Type ? for help.
‘‘/usr/spool/mail/tito’’: 5 messages 2 new 5 unread
U  1 satoh@clim      Wed Dec  7 13:51   78/4403  Re: PC or WS sending
   2 nakamoto        Wed Dec  7 13:56   12/288   We'll win the game!
N  3 panda@gpsun03   Wed Dec  7 14:08   19/933   request for help
>N 4 kino002         Wed Dec  7 14:20   16/392   Hard disk unit
U  5 jakky@lowtem    Wed Dec  7 14:30   42/1858  ftp site of transport
&
```

About the detail of each message, consult the on-line manual of mail. Character (U, N or empty) in the leftside column means

N ... The message is a New one.

U ... The message is not a new one, but still Unread.

(empty) ... You already read the message.

and the sign > denotes that it is the *current* message.

Here, the letter & denotes the command prompt in mail command. You can enter various subcommands after this &. All the subcommands are listed by typing ? after &. For example,

```
& ?
cd [directory]          chdir to directory or home if none given
d [message list]        delete messages
e [message list]        edit messages
f [message list]        show from lines of messages
h                        print out active message headers
m [user list]           mail to specific users
n                        goto and type next message
p [message list]        print messages
pre [message list]      make messages go back to system mailbox
q                        quit, saving unresolved messages in mbox
r [message list]        reply to sender (only) of messages
R [message list]        reply to sender and all recipients of messages
s [message list] file   append messages to file
t [message list]        type messages (same as print)
top [message list]      show top lines of messages
u [message list]        undelete messages
v [message list]        edit messages with display editor
w [message list] file   append messages to file, without from line
```

```
x          quit, do not change system mailbox
z [-]      display next [previous] page of headers
!          shell escape

A [message list] consists of integers, ranges of same, or user names separated
by spaces.  If omitted, Mail uses the current message.
&
```

Though there are many subcommands in `mail` command, only a few of them are important. What you should remember are probably following 6 only.

- & s ... Save the message in a file named `~/mbox` .
- & q ... Quit, saving unresolved messages in `~/mbox` .
- & x ... EXit, do not give any changes to your messages.
- & d ... Delete the message.
- & h ... Show the Headers again.
- & (*number*) ... Show the *n*-th message.

If you don't have any messages, a simple message such as

```
% mail
No mail for your_name
```

will be shown.

6.3 Structure of a E-mail message

An E-mail messages is structured very much like a paper letter — there is addressing information and salutatory material, like the return address and the date, and there is the actual message. Messages are divided into two parts; the system *header*, and the mail *body*. The header, at the top of the message, is the envelop. The body is the actual message. We show a very simple example first.

```
Return-Path: hiroshi
Received: by pluto.mtk.nao.ac.jp (5.67+1.6W/TISN-1.3/R2)
id AA00664; Wed, 7 Dec 94 17:47:12 JST
Date: Wed, 7 Dec 94 17:47:12 JST
From: Hiroshi Kinoshita <hiroshi>
Message-Id: <9412070847.AA00664@pluto.mtk.nao.ac.jp>
To: tito
Subject: Greetings

This is Hiroshi.
Bye!
```

Here, two lines at the bottom are the body, and the remaining part are headers. The header describes information about the sender, the messages, the route, and the recipient. Header ends with a blank line. For example, some common headers include:

Return-Path: ... The address where the reply should be sent.
Received: ... The machine where the message went through or arrived at.
Date: ... The date when the messages was sent.
From: ... The name of sender.
Message-Id: ... The ID number of the mail on the machine.
To: ... The recipient(s) of the message.
Subject: ... The subject of the message.

There are numerous kinds of headers in UNIX world, and it is permitted for us to invent new headers and put them on the mail message. For example, below is a mail I received recently. I myself can not understand the meaning of some headers in it.

```
Replied: Wed, 07 Dec 1994 10:38:17 +0900
Replied: nao-seminar@athena.mtk.nao.ac.jp
Return-Path: panda-admin@gpsun03.geoph.s.u-tokyo.ac.jp
Received: from geoph.geoph.s.u-tokyo.ac.jp by pluto.mtk.nao.ac.jp
(5.67+1.6W/TISN-1.3/R2) id AA09754; Tue, 6 Dec 94 17:02:15 JST
Received: from gpsun03.geoph.s.u-tokyo.ac.jp by geoph.geoph.s.u-tokyo.ac.jp
(8.6.8+2.4Wb/TISN-1.3M/R2) id QAA04783; Tue, 6 Dec 1994 16:58:35 +0900
Resent-From: panda-admin@gpsun03.geoph.s.u-tokyo.ac.jp
Received: from (localhost) by gpsun03.geoph.s.u-tokyo.ac.jp (4.1/TISN-1.2L/R1)
id AA10076; Tue, 6 Dec 94 17:00:25 JST
Resent-Date: Tue, 6 Dec 94 17:00:21 +0900
Resent-Message-Id: <9412060800.AA10076@gpsun03.geoph.s.u-tokyo.ac.jp>
Resent-Sender: panda@gpsun03.geoph.s.u-tokyo.ac.jp (Panda-Net mailing list)
Errors-To: panda-admin@gpsun03.geoph.s.u-tokyo.ac.jp
Old-Return-Path: <k2@jiro.eri.u-tokyo.ac.jp>
Received: from jiro.eri.u-tokyo.ac.jp by gpsun03.geoph.s.u-tokyo.ac.jp
(4.1/TISN-1.2L/R1) id AA10069; Tue, 6 Dec 94 16:57:55 JST
Received: from localhost (localhost [127.0.0.1]) by jiro.eri.u-tokyo.ac.jp
(8.6.5/3.3Wb) with SMTP id QAA20127; Tue, 6 Dec 1994 16:56:13 +0900
Message-Id: <199412060756.QAA20127@jiro.eri.u-tokyo.ac.jp>
X-Authentication-Warning: jiro.eri.u-tokyo.ac.jp:
Host localhost didn't use HELO protocol
X-Ecom-Version: 3.00.15 (PC98/PCTCP)
Mime-Version: 1.0
Content-Type: text/plain; charset=iso-2022-jp
Procedure: bulk
Newsgroup: page.general,page.misc
Posted: Tue, 06 Dec 1994 16:56:12 +0900
Reply-To: panda@gpsun03.geoph.s.u-tokyo.ac.jp
X-Ml-Server: nml.pl 1.2 (specially modified for Panda)
X-Ml-Name: Panda-Net mailing list
Lines: 126
To: panda@gpsun03.geoph.s.u-tokyo.ac.jp
Cc: page-general@eri.u-tokyo.ac.jp, eri_folks@eri.u-tokyo.ac.jp
From: Kazuki Koketsu <k2@jiro.eri.u-tokyo.ac.jp>
```

Date: Tue, 6 Dec 94 17:00:21 +0900

Subject: [Panda 3058] Workshop on Seismic Numerical Simulation

Workshop 'Numerical Simulation of Seismic Ground Motion'

Date : December 15, 1994 (Thu)

Time : 10:00am -- 4:30pm


Place : 2nd meeting room (2F, No.202),
Earthquake Research Institute, University of Tokyo

.....

6.4 xbiff

There are a useful command on X-Window System which can give notice of incoming mail messages, named **xbiff**. The **xbiff** program displays a little image of a mailbox. When there is no mail, the flag on the mailbox is down. When mail arrives, the flag goes up and the mailbox beeps. By default, pressing any mouse button in the image forces **xbiff** to remember the current size of the mail file as being the “empty” size and to lower the flag.

The usage is simple enough.

% **xbiff** & 

Don't forget to **run it as a background job** because the window of **xbiff** should be always working on the screen.

Check List on E-mail

1. At first, send an E-mail to yourself.
`% mail your_login_name`
2. Read it by
`% mail`
3. Then, stand the flag by
`% xbiff &`
4. Again, send an E-mail to yourself, and wait a little while.
5. Consult on-line manual of `xbiff` by
`% man xbiff`
6. Next, try to send E-mails to someone by direct method such as
`% mail someone`
or, through a textfile using redirection of input
`% mail -s 'This is subject.' someone < file`
7. Consult on-line manual of `mail` by
`% man mail`

Chapter 7

vi editor

`vi` is a standard editor on UNIX. `vi` is available everywhere UNIX is working. `vi` seems rather incapable and complicated than other editors in MS-DOS, but it is because `vi` was originally designed to work on many kinds of environment — some types of keyboard don't have function key or even ten keys. However, once you have accustomed to `vi`, you will find it is very fast and easy to use. The only way to be accustomed to editors such as `vi` is to repeat practices. So don't hesitate to use `vi`.

7.1 Major modes of vi

`vi` editor operates in two modes: *command mode* and *input mode*. In command mode, `vi` interprets your keystrokes as commands; there are many `vi` commands. You can use commands to save a file, exit `vi`, move the cursor to various positions in a file, or modify, rearrange, delete, substitute, or search for text. If you enter a character as a command but the character is not a command, `vi` beeps. The beep is an audible indication for you to check what you are doing and correct any errors.

You can enter text in input mode (also called text-entry mode) by either *appending* after the cursor or *inserting* before the cursor. At the beginning of the line, this doesn't make much difference. To go from command mode to input mode, press `a` to append text after the cursor, or press `i` to insert text in front of the cursor.

Use input mode only for entering text. Most word processing programs start in input mode, **but `vi` doesn't**. When you use a word processing program, you can type away, entering text; to issue a command, you have to use function keys or keys different than you use when typing normal text. `vi` doesn't work that way; you must go into input mode by pressing `a` or `i` before you start entering text and then explicitly press `Esc` to return command mode.

7.2 At least you should know

7.2.1 To start vi

When you edit a new file with `vi`, simply

`% vi filename` 

Notice that you are in *command mode* just after the screen of `vi` opened. Press `a` or `i` to change the mode to *input mode* to enter characters.

7.2.2 To quit vi, saving the text

To save text and quit, you have to first enter into *command mode* by pressing **Esc**. Then press **ZZ**. vi will save your text and quit.

7.2.3 Most basic cursor movement

You must be in *command mode* to move cursor. You cannot move cursor in *input mode*. Functions to move cursor are

- Cursor Up: **k**
- Cursor Down: **j**
- Cursor Left: **h**
- Cursor right: **l**

These are illustrated in Figure 7.1.

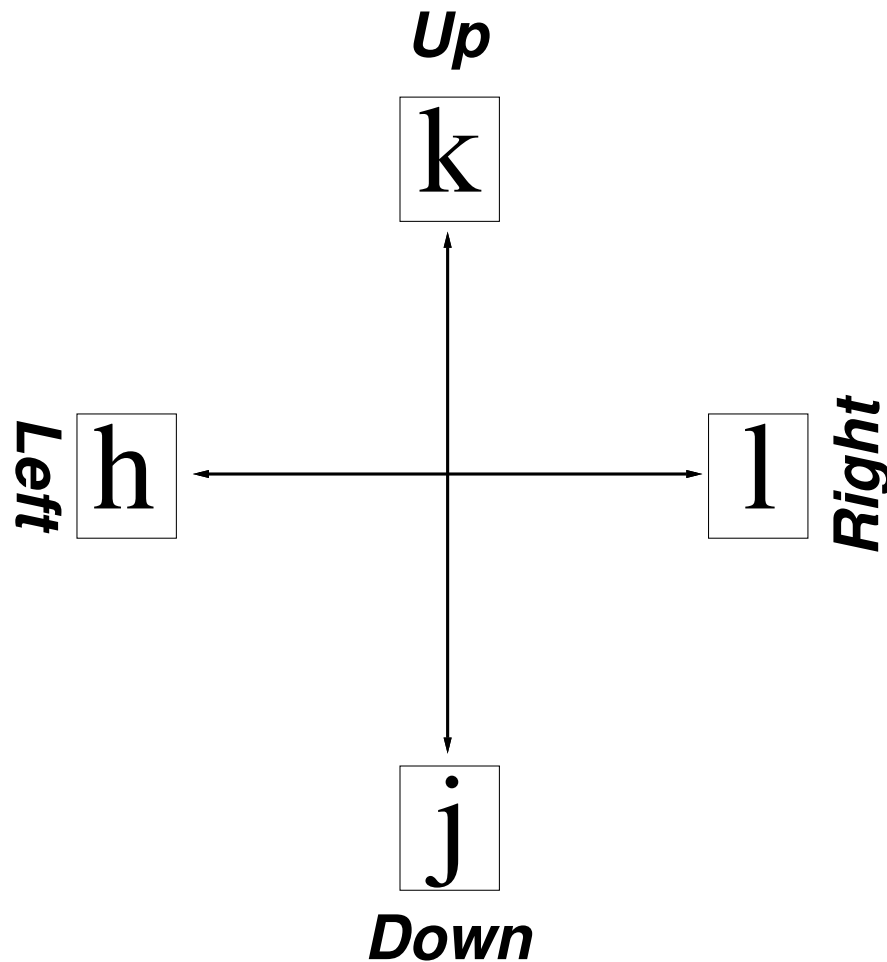


Figure 7.1. Schematic illustration of cursor movement in vi

Check List on vi

1. Read the description about `vi` in “*Workstation Users Guide 1993*” (Polytechnic University) p.25~, and try to check each sample.
2. Consult on-line manual of `vi` by
`% man vi`
3. Choose some of the proverbs listed in the accompanying papers, and type them in the file by `vi`. Then, save and send it to `tito@csa0` by E-mail. For example, suppose your filename is *samples*
`% mail -s 'Proverbs' tito@csa0 < samples`

Chapter 8

Environment files

Most users find that, every time they login, they type the same commands to set up the computer the way they like it. You may typically change your favorite directory, for example, change your terminal settings, check your mail, or any of a dozen other things.

Environment files (usually their name start with dot `.`) are used to set up your terminal settings automatically whenever you login, start shells, or start applications. Most environment files are some sorts of *shell script* (which corresponds to *batch file* in MS-DOS). Below we will explain four typical environment files: `.login`, `.cshrc`, `.logout`, and `.xinitrc`.

8.1 `.login`

`.login` is automatically loaded when you login the UNIX system. A sample of `.login` is listed below.

```
setenv PATH ~/bin:~tito/bin:/usr/local/bin:/bin:/usr/bin/X11:/usr/ucb:/usr/bin
setenv MANPATH /usr/man:/usr/lang/man:/usr/newsprint/man:/usr/openwin/man
setenv LD_LIBRARY_PATH /usr/lib/X11:/usr/lib
```

Here, `setenv` is a command for setting an *environment variable* which also exists in MS-DOS world. The formal syntax of `setenv` command is

```
% setenv variable value ...
```

Traditionally, environment variables are expressed by upper cases such as `PATH` or `TERM`. To list all the environment variables, you simply enter

```
% printenv
```

8.2 `.cshrc`

`.cshrc` is automatically executed when you start up `csh` (or when you login). A sample of `.cshrc` is listed below.

```
set    history = 100
set    prompt = "'whoami'@'hostname'[\!]%"
```

```
alias c      clear
alias h      history
alias key    man -k
alias la     ls -aF
alias ll     ls -lgF
alias lla    ls -lgFa
alias llm    ls -lagF | more
alias llt    ls -ltgF
alias ls     ls -CF
alias md     mkdir
alias ps     /bin/ps -aux | sort -n +1 | more
alias s      source
alias cd     'cd \!*;echo $cwd'
alias cp     cp -i
alias k14    kterm -fk k14 -fn a14 -km euc
alias k16    kterm -fk rom-k16 -fn rom-a16 -km euc
alias kterm  kterm -fk kanji16 -fn 8x16 -fr 8x16kana -km euc
alias mv     mv -i
alias pwd    echo $cwd
alias m      more
alias rm     rm -i
alias xinit  '/usr/bin/X11/xinit; logout'
alias xterm  xterm -fn 8x16 -sb
alias xterm14 xterm -fn 7x14 -sb
alias xterm18 xterm -fn a18 -sb
alias xterm24 xterm -fn 12x24 -sb
```

Here, **set** is a command for setting an *cs*h variable. It is rather difficult to explain accurately between the *cs*h variables and environment variables, but if we dare to say,

- *environment variables* ... could be consulted by any commands.
- *cs*h variables ... mainly consulted by *cs*h commands.

and some of *cs*h variables are common with environment variables (*e.g.* **term** and **TERM**, **path** and **PATH**). The formal syntax of **set** command is

```
% set variable = value ...
```

Traditionally, *cs*h variables are expressed by lower cases such as **history** or **tty**. To show all the *cs*h variables, you should enter the **set** command with no argument;

```
% set
```

8.2.1 Backquotes in C-shell

Backquotes in C-shell (‘ ‘) have a special meaning. Backquotes express the *output of command* which is bracketed between them; we show a simple example. **hostname** is a command to display your machine name, for example

```
% hostname  
csa2
```

Then, try to perform below two commands. One is simply

```
% echo hostname
```

The other is using backquotes

```
% echo `hostname`
```

The result will strikingly differ.

8.3 .logout

.logout is easy to understand, and it is executed when you logout from the system. For example, a sample .logout

```
clear  
black  
/usr/games/fortune -l
```

works as

1. Clear the text screen
2. Black the screen
3. Show some proverbs on the screen

when you logout. As you can see, .logout is not necessary needed.

8.4 .xinitrc

.xinitrc preserves the geometries, locations, sizes, and other information about the windows when you start X-Window system by xinit. A sample of .xinitrc is listed below.

```
twm &  
xbiff -geometry 62x62+0-0 &  
oclock -geometry 300x300+4-0 -trans &  
xterm -sb -fn 12x24 -geometry 80x36-4+4 &  
xterm -sb -fn 8x16 -geometry 80x27+4+120 &  
xrefresh  
exec xterm -C -sb -geometry 78x8+0+0 -n login
```

In the above script,

1. Start up terminal window manager (**twm**)
2. Start up **xbiff**

3. Start up `oclock`
4. Start up two `xterms`
5. Refresh the screen by `xrefresh`
6. Start up login window

Command `exec` in the final line is rather difficult, but for now it is enough to understand that the purpose of `exec ... -n login` is to finish the window system by exiting the login window (the window system will be finished together with the login window at the same time). Try to search the meaning of each option of each command by on-line manuals.

Check List on environment files

1. We prepared a typical example of useful environment files. Use this instead of present version. To do so, at first you should copy them from `~tito/Dec15` by


```
cp ~tito/Dec15/.login.sample ~/.login
cp ~tito/Dec15/.cshrc.sample ~/.cshrc
cp ~tito/Dec15/.logout.sample ~/.logout
cp ~tito/Dec15/.xinitrc.sample ~/.xinitrc
```

 When you are asked to “Overwrite?”, enter “y”. Then, logout once, and login again. What will happen?
2. Consult on-line manuals of each command for each option.
3. Verify the characteristics of backquotes in C-shell. Try to set two environment variables in different way. First, simply


```
% setenv TMP1 hostname
```

 and second, using backquotes


```
% setenv TMP2 `hostname`
```

 Confirm the result by


```
% printenv
```
4. Try to revise these environment files as you like. If you fail to rewrite them, some kind of troubles may occur on your terminal. In such cases, copy the original files again from `~tito/Dec15`.

Chapter 9

Using compilers

Three language compilers (`cc`, `f77`, `pc`) are available. Each of them corresponds to C, Fortran77, and Pascal language.

`cc` for C

To compile a C source code, we use `cc` (C Compiler) as

```
% cc source.c ↵
```

`f77` for Fortran77

To compile a Fortran (Fortran77) source code, we use `f77`.

```
% f77 source.f ↵
```

`pc` for Pascal

To compile a Pascal source code, we use `pc` (Pascal Compiler).

```
% pc source.p ↵
```

9.1 The name of output

It should be noted that the name of output executable from these compiler is by default fixed to `a.out` (it is of course due to the traditional reason). So when you intend to run the executable,

```
% a.out ↵
```

is needed. If you want to change the name of output executable, you should use `-o` option of the compiler, such as

```
% cc -o testexec source.c ↵
```

Then, an executable file named `testexec` will be produced instead of `a.out`. You can run it by

```
% testexec ↵
```

9.2 Using mathematical functions

You may notice that when you use mathematical functions (`sin`, `cos`, `exp`, `ln`, ...) in the source program, it is necessary for you to put `-lm` option **after the name of source program when compiling**. For example,

```
% cc -o testexec source.c -lm ↵
```

The option `-lm` means that the code is Linked with the Mathematical libraries when compiling. It is a common regulation for the compilers on UNIX. However, since it is a OS/compiler dependent specification, you may not need to put `-lm` option when compiling. Try to check it.

Check List on compilers

1. Type the list 1, 2, 3 below in some files, save them as proper names, and compile them. Then, execute each of them.
2. Compilers `cc`, `f77`, `pc` have many options. Consult the each meaning of the options by on-line manuals.
`man cc f77 pc`

List 1

```
/*
  Sample code of C
*/
#include <stdio.h>
main(){
    printf("Hello, C world!\n");
}
```

List 2

```
*
*   Sample code of Fortran
*
    Program Hello
    write (*,*) 'Hello, Fortran world!'
    Stop
    End
```

List 3

```
{
  Sample code of Pascal
}
program hello;
begin
    writeln('Hello, Pascal world!');
end.
```

Chapter 10

Emacs editor

Emacs is an editor which is much more powerful and strong than vi. One of the reasons is that it doesn't have the mysterious modes that require you to remember at every moment whether the program is expecting a command or text.

Although emacs doesn't come with standard UNIX, a popular version called GNU Emacs is distributed free, so most systems have it or can get it. Also some commercial versions such as Epsilon, Unipress Emacs are available. Here we use GNU Emacs in this class.

10.1 At least you should know about emacs

Like vi, there are only a few things to remember when you intend to use emacs within minimum functions.

10.1.1 Start up

When you edit a new file with emacs, simply

```
% emacs filename & ↵
```

Don't forget to run emacs as a background job with &, because emacs creates a new window for himself (on X-Window only). If you feel the font in emacs is too small, use -fn option to specify font. For example,

```
% emacs -fn 8x16 filename & ↵
```

will use the eight times sixteen dot fonts, and

```
% emacs -fn 12x24 filename & ↵
```

will use the twelve times twenty-four dot fonts in the emacs window.

10.1.2 To quit emacs, saving the text

To save your text and quit emacs, type

control+x control+c

in a row. If you are asked to save the file or not, you should reply to it by “y” or “n” (if you answered “n”, you may be asked again for confirmation). Then, **emacs** will quit automatically.

10.1.3 Basic cursor movement

Keybinds of Emacs is based on the pattern of “control + key”. For example, functions to move cursor are binded as

- To Previous line: `control + p`
- To Next line: `control + n`
- To Forward one character: `control + f`
- To Backward one character: `control + b`

These are illustrated in Figure 10.1.

10.2 On-line tutorial

GNU Emacs has a detailed on-line tutorial. You should at least once read through this tutorial. It is very helpful for you to study and learn about **emacs** by yourself. Procedures to read the tutorial for **emacs** is very simple. First, you type

`control+h`

and you will see a message like

`C-h (Type ? for further options)`

Expression `C-h` means that you have typed “control+h” now. Then, enter `t` (lower case). Then you can get into the tutorial mode.

10.3 Important keybinds

In Table 10.1 we listed other important keybinds in **emacs**. In the table, the expression `C-` is equivalent to “control+”.

10.4 Undo function

Undo function is one of the strongest functions in **emacs**. Any time you make a change to the text and wish you had not done so, you can undo the change (return the text to its previous state) with the undo command, `C-x u`. Normally, `C-x u` undoes one command’s worth of changes; if you repeat the `C-x u` several times in a row, each time undoes one more command. There are two exceptions: commands that made no change (just moved the cursor) do not count, and self-inserting characters are often lumped together in groups of up to 20. This is to reduce the number of `C-x u`’s you have to type.

`C-_` is another command for undoing; it is just the same as `C-x u` but easier to type several times in a row. The problem with `C-_` is that on some keyboards it is not obvious how to type it. That is why `C-x u` is provided as well. On some DEC terminals, you can type `C-_` by typing `/` while holding down Control.

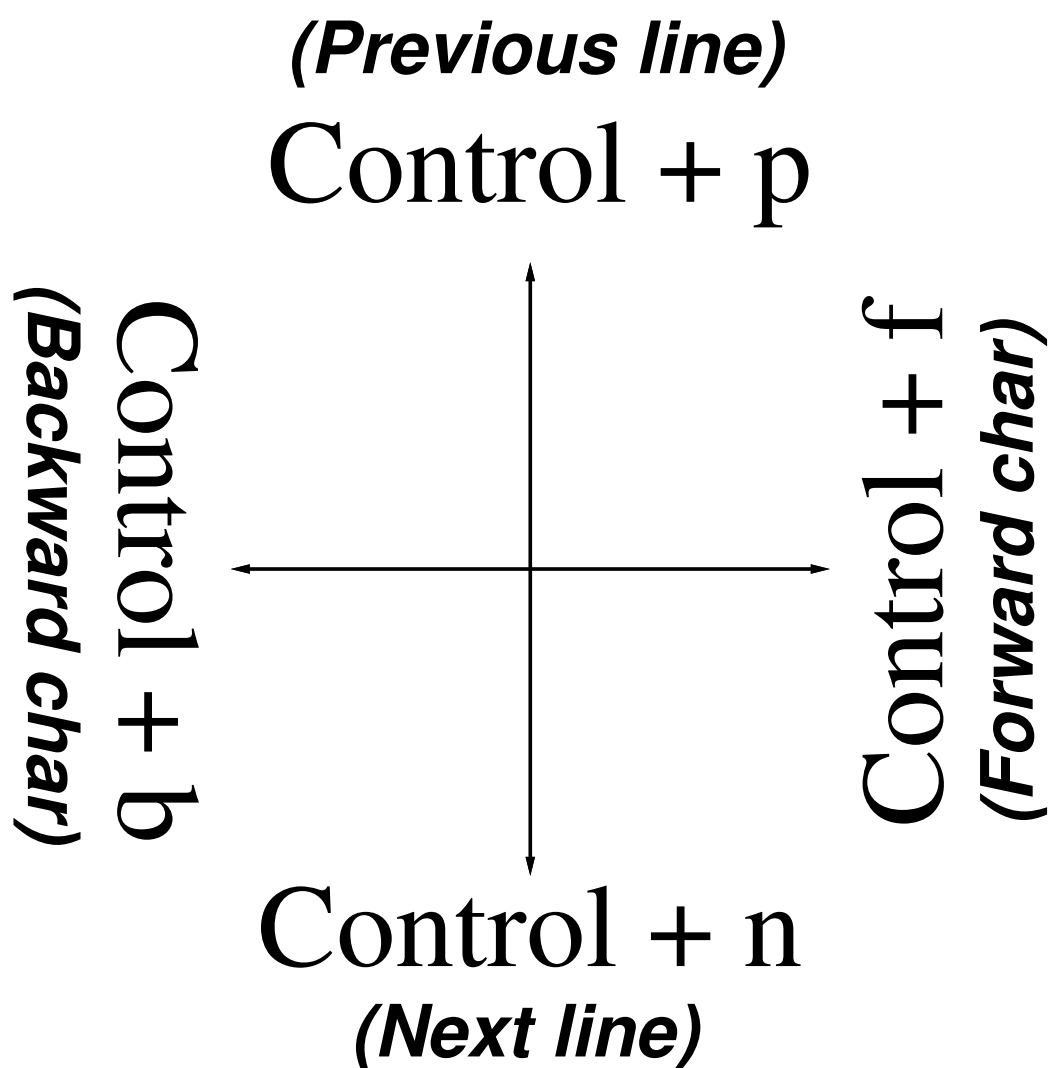


Figure 10.1. Schematic illustration of cursor movement in `emacs`

Table 10.1. Important keybindings in GNU Emacs.

Moving cursor	
Move over a char forward	C-f
Move over a char backward	C-b
Move over a word forward	ESC-f
Move over a word backward	ESC-b
Move to the top of line	C-a
Move to the end of line	C-e
Move to the top of file	ESC-<
Move to the end of file	ESC->
Deletion	
Delete a char on the cursor	C-d
Delete a char backward	DEL
Delete a word forward	ESC-d
Delete a word backward	ESC-DEL
Delete a line forward	C-k
Delete a line backward	ESC-0 DEL
Window operation	
Split into two	C-x 2
Move to other window	C-x o
Close all the other windows	C-x 1
Close the window you are in now	C-x 0
Buffer list	
Show the buffer list	C-x C-b
Choose the other buffer	C-x b
File handling	
Read in a file	C-x C-f
Save the current file	C-x C-s
Insert another file at the cursor	C-x i
Save as a new file	C-x C-w
Save and quit emacs	C-x C-c

Check List on emacs

1. Start up emacs with various fonts. For example,
`emacs &`
`emacs -fn 8x16 &`
`emacs -fn 12x24 &`
2. Try to read and check by yourself the tutorial for emacs. Procedures are:
 1. Start up emacs,
 2. Call the tutorial by “C-h t” ,
 3. Quit emacs by “C-x C-c” .

Subjects today

Choose each of the source codes listed below (C and Fortran, both perform the same work). Type it in file by using editor (`vi` or `emacs`). Then compile them, and send the output of executed result to `tito@csa0` by E-mail. (If you are familiar with Pascal, try to translate them into Pascal, and compile. Send the source code of Pascal to `yokoyama@csa0`)

The source programs will plot a cycloid on the terminal. The exact numerical expression of the figure is

$$\begin{cases} x &= Ca \cos 2\theta \cos \theta \\ y &= a \cos 2\theta \sin \theta \end{cases}$$

where radius $a = 18$ and a parameter $0 \leq \theta \leq 2\pi$ here. Aspect rate between the horizontal and vertical axis on the display C equals 2.0 because the font used by `xterm` is designed to have this ratio (7x14, 8x16, ...).

List 1 (C version)

```
#include <stdio.h>
#include <math.h>
#define PI 3.141592

main(){
char  area[73][73];
int   i, j, l, ix, iy;
float a, cvalue, r, t;

/* Fill area[][] with blank */
  for (i=0; i<73; i++){
    for (j=0; j<73; j++){
      area[i][j] = ' ';
    }
  }

/* Make x/y axe */
  for (l=0; l<73; l++){
    area[36][l] = '|';
    area[l][36] = '-';
  }

/* Specify the radius of the cycloid */
  a = 18.0;
/* Specify the aspect rate on the screen */
  cvalue = 2.0;

  for (t=0.0; t<2*PI; t+=0.01){
    r = a*cos(2*t);
    ix = r*cos(t)*cvalue;
    iy = r*sin(t);
```



```

        area[ix+36][iy+36] = '*';
    }

/* Output on the screen */
    for (j=18; j<55; j++){
        for (i=1; i<72; i++){
            fprintf(stdout, "%c", area[i][j]);
        }
        fprintf(stdout, "\n");
    }
}

```

List 2 (Fortran version)

```

Program cycloid
character*1 area
dimension area(-36:36, -36:36)
*
    Do 10 i=-36,36
        Do 11 j=-36,36
            area(i,j)=' '
11      Continue
10      Continue
        Do 12 l=-36,36
            area(0,l)='|'
            area(l,0)='- '
12      Continue
*
    a = 18.0
    cvalue = 2
    Do 20 t=0.0, 2.0*3.141592, 0.01
        r = a*cos(2*t)
        ix = r*cos(t)*cvalue
        iy = r*sin(t)
        area(ix,iy) = '*'
20      Continue
*
    Do 30 j=-18,18
        write(6,100) (area(i,-j), i=-34,34)
100      format(1h ,121a1)
30      Continue

    Stop
End

```

Chapter 11

C shell

From the next page, we list a sample of on-line manual of `cs`h from NEWS-OS 4.1 (Sony inc.). Though there are a lot of textbooks about C shell in the world, the original on-line manual is the most understandable, detailed, and helpful.

NAME

`csh` – a shell (command interpreter) with C-like syntax

SYNOPSIS

`csh` [`-bcefinstvVxX`] [`arg ...`]

DESCRIPTION

Csh is a first implementation of a command language interpreter incorporating a history mechanism (see **History Substitutions**), job control facilities (see **Jobs**), interactive file name and user name completion (see **File Name Completion**), and a C-like syntax. So as to be able to use its job control facilities, users of *csh* must (and automatically) use the new tty driver fully described in *tty(4)*. This new tty driver allows generation of interrupt characters from the keyboard to tell jobs to stop. See *stty(1)* for details on setting options in the new tty driver.

An instance of *csh* begins by executing commands from the file `‘.cshrc’` in the *home* directory of the invoker. If this is a login shell then it also executes commands from the file `‘.login’` there. It is typical for users on crt’s to put the command `“stty crt”` in their *.login* file, and to also invoke *tset(1)* there.

In the normal case, the shell will then begin reading commands from the terminal, prompting with `‘% ’`. Processing of arguments and the use of the shell to process files containing command scripts will be described later.

The shell then repeatedly performs the following actions: a line of command input is read and broken into *words*. This sequence of words is placed on the command history list and then parsed. Finally each command in the current line is executed.

When a login shell terminates it executes commands from the file `‘.logout’` in the users home directory.

Lexical structure

The shell splits input lines into words at blanks and tabs with the following exceptions. The characters `‘&’` `‘|’` `‘;’` `‘<’` `‘>’` `‘(’` `‘)’` form separate words. If doubled in `‘&&’`, `‘||’`, `‘<<’` or `‘>>’` these pairs form single words. These parser metacharacters may be made part of other words, or prevented their special meaning, by preceding them with `‘\’`. A newline preceded by a `‘\’` is equivalent to a blank.

In addition strings enclosed in matched pairs of quotations, `‘,’` `‘’` or `“”`, form parts of a word; metacharacters in these strings, including blanks and tabs, do not form separate words. These quotations have semantics to be described subsequently. Within pairs of `“”` or `“”` characters a newline preceded by a `‘\’` gives a true newline character.

When the shell’s input is not a terminal, the character `‘#’` introduces a comment which continues to the end of the input line. It is prevented this special meaning when preceded by `‘\’` and in quotations using `“”, “”, and “”`.

Commands

A simple command is a sequence of words, the first of which specifies the command to be executed. A simple command or a sequence of simple commands separated by `‘|’`

characters forms a pipeline. The output of each command in a pipeline is connected to the input of the next. Sequences of pipelines may be separated by ‘;’, and are then executed sequentially. A sequence of pipelines may be executed without immediately waiting for it to terminate by following it with an ‘&’.

Any of the above may be placed in ‘(‘)’ to form a simple command (which may be a component of a pipeline, etc.) It is also possible to separate pipelines with ‘| ’ or ‘&&’ indicating, as in the C language, that the second is to be executed only if the first fails or succeeds respectively. (See *Expressions*.)

Jobs

The shell associates a *job* with each pipeline. It keeps a table of current jobs, printed by the *jobs* command, and assigns them small integer numbers. When a job is started asynchronously with ‘&’, the shell prints a line which looks like:

```
[1] 1234
```

indicating that the job which was started asynchronously was job number 1 and had one (top-level) process, whose process id was 1234.

If you are running a job and wish to do something else you may hit the key **^Z** (control-Z) which sends a STOP signal to the current job. The shell will then normally indicate that the job has been ‘Stopped’, and print another prompt. You can then manipulate the state of this job, putting it in the background with the *bg* command, or run some other commands and then eventually bring the job back into the foreground with the foreground command *fg*. A **^Z** takes effect immediately and is like an interrupt in that pending output and unread input are discarded when it is typed. There is another special key **^Y** which does not generate a STOP signal until a program attempts to *read(2)* it. This can usefully be typed ahead when you have prepared some commands for a job which you wish to stop after it has read them.

A job being run in the background will stop if it tries to read from the terminal. Background jobs are normally allowed to produce output, but this can be disabled by giving the command “*stty tostop*”. If you set this tty option, then background jobs will stop when they try to produce output like they do when they try to read input.

There are several ways to refer to jobs in the shell. The character ‘%’ introduces a job name. If you wish to refer to job number 1, you can name it as ‘%1’. Just naming a job brings it to the foreground; thus ‘%1’ is a synonym for ‘fg %1’, bringing job 1 back into the foreground. Similarly saying ‘%1 &’ resumes job 1 in the background. Jobs can also be named by prefixes of the string typed in to start them, if these prefixes are unambiguous, thus ‘%ex’ would normally restart a suspended *ex(1)* job, if there were only one suspended job whose name began with the string ‘ex’. It is also possible to say ‘%?string’ which specifies a job whose text contains *string*, if there is only one such job.

The shell maintains a notion of the current and previous jobs. In output pertaining to jobs, the current job is marked with a ‘+’ and the previous job with a ‘-’. The abbreviation ‘%+’ refers to the current job and ‘%-’ refers to the previous job. For close analogy with the syntax of the *history* mechanism (described below), ‘%%’ is also a synonym for the current job.

Status reporting

This shell learns immediately whenever a process changes state. It normally informs you whenever a job becomes blocked so that no further progress is possible, but only just before it prints a prompt. This is done so that it does not otherwise disturb your work. If, however, you set the shell variable *notify*, the shell will notify you immediately of changes of status in background jobs. There is also a shell command *notify* which marks a single process so that its status changes will be immediately reported. By default *notify* marks the current process; simply say ‘notify’ after starting a background job to mark it.

When you try to leave the shell while jobs are stopped, you will be warned that ‘There are stopped jobs.’ You may use the *jobs* command to see what they are. If you do this or immediately try to exit again, the shell will not warn you a second time, and the suspended jobs will be terminated.

File Name Completion

When the file name completion feature is enabled by setting the shell variable *filec* (see **set**), *csh* will interactively complete file names and user names from unique prefixes, when they are input from the terminal followed by the escape character (the escape key, or control-`]`). For example, if the current directory looks like

```
DSC.OLD bin cmd lib xmpl.c
DSC.NEW chaosnet cmtest mail xmpl.o
bench class dev mbox xmpl.out
```

and the input is

```
% vi ch<escape>
```

csh will complete the prefix “ch” to the only matching file name “chaosnet”, changing the input line to

```
% vi chaosnet
```

However, given

```
% vi D<escape>
```

csh will only expand the input to

```
% vi DSC.
```

and will sound the terminal bell to indicate that the expansion is incomplete, since there are two file names matching the prefix “D”.

If a partial file name is followed by the end-of-file character (usually control-D), then, instead of completing the name, *csh* will list all file names matching the prefix. For example, the input

```
% vi D<control-D>
```

causes all files beginning with “D” to be listed:

```
DSC.NEW DSC.OLD
```

while the input line remains unchanged.

The same system of escape and end-of-file can also be used to expand partial user names, if the word to be completed (or listed) begins with the character “~”. For example, typing

```
cd ~ro<control-D>
```

may produce the expansion

```
cd ~root
```

The use of the terminal bell to signal errors or multiple matches can be inhibited by

setting the variable *nobeep*.

Normally, all files in the particular directory are candidates for name completion. Files with certain suffixes can be excluded from consideration by setting the variable *ignore* to the list of suffixes to be ignored. Thus, if *ignore* is set by the command

```
% set ignore = (.o .out)
```

then typing

```
% vi x<escape>
```

would result in the completion to

```
% vi xmpl.c
```

ignoring the files "xmpl.o" and "xmpl.out". However, if the only completion possible requires not ignoring these suffixes, then they are not ignored. In addition, *ignore* does not affect the listing of file names by control-D. All files are listed regardless of their suffixes.

Substitutions

We now describe the various transformations the shell performs on the input in the order in which they occur.

History substitutions

History substitutions place words from previous command input as portions of new commands, making it easy to repeat commands, repeat arguments of a previous command in the current command, or fix spelling mistakes in the previous command with little typing and a high degree of confidence. History substitutions begin with the character '!' and may begin **anywhere** in the input stream (with the proviso that they **do not** nest.) This '!' may be preceded by an '\ ' to prevent its special meaning; for convenience, a '!' is passed unchanged when it is followed by a blank, tab, newline, '=' or '('. (History substitutions also occur when an input line begins with '. This special abbreviation will be described later.) Any input line which contains history substitution is echoed on the terminal before it is executed as it could have been typed without history substitution. Commands input from the terminal which consist of one or more words are saved on the history list. The history substitutions reintroduce sequences of words from these saved commands into the input stream. The size of which is controlled by the *history* variable; the previous command is always retained, regardless of its value. Commands are numbered sequentially from 1.

For definiteness, consider the following output from the *history* command:

```
9 write michael
10 ex write.c
11 cat oldwrite.c
12 diff *write.c
```

The commands are shown with their event numbers. It is not usually necessary to use event numbers, but the current event number can be made part of the *prompt* by placing an '!' in the prompt string.

With the current event 13 we can refer to previous events by event number '!11', relatively as in '!-2' (referring to the same event), by a prefix of a command word as in '!d' for event 12 or '!wri' for event 9, or by a string contained in a word in the command as

in ‘!?mic?’ also referring to event 9. These forms, without further modification, simply reintroduce the words of the specified events, each separated by a single blank. As a special case ‘!!’ refers to the previous command; thus ‘!!’ alone is essentially a *redo*. To select words from an event we can follow the event specification by a ‘.’ and a designator for the desired words. The words of an input line are numbered from 0, the first (usually command) word being 0, the second word (first argument) being 1, etc. The basic word designators are:

0 first (command) word
 n n ’th argument
 first argument, i.e. ‘1’
 \$ last argument
 % word matched by (immediately preceding) ? s ? search
 $x - y$ range of words
 $-y$ abbreviates ‘0- y ’
 * abbreviates ‘-\$’, or nothing if only 1 word in event
 $x *$ abbreviates ‘ x -\$’
 $x -$ like ‘ $x *$ ’ but omitting word ‘\$’

The ‘.’ separating the event specification from the word designator can be omitted if the argument selector begins with a ‘,’ ‘\$’, ‘*’ ‘-’ or ‘%’. After the optional word designator can be placed a sequence of modifiers, each preceded by a ‘.’. The following modifiers are defined:

h Remove a trailing pathname component, leaving the head.
 r Remove a trailing ‘.xxx’ component, leaving the root name.
 e Remove all but the extension ‘.xxx’ part.
 $s / l / r /$ Substitute l for r t Remove all leading pathname components, leaving the tail.
 & Repeat the previous substitution.
 g Apply the change globally, prefixing the above, e.g. ‘g&’.
 p Print the new command but do not execute it.
 q Quote the substituted words, preventing further substitutions.
 x Like q, but break into words at blanks, tabs and newlines.

Unless preceded by a ‘g’ the modification is applied only to the first modifiable word. With substitutions, it is an error for no word to be applicable.

The left hand side of substitutions are not regular expressions in the sense of the editors, but rather strings. Any character may be used as the delimiter in place of ‘/’; a ‘\’ quotes the delimiter into the l and r strings. The character ‘&’ in the right hand side is replaced by the text from the left. A ‘\’ quotes ‘&’ also. A null l uses the previous string either from a l or from a contextual scan string s in ‘!? s ?’. The trailing delimiter in the substitution may be omitted if a newline follows immediately as may the trailing ‘?’ in a contextual scan.

A history reference may be given without an event specification, e.g. ‘!\$’. In this case the reference is to the previous command unless a previous history reference occurred on the same line in which case this form repeats the previous reference. Thus ‘!?foo? !\$’ gives the first and last arguments from the command matching ‘?foo?’.

A special abbreviation of a history reference occurs when the first non-blank character of an input line is a ‘.’. This is equivalent to ‘!:s’ providing a convenient shorthand for

substitutions on the text of the previous line. Thus 'lbib' fixes the spelling of 'lib' in the previous command. Finally, a history substitution may be surrounded with '{' and '}' if necessary to insulate it from the characters which follow. Thus, after 'ls -ld ~paul' we might do '!{l}a' to do 'ls -ld ~paula', while '!la' would look for a command starting 'la'.

Quotations with ' and "

The quotation of strings by " and "" can be used to prevent all or some of the remaining substitutions. Strings enclosed in " are prevented any further interpretation. Strings enclosed in "" may be expanded as described below.

In both cases the resulting text becomes (all or part of) a single word; only in one special case (see *Command Substitution* below) does a "" quoted string yield parts of more than one word; " quoted strings never do.

Alias substitution

The shell maintains a list of aliases which can be established, displayed and modified by the *alias* and *unalias* commands. After a command line is scanned, it is parsed into distinct commands and the first word of each command, left-to-right, is checked to see if it has an alias. If it does, then the text which is the alias for that command is reread with the history mechanism available as though that command were the previous input line. The resulting words replace the command and argument list. If no reference is made to the history list, then the argument list is left unchanged.

Thus if the alias for 'ls' is 'ls -l' the command 'ls /usr' would map to 'ls -l /usr', the argument list here being undisturbed. Similarly if the alias for 'lookup' was 'grep ! /etc/passwd' then 'lookup bill' would map to 'grep bill /etc/passwd'.

If an alias is found, the word transformation of the input text is performed and the aliasing process begins again on the reformed input line. Looping is prevented if the first word of the new text is the same as the old by flagging it to prevent further aliasing. Other loops are detected and cause an error.

Note that the mechanism allows aliases to introduce parser metasyntax. Thus we can 'alias print 'pr \!* | lpr' to make a command which *pr's* its arguments to the line printer.

Variable substitution

The shell maintains a set of variables, each of which has as value a list of zero or more words. Some of these variables are set by the shell or referred to by it. For instance, the *argv* variable is an image of the shell's argument list, and words of this variable's value are referred to in special ways.

The values of variables may be displayed and changed by using the *set* and *unset* commands. Of the variables referred to by the shell a number are toggles; the shell does not care what their value is, only whether they are set or not. For instance, the *verbose* variable is a toggle which causes command input to be echoed. The setting of this variable results from the *-v* command line option.

Other operations treat variables numerically. The '@' command permits numeric calculations to be performed and the result assigned to a variable. Variable values are, however, always represented as (zero or more) strings. For the purposes of numeric operations, the null string is considered to be zero, and the second and subsequent words

of multiword values are ignored.

After the input line is aliased and parsed, and before each command is executed, variable substitution is performed keyed by '\$' characters. This expansion can be prevented by preceding the '\$' with a '\ ' except within ""'s where it **always** occurs, and within ""'s where it **never** occurs. Strings quoted by ' ' are interpreted later (see *Command substitution* below) so '\$' substitution does not occur there until later, if at all. A '\$' is passed unchanged if followed by a blank, tab, or end-of-line.

Input/output redirections are recognized before variable expansion, and are variable expanded separately. Otherwise, the command name and entire argument list are expanded together. It is thus possible for the first (command) word to this point to generate more than one word, the first of which becomes the command name, and the rest of which become arguments.

Unless enclosed in "" or given the ':q' modifier the results of variable substitution may eventually be command and filename substituted. Within "", a variable whose value consists of multiple words expands to a (portion of) a single word, with the words of the variables value separated by blanks. When the ':q' modifier is applied to a substitution the variable will expand to multiple words with each word separated by a blank and quoted to prevent later command or filename substitution.

The following metasequences are provided for introducing variable values into the shell input. Except as noted, it is an error to reference a variable which is not set.

\$name

\${name}

Are replaced by the words of the value of variable *name*, each separated by a blank. Braces insulate *name* from following characters which would otherwise be part of it. Shell variables have names consisting of up to 20 letters and digits starting with a letter. The underscore character is considered a letter. If *name* is not a shell variable, but is set in the environment, then that value is returned (but : modifiers and the other forms given below are not available in this case).

\$name[selector]

\${name[selector]}

May be used to select only some of the words from the value of *name*. The selector is subjected to '\$' substitution and may consist of a single number or two numbers separated by a '-'. The first word of a variables value is numbered '1'. If the first number of a range is omitted it defaults to '1'. If the last member of a range is omitted it defaults to '\$#name'. The selector '*' selects all words. It is not an error for a range to be empty if the second argument is omitted or in range.

\$#name

\${#name}

Gives the number of words in the variable. This is useful for later use in a '[selector]'.

\$0

Substitutes the name of the file from which command input is being read. An error occurs if the name is not known.

\$number

\${number}

Equivalent to '\$argv[number]'.

\$*

Equivalent to '\$argv[*]'.

The modifiers ':h', ':t', ':r', ':q' and ':x' may be applied to the substitutions above as may ':gh', ':gt' and ':gr'. If braces '{' '}' appear in the command form then the modifiers must appear within the braces. **The current implementation allows only one ':' modifier on each '\$' expansion.**

The following substitutions may not be modified with ':' modifiers.

\$?name

\${?name}

Substitutes the string '1' if name is set, '0' if it is not.

\$?0

Substitutes '1' if the current input filename is known, '0' if it is not.

\$\$

Substitute the (decimal) process number of the (parent) shell.

\$<

Substitutes a line from the standard input, with no further interpretation thereafter. It can be used to read from the keyboard in a shell script.

Command and filename substitution

The remaining substitutions, command and filename substitution, are applied selectively to the arguments of builtin commands. This means that portions of expressions which are not evaluated are not subjected to these expansions. For example, the builtin command like '@' does not substitute its argument '*' as follows.

```
% @ a=2 *3
```

```
% echo $a
```

```
6
```

For commands which are not internal to the shell, the command name is substituted separately from the argument list. This occurs very late, after input-output redirection is performed, and in a child of the main shell.

Command substitution

Command substitution is indicated by a command enclosed in ```. The output from such a command is normally broken into separate words at blanks, tabs and newlines, with null words being discarded, this text then replacing the original string. Within `''`s, only newlines force new words; blanks and tabs are preserved.

In any case, the single final newline does not force a new word. Note that it is thus possible for a command substitution to yield only part of a word, even if the command outputs a complete line.

Filename substitution

If a word contains any of the characters `*`, `?`, `[` or `{` or begins with the character `~`, then that word is a candidate for filename substitution, also known as ‘globbing’. This word is then regarded as a pattern, and replaced with an alphabetically sorted list of file names which match the pattern. In a list of words specifying filename substitution it is an error for no pattern to match an existing file name, but it is not required for each pattern to match. Only the metacharacters `*`, `?` and `[` imply pattern matching, the characters `~` and `{` being more akin to abbreviations.

In matching filenames, the character `.` at the beginning of a filename or immediately following a `/`, as well as the character `/` must be matched explicitly. The character `*` matches any string of characters, including the null string. The character `?` matches any single character. The sequence `[...]` matches any one of the characters enclosed. Within `[...]`, a pair of characters separated by `-` matches any character lexically between the two.

The character `~` at the beginning of a filename is used to refer to home directories. Standing alone, i.e. `~` it expands to the invokers home directory as reflected in the value of the variable *home*. When followed by a name consisting of letters, digits and `-` characters the shell searches for a user with that name and substitutes their home directory; thus `~ken` might expand to `/usr/ken` and `~ken/chmach` to `/usr/ken/chmach`. If the character `~` is followed by a character other than a letter or `/` or appears not at the beginning of a word, it is left undisturbed.

The metanotation `a{b,c,d}e` is a shorthand for `abe ace ade`. Left to right order is preserved, with results of matches being sorted separately at a low level to preserve this order. This construct may be nested. Thus `~source/s1/{oldls,ls}.c` expands to `/usr/source/s1/oldls.c /usr/source/s1/ls.c` whether or not these files exist without any chance of error if the home directory for ‘source’ is `/usr/source`. Similarly `../{memo,*box}` might expand to `../memo ../box ../mbox`. (Note that ‘memo’ was not sorted with the results of matching `*box`.) As a special case `{`, `}` and `{ }` are passed undisturbed.

Input/output

The standard input and standard output of a command may be redirected with the following syntax:

`< name`

Open file *name* (which is first variable, command and filename expanded) as the standard input.

`<< word`

Read the shell input up to a line which is identical to *word*. *Word* is not subjected to variable, filename or command substitution, and each input line is compared to *word* before any substitutions are done on this input line. Unless a quoting ‘\’, ‘”’, ‘ ’ or ‘ ’ appears in *word* variable and command substitution is performed on the intervening lines, allowing ‘\’ to quote ‘\$’, ‘\’ and ‘. Commands which are substituted have all blanks, tabs, and newlines preserved, except for the final newline which is dropped. The resultant text is placed in an anonymous temporary file which is given to the command as standard input.

> name

>! name

>& name

>&! name

The file *name* is used as standard output. If the file does not exist then it is created; if the file exists, its is truncated, its previous contents being lost.

If the variable *noclobber* is set, then the file must not exist or be a character special file (e.g. a terminal or ‘/dev/null’) or an error results. This helps prevent accidental destruction of files. In this case the ‘!’ forms can be used and suppress this check.

The forms involving ‘&’ route the diagnostic output into the specified file as well as the standard output. *Name* is expanded in the same way as ‘<’ input filenames are.

>> name

>>& name

>>! name

>>&! name

Uses file *name* as standard output like ‘>’ but places output at the end of the file. If the variable *noclobber* is set, then it is an error for the file not to exist unless one of the ‘!’ forms is given. Otherwise similar to ‘>’.

A command receives the environment in which the shell was invoked as modified by the input-output parameters and the presence of the command in a pipeline. Thus, unlike some previous shells, commands run from a file of shell commands have no access to the text of the commands by default; rather they receive the original standard input of the shell. The ‘<<’ mechanism should be used to present inline data. This permits shell command scripts to function as components of pipelines and allows the shell to block

read its input. Note that the default standard input for a command run detached is **not** modified to be the empty file `/dev/null`; rather the standard input remains as the original standard input of the shell. If this is a terminal and if the process attempts to read from the terminal, then the process will block and the user will be notified (see **Jobs** above).

Diagnostic output may be directed through a pipe with the standard output. Simply use the form `'|&'` rather than just `'|'`.

Expressions

A number of the builtin commands (to be described subsequently) take expressions, in which the operators are similar to those of C, with the same precedence. These expressions appear in the `@`, `exit`, `if`, and `while` commands. The following operators are available:

`|` `&&` `|&` `==` `!=` `=~` `!~` `<=` `>=` `<` `>` `<<` `>>` `+` `-` `*` `/` `%` `!` `~` `()`

Here the precedence increases to the right, `'=='`, `'!=`', `'=~`' and `'!~`', `'<=`', `'>=`', `'<`' and `'>`', `'<<`' and `'>>`', `'+'` and `'-'`, `'*'`, `'/'` and `'%'` being, in groups, at the same level. The `'=='`, `'!=`', `'=~`' and `'!~`' operators compare their arguments as strings; all others operate on numbers. The operators `'=~`' and `'!~`' are like `'!=`' and `'=='` except that the right hand side is a *pattern* (containing, e.g. `'*'`s, `'?'`s and instances of `'[...]`') against which the left hand operand is matched. This reduces the need for use of the *switch* statement in shell scripts when all that is really needed is pattern matching.

Strings which begin with `'0'` are considered octal numbers. Null or missing arguments are considered `'0'`. The result of all expressions are strings, which represent decimal numbers. It is important to note that no two components of an expression can appear in the same word; except when adjacent to components of expressions which are syntactically significant to the parser (`'&'`, `'|'`, `'<'`, `'>'`, `'('`, `')'`) they should be surrounded by spaces.

Also available in expressions as primitive operands are command executions enclosed in `'{'` and `'}'` and file enquiries of the form `'-l name'` where *l* is one of:

r read access
w write access
x execute access
e existence
o ownership
z zero size
f plain file
d directory

The specified name is command and filename expanded and then tested to see if it has the specified relationship to the real user. If the file does not exist or is inaccessible then all enquiries return false, i.e. `'0'`. Command executions succeed, returning true, i.e. `'1'`, if the command exits with status 0, otherwise they fail, returning false, i.e. `'0'`. If more detailed status information is required then the command should be executed outside of an expression and the variable *status* examined.

Control flow

The shell contains a number of commands which can be used to regulate the flow of control in command files (shell scripts) and (in limited but useful ways) from terminal input. These commands all operate by forcing the shell to reread or skip in its input and, due to the implementation, restrict the placement of some of the commands.

The *foreach*, *switch*, and *while* statements, as well as the *if-then-else* form of the *if* statement require that the major keywords appear in a single simple command on an input line as shown below.

If the shell's input is not seekable, the shell buffers up input whenever a loop is being read and performs seeks in this internal buffer to accomplish the rereading implied by the loop. (To the extent that this allows, backward goto's will succeed on non-seekable inputs.)

Builtin commands

Builtin commands are executed within the shell. If a builtin command occurs as any component of a pipeline except the last then it is executed in a subshell.

alias

alias name

alias name wordlist

The first form prints all aliases. The second form prints the alias for name. The final form assigns the specified *wordlist* as the alias of *name*; *wordlist* is command and filename substituted. *Name* is not allowed to be *alias* or *unalias*.

alloc

Shows the amount of dynamic memory acquired, broken down into used and free memory. With an argument shows the number of free and used blocks in each size category. The categories start at size 8 and double at each step.

bg

bg %job ...

Puts the current or specified jobs into the background, continuing them if they were stopped.

break

Causes execution to resume after the *end* of the nearest enclosing *foreach* or *while*. The remaining commands on the current line are executed. Multi-level breaks are thus possible by writing them all on one line.

breaksw

Causes a break from a *switch*, resuming after the *endsw*.

case label:

A label in a *switch* statement as discussed below.

cd

cd name

chdir

chdir name

Change the shell's working directory to directory *name*. If no argument is given then change to the home directory of the user.

If *name* is not found as a subdirectory of the current directory (and does not begin with '/', './' or '../'), then each component of the variable *cdpath* is checked to see if it has a subdirectory *name*. Finally, if all else fails but *name* is a shell variable whose value begins with '/', then this is tried to see if it is a directory.

continue

Continue execution of the nearest enclosing *while* or *foreach*. The rest of the commands on the current line are executed.

default:

Labels the default case in a *switch* statement. The default should come after all *case* labels.

dirs

Prints the directory stack; the top of the stack is at the left, the first directory in the stack being the current directory.

echo wordlist

echo -n wordlist

The specified words are written to the shells standard output, separated by spaces, and terminated with a newline unless the **-n** option is specified.

else

end

endif

endsw

See the description of the *foreach*, *if*, *switch*, and *while* statements below.

eval arg ...

(As in *sh*(1).) The arguments are read as input to the shell and the resulting command(s) executed in the context of the current shell. This is usually used to execute commands generated as the result of command or variable substitution,

since parsing occurs before these substitutions. See *tset(1)* for an example of using *eval*.

exec command

The specified command is executed in place of the current shell.

exit

exit(*expr*)

The shell exits either with the value of the *status* variable (first form) or with the value of the specified *expr* (second form).

fg

fg %job ...

Brings the current or specified jobs into the foreground, continuing them if they were stopped.

foreach name (*wordlist*)

...

end

The variable *name* is successively set to each member of *wordlist* and the sequence of commands between this command and the matching *end* are executed. (Both *foreach* and *end* must appear alone on separate lines.)

The builtin command *continue* may be used to continue the loop prematurely and the builtin command *break* to terminate it prematurely. When this command is read from the terminal, the loop is read up once prompting with ‘?’ before any statements in the loop are executed. If you make a mistake typing in a loop at the terminal you can rub it out.

glob *wordlist*

Like *echo* but no ‘\’ escapes are recognized and words are delimited by null characters in the output. Useful for programs which wish to use the shell to filename expand a list of words.

goto word

The specified *word* is filename and command expanded to yield a string of the form ‘label’. The shell rewinds its input as much as possible and searches for a line of the form ‘label:’ possibly preceded by blanks or tabs. Execution continues after the specified line.

hashstat

Print a statistics line indicating how effective the internal hash table has been at locating commands (and avoiding *exec*’s). An *exec* is attempted for each component of the *path* where the hash function indicates a possible hit, and in each component which does not begin with a ‘/’.

history

history *n*

history **-r** *n*

history **-h** *n*

Displays the history event list; if *n* is given only the *n* most recent events are printed. The **-r** option reverses the order of printout to be most recent first rather than oldest first. The **-h** option causes the history list to be printed without leading numbers. This is used to produce files suitable for sourcing using the **-h** option to *source*.

if (*expr*) *command*

If the specified expression evaluates true, then the single *command* with arguments is executed. Variable substitution on *command* happens early, at the same time it does for the rest of the *if* command. *Command* must be a simple command, not a pipeline, a command list, or a parenthesized command list. Input/output redirection occurs even if *expr* is false, when command is **not** executed (this is a bug).

if (*expr*) **then**

...

else if (*expr2*) **then**

...

else

...

endif

If the specified *expr* is true then the commands to the first *else* are executed; otherwise if *expr2* is true then the commands to the second *else* are executed, etc. Any number of *else-if* pairs are possible; only one *endif* is needed. The *else* part is likewise optional. (The words *else* and *endif* must appear at the beginning of input lines; the *if* must appear alone on its input line or after an *else*.)

jobs

jobs -l

Lists the active jobs; given the **-l** options lists process id's in addition to the normal information.

kill %job**kill -sig %job ...****kill pid****kill -sig pid ...****kill -l**

Sends either the TERM (terminate) signal or the specified signal to the specified jobs or processes. Signals are either given by number or by names (as given in */usr/include/signal.h*, stripped of the prefix "SIG"). The signal names are listed by "kill -l". There is no default, saying just 'kill' does not send a signal to the current job. If the signal being sent is TERM (terminate) or HUP (hangup), then the job or process will be sent a CONT (continue) signal as well.

limit**limit resource****limit resource maximum-use****limit -h****limit -h resource****limit -h resource maximum-use**

Limits the consumption by the current process and each process it creates to not individually exceed *maximum-use* on the specified *resource*. If no *maximum-use* is given, then the current limit is printed; if no *resource* is given, then all limitations are given. If the **-h** flag is given, the hard limits are used instead of the current limits. The hard limits impose a ceiling on the values of the current limits. Only the super-user may raise the hard limits, but a user may lower or raise the current limits within the legal range.

Resources controllable currently include *cputime* (the maximum number of cpu-seconds to be used by each process), *filesize* (the largest single file which can be created), *datasize* (the maximum growth of the data+stack

region via *sbrk*(2) beyond the end of the program text), *stacksize* (the maximum size of the automatically-extended stack region), and *coredumpsize* (the size of the largest core dump that will be created).

The *maximum-use* may be given as a (floating point or integer) number followed by a scale factor. For all limits other than *cputime* the default scale is ‘k’ or ‘kilobytes’ (1024 bytes); a scale factor of ‘m’ or ‘megabytes’ may also be used. For *cputime* the default scaling is ‘seconds’, while ‘m’ for minutes or ‘h’ for hours, or a time of the form ‘mm:ss’ giving minutes and seconds may be used.

For both *resource* names and scale factors, unambiguous prefixes of the names suffice.

login

Terminate a login shell, replacing it with an instance of **/bin/login**. This is one way to log off, included for compatibility with *sh*(1).

logout

Terminate a login shell. Especially useful if *ignoreeof* is set.

nice

nice +number

nice command

nice +number command

The first form sets the scheduling priority for this shell to 4. The second form sets the priority to the given number. The final two forms run command at priority 4 and *number* respectively. The greater the number, the less cpu the process will get. The super-user may specify negative priority by using ‘nice -number ...’. Command is always executed in a sub-shell, and the restrictions placed on commands in simple *if* statements apply.

nohup

nohup command

The first form can be used in shell scripts to cause hangups to be ignored for the remainder of the script. The second form causes the specified command to be run with hangups ignored. All processes detached with ‘&’ are effectively *nohup*’ed.

notify

notify %job ...

Causes the shell to notify the user asynchronously when the status of the current or specified jobs changes; normally notification is presented before a prompt. This is automatic if the shell variable *notify* is set.

onintr

onintr -

onintr label

Control the action of the shell on interrupts. The first form restores the default action of the shell on interrupts which is to terminate shell scripts or to return to the terminal command input level. The second form ‘onintr -’ causes all interrupts to be ignored. The final form causes the shell to execute a ‘goto label’ when an interrupt is received or a child process terminates because it was interrupted.

In any case, if the shell is running detached and interrupts are being ignored, all forms of *onintr* have no meaning and interrupts continue to be ignored by the shell and all invoked commands.

popd

popd +n

Pops the directory stack, returning to the new top directory. With an argument ‘+n’ discards the *n*th entry in the stack. The elements of the directory stack are numbered from 0 starting at the top.

pushd

pushd name

pushd +n

With no arguments, *pushd* exchanges the top two elements of the directory stack. Given a *name* argument, *pushd* changes to the new directory (ala *cd*) and pushes the old current working directory (as in *csw*) onto the directory stack. With a numeric argument, rotates the *n*th argument of the directory stack around to be the top element and changes to it. The members of the directory stack are numbered from the top starting at 0.

rehash

Causes the internal hash table of the contents of the directories in the *path* variable to be recomputed. This is needed if new commands are added to directories in the *path* while you are logged in. This should only be necessary if you add commands to one of your own directories, or if a systems programmer changes the contents of one of the system directories.

repeat count command

The specified *command* which is subject to the same restrictions as the *command* in the one line *if* statement above, is executed *count* times. I/O redirections occur exactly once, even if *count* is 0.

set

set name

set name=word

set name[index]=word

set name=(wordlist)

The first form of the command shows the value of all shell variables. Variables which have other than a single word as value print as a parenthesized word list. The second form sets *name* to the null string. The third form sets *name* to the single *word*. The fourth form sets the *index*'th component of *name* to *word*; this component must already exist. The final form sets *name* to the list of words in *wordlist*. In all cases the value is command and filename expanded.

These arguments may be repeated to set multiple values in a single **set** command. Note however, that variable expansion happens for all arguments before any setting occurs.

setenv

setenv name value

setenv name

The first form lists all current environment variables. The last form sets the value of environment variable *name* to be *value*, a single string. The second form sets *name* to an empty string. The most commonly used environment variable **USER**, **TERM**, and **PATH** are automatically imported to and exported from the *cs**h* variables *user*, *term*, and *path*; there is no need to use *setenv* for these.

shift

shift variable

The members of *argv* are shifted to the left, discarding *argv*[1]. It is an error for *argv* not to be set or to have less than one word as value. The second form performs the same function on the specified variable.

source name

source -h name

The shell reads commands from *name*. *Source* commands may be nested; if they are nested too deeply the shell may run out of file descriptors. An error in

a *source* at any level terminates all nested *source* commands. Normally input during *source* commands is not placed on the history list; the `-h` option causes the commands to be placed in the history list without being executed.

stop

stop %job ...

Stops the current or specified job which is executing in the background.

suspend

Causes the shell to stop in its tracks, much as if it had been sent a stop signal with `^Z`. This is most often used to stop shells started by *su*(1).

switch (string)

case str1:

...

breaksw

...

default:

...

breaksw

endsw

Each case label is successively matched, against the specified *string* which is first command and filename expanded. The file metacharacters `*`, `?` and `[...]` may be used in the case labels, which are variable expanded. If none of the labels match before a `'default'` label is found, then the execution begins after the default label. Each case label and the default label must appear at the beginning of a line. The command *breaksw* causes execution to continue after the *endsw*. Otherwise control may fall through case labels and default labels as in C. If no label matches and there is no default, execution continues after the *endsw*.

time

time command

With no argument, a summary of time used by this shell and its children is printed. If arguments are given the specified simple command is timed and a time summary as described under the *time* variable is printed. If necessary, an extra shell is created to print the time statistic when the command completes.

umask**umask** value

The file creation mask is displayed (first form) or set to the specified value (second form). The mask is given in octal. Common values for the mask are 002 giving all access to the group and read and execute access to others or 022 giving all access except no write access for users in the group or others.

unalias pattern

All aliases whose names match the specified pattern are discarded. Thus all aliases are removed by ‘unalias *’. It is not an error for nothing to be *unaliased*.

unhash

Use of the internal hash table to speed location of executed programs is disabled.

unlimit**unlimit** *resource***unlimit** -h**unlimit** -h *resource*

Removes the limitation on *resource*. If no *resource* is specified, then all *resource* limitations are removed. If -h is given, the corresponding hard limits are removed. Only the super-user may do this.

unset pattern

All variables whose names match the specified pattern are removed. Thus all variables are removed by ‘unset *’; this has noticeably distasteful side-effects. It is not an error for nothing to be *unsetenv*.

unsetenv pattern

Removes all variables whose name match the specified pattern from the environment. See also the *setenv* command above and *printenv*(1).

wait

All background jobs are waited for. If the shell is interactive, then an interrupt can disrupt the wait, at which time the shell prints names and job numbers of all jobs known to be outstanding.

while (expr)

...

end

While the specified expression evaluates non-zero, the commands between the *while* and the matching end are evaluated. *Break* and *continue* may be used to terminate or continue the loop prematurely. (The *while* and *end* must appear alone on their input lines.) Prompting occurs here the first time through the loop as for the *foreach* statement if the input is a terminal.

%job

Brings the specified job into the foreground.

%job &

Continues the specified job in the background.

@

@ name = expr

@ name[index] = expr

The first form prints the values of all the shell variables. The second form sets the specified *name* to the value of *expr*. If the expression contains '<', '>', '&' or '|' then at least this part of the expression must be placed within '(' ')'. The third form assigns the value of *expr* to the *index*'th argument of *name*. Both *name* and its *index*'th component must already exist.

The operators '=', '+=', etc are available as in C. The space separating the name from the assignment operator is optional. Spaces are, however, mandatory in separating components of *expr* which would otherwise be single words.

Special postfix '++' and '--' operators increment and decrement *name* respectively, i.e. '@ i++'.

Pre-defined and environment variables

The following variables have special meaning to the shell. Of these, *argv*, *cwd*, *home*, *path*, *prompt*, *shell* and *status* are always set by the shell. Except for *cwd* and *status* this setting occurs only at initialization; these variables will not then be modified unless this is done explicitly by the user.

This shell copies the environment variable USER into the variable *user*, TERM into *term*, and HOME into *home*, and copies these back into the environment whenever the normal shell variables are reset. The environment variable PATH is likewise handled; it is not necessary to worry about its setting other than in the file *.cshrc* as inferior *csh* processes will import the definition of *path* from the environment, and re-export it if you then change it.

argv

Set to the arguments to the shell, it is from this variable that positional parameters are substituted, i.e. '\$1' is replaced by '\$argv[1]', etc.

- cdpath** Gives a list of alternate directories searched to find subdirectories in *chdir* commands.
- cwd** The full pathname of the current directory.
- echo** Set when the *-x* command line option is given. Causes each command and its arguments to be echoed just before it is executed. For non-builtin commands all expansions occur before echoing. Builtin commands are echoed before command and filename substitution, since these substitutions are then done selectively.
- filec** Enable file name completion.
- histchars** Can be given a string value to change the characters used in history substitution. The first character of its value is used as the history substitution character, replacing the default character *!*. The second character of its value replaces the character in quick substitutions.
- history** Can be given a numeric value to control the size of the history list. Any command which has been referenced in this many events will not be discarded. Too large values of *history* may run the shell out of memory. The last executed command is always saved on the history list.
- home** The home directory of the invoker, initialized from the environment. The filename expansion of *~* refers to this variable.
- ignoreeof** If set the shell ignores end-of-file from input devices which are terminals. This prevents shells from accidentally being killed by control-D's.
- mail** The files where the shell checks for mail. This is done after each command completion which will result in a prompt, if a specified interval has elapsed. The shell says 'You have new mail.' if the file exists with an access time not greater than its modify time.
- If the first word of the value of *mail* is numeric it specifies a different mail checking interval, in seconds, than the default, which is 10 minutes.
- If multiple mail files are specified, then the shell says 'New mail in *name*' when there is mail in the file *name*.
- noclobber** As described in the section on *Input/output*, restrictions are placed on output redirection to insure that files are not accidentally destroyed, and that '>>' redirections refer to existing files.
- noglob** If set, filename expansion is inhibited. This is most useful in shell scripts which are not dealing with filenames, or after a list of filenames has been obtained and further expansions are not desirable.
- nonomatch** If set, it is not an error for a filename expansion to not match any existing files; rather the primitive pattern is returned. It is still an error for the primitive pattern to be malformed, i.e. 'echo [' still gives an error.
- notify** If set, the shell notifies asynchronously of job completions. The default is to rather present job completions just before printing a prompt.

- path** Each word of the path variable specifies a directory in which commands are to be sought for execution. A null word specifies the current directory. If there is no *path* variable then only full path names will execute. The usual search path is `'.'`, `'/bin'` and `'/usr/bin'`, but this may vary from system to system. For the super-user the default search path is `'/etc'`, `'/bin'` and `'/usr/bin'`. A shell which is given neither the `-c` nor the `-t` option will normally hash the contents of the directories in the *path* variable after reading *.cshrc*, and each time the *path* variable is reset. If new commands are added to these directories while the shell is active, it may be necessary to give the *rehash* or the commands may not be found.
- prompt** The string which is printed before each command is read from an interactive terminal input. If a `'!'` appears in the string it will be replaced by the current event number unless a preceding `'\'` is given. Default is `'% '`, or `'# '` for the super-user.
- savehist** is given a numeric value to control the number of entries of the history list that are saved in `~/.history` when the user logs out. Any command which has been referenced in this many events will be saved. During start up the shell sources `~/.history` into the history list enabling history to be saved across logins. Too large values of *savehist* will slow down the shell during start up.
- shell** The file in which the shell resides. This is used in forking shells to interpret files which have execute bits set, but which are not executable by the system. (See the description of *Non-builtin Command Execution* below.) Initialized to the (system-dependent) home of the shell.
- status** The status returned by the last command. If it terminated abnormally, then 0200 is added to the status. Builtin commands which fail return exit status `'1'`, all other builtin commands set status `'0'`.
- time** Controls automatic timing of commands. If set, then any command which takes more than this many cpu seconds will cause a line giving user, system, and real times and a utilization percentage which is the ratio of user plus system times to real time to be printed when it terminates.
- verbose** Set by the `-v` command line option, causes the words of each command to be printed after history substitution.

Non-builtin command execution

When a command to be executed is found to not be a builtin command the shell attempts to execute the command via *execve*(2). Each word in the variable *path* names a directory from which the shell will attempt to execute the command. If it is given neither a `-c` nor a `-t` option, the shell will hash the names in these directories into an internal table so that it will only try an *exec* in a directory if there is a possibility that the command resides there. This greatly speeds command location when a large number of directories are present in the search path. If this mechanism has been turned off (via *unhash*), or if the shell was given a `-c` or `-t` argument, and in any case for each directory component

of *path* which does not begin with a '/', the shell concatenates with the given command name to form a path name of a file which it then attempts to execute.

Parenthesized commands are always executed in a subshell. Thus '(cd ; pwd) ; pwd' prints the *home* directory; leaving you where you were (printing this after the home directory), while 'cd ; pwd' leaves you in the *home* directory. Parenthesized commands are most often used to prevent *chdir* from affecting the current shell.

If the file has execute permissions but is not an executable binary to the system, then it is assumed to be a file containing shell commands and a new shell is spawned to read it.

If there is an *alias* for *shell* then the words of the alias will be prepended to the argument list to form the shell command. The first word of the *alias* should be the full path name of the shell (e.g. '\$shell'). Note that this is a special, late occurring, case of *alias* substitution, and only allows words to be prepended to the argument list without modification.

Argument list processing

If argument 0 to the shell is '-' then this is a login shell. The flag arguments are interpreted as follows:

- b This flag forces a "break" from option processing, causing any further shell arguments to be treated as non-option arguments. The remaining arguments will not be interpreted as shell options. This may be used to pass options to a shell script without confusion or possible subterfuge. The shell will not run a set-user ID script without this option.
- c Commands are read from the (single) following argument which must be present. Any remaining arguments are placed in *argv*.
- e The shell exits if any invoked command terminates abnormally or yields a non-zero exit status.
- f The shell will start faster, because it will neither search for nor execute commands from the file '.cshrc' in the invoker's home directory.
- i The shell is interactive and prompts for its top-level input, even if it appears to not be a terminal. Shells are interactive without this option if their inputs and outputs are terminals.
- n Commands are parsed, but not executed. This aids in syntactic checking of shell scripts.
- s Command input is taken from the standard input.
- t A single line of input is read and executed. A '\ ' may be used to escape the newline at the end of this line and continue onto another line.
- v Causes the *verbose* variable to be set, with the effect that command input is echoed after history substitution.
- x Causes the *echo* variable to be set, so that commands are echoed immediately before execution.

- V Causes the *verbose* variable to be set even before ‘.cshrc’ is executed.
- X Is to –x as –V is to –v.

After processing of flag arguments, if arguments remain but none of the –c, –i, –s, or –t options was given, the first argument is taken as the name of a file of commands to be executed. The shell opens this file, and saves its name for possible resubstitution by ‘\$0’. Since many systems use either the standard version 6 or version 7 shells whose shell scripts are not compatible with this shell, the shell will execute such a ‘standard’ shell if the first character of a script is not a ‘#’, i.e. if the script does not start with a comment. Remaining arguments initialize the variable *argv*.

Signal handling

The shell normally ignores *quit* signals. Jobs running detached (either by ‘&’ or the *bg* or %... & commands) are immune to signals generated from the keyboard, including hangups. Other signals have the values which the shell inherited from its parent. The shells handling of interrupts and terminate signals in shell scripts can be controlled by *onintr*. Login shells catch the *terminate* signal; otherwise this signal is passed on to children from the state in the shell’s parent. In no case are interrupts allowed when a login shell is reading the file ‘.logout’.

MULTI LANGUAGE SUPPORT

Csh processes the multiple languages of input text containing the Kanji character-set for Japanese.

You can enter Kanji as you enter ASCII characters by setting the tty mode to *sjis*, *euc*, *jis*, or *tca*(*Taiwan* code) mode. Metacharacters such as the double quotes (”), ampersand (&), vertical bar (|), semicolon (;), less than sign (<), greater than sign (>), open parenthesis ((), and close parenthesis ()) must be ASCII characters. The Zenkaku characters such as double quotes and ampersand are treated as regular characters. Consequently, a full size space (the shift JIS code 0x8140 and EUC code 0xa1a1) cannot be used as the separator for arguments.

A two-byte code is treated as one character for the metacharacters in filenames, asterisk character (*), question-mark character (?), and brackets ([and]).

Kanji can be used for the history function, Shell variables, and environmental variables.

AUTHOR

William Joy. Job control and directory stack features first implemented by J.E. Kulp of I.I.A.S.A, Laxenburg, Austria, with different syntax than that used now. File name completion code written by Ken Greer, HP Labs.

FILES

- ~/ .cshrc Read at beginning of execution by each shell.
- ~/ .login Read by login shell, after ‘.cshrc’ at login.
- ~/ .logout Read by login shell, at logout.
- /bin/sh Standard shell, for shell scripts not starting with a ‘#’.

/tmp/sh* Temporary file for '<<'.
/etc/passwd Source of home directories for '~name'.

LIMITATIONS

Words can be no longer than 1024 characters. The system limits argument lists to 10240 characters. The number of arguments to a command which involves filename expansion is limited to 1/6'th the number of characters allowed in an argument list. Command substitutions may substitute no more characters than are allowed in an argument list. To detect looping, the shell restricts the number of *alias* substitutions on a single line to 20.

SEE ALSO

sh(1), access(2), execve(2), fork(2), killpg(2), pipe(2), sigvec(2), umask(2), setrlimit(2), wait(2), tty(4), a.out(5), environ(7), 'An introduction to the C shell'

BUGS

When a command is restarted from a stop, the shell prints the directory it started in if this is different from the current directory; this can be misleading (i.e. wrong) as the job may have changed directories internally.

Shell builtin functions are not stoppable/restartable. Command sequences of the form 'a ; b ; c' are also not handled gracefully when stopping is attempted. If you suspend 'b', the shell will then immediately execute 'c'. This is especially noticeable if this expansion results from an *alias*. It suffices to place the sequence of commands in ()'s to force it to a subshell, i.e. '(a ; b ; c)'.

Control over tty output after processes are started is primitive; perhaps this will inspire someone to work on a good virtual terminal interface. In a virtual terminal interface much more interesting things could be done with output control.

Alias substitution is most often used to clumsily simulate shell procedures; shell procedures should be provided rather than aliases.

Commands within loops, prompted for by '?', are not placed in the *history* list. Control structure should be parsed rather than being recognized as built-in commands. This would allow control commands to be placed anywhere, to be combined with '|', and to be used with '&' and ';' metasyntax.

It should be possible to use the ':' modifiers on the output of command substitutions. All and more than one ':' modifier should be allowed on '\$' substitutions.

The way the **filec** facility is implemented is ugly and expensive.

REMARKS

setuid / setgid Shell Script

Due to security problems, command search paths have changed when executing **setuid** or **setgid** shell scripts in NEWS-OS Release 4.0 and later releases as follows:

Shell Search Paths Being Set /bin/csh (/usr/ucb /bin /usr/bin /usr/sony/bin)
/bin/sh /bin:/usr/bin:/usr/sony/bin

Reset search paths in shell scripts as required.

The above description also applies when using the `system()` or `popen()` function from the `setuid` application program. In this case, specify commands by full path.

Exercises on C shell basics (1)

1. Try to execute several commands in succession. For example,
% `ls; who; man whoami; echo $path`
% `(echo '1st line'; echo '2nd line') > file`
2. Try to control command execution and compare the results. For example,
% `ls -CF || ls -l`
% `lss -CF || ls -l`
% `ls -CF && ls -l`
% `lss -CF && ls -l`
Notice that the command `lss` does not exist actually.
3. Confirm the substitution of variables. For example,
% `echo $path`
% `echo path`
and what is the difference?
4. Let a variable have several values at a time. For example,
% `set list = (abc def ghi jkl mno)`
% `echo $list`
% `echo $list[1]`
% `echo $list[4]`
5. Confirm the command substitution. For example, try
% `ls `which echo``
and
% `ls 'which echo'`
and check the difference.
6. Try to set the toggle variable `ignoreeof`.
 - (a) Open a `xterm`. Then, exit by `^D`.
 - (b) Again open a `xterm` and set `ignoreeof`. Then, exit by `^D`. What will happen?
7. Try to set the toggle variable `noclobber`.
 - (a) Create a new file such as
% `echo NEWFILE > testfile`
 - (b) Set `noclobber` by
% `set noclobber`
 - (c) Overwrite a file such as
% `echo AGAIN > testfile`
Then, what will happen?
 - (d) Next, unset `noclobber` by
% `unset noclobber`
 - (e) Overwrite a file such as
% `echo AGAIN > testfile`

8. Similarly, confirm the working of `noglob`, `notify`, `filec`, and `nonomatch`. As a detail, read the manual of `cs`.
9. Check the values of value variables. For example,
% echo \$cwd
% echo \$home
% echo \$path
% echo \$history
10. Check the working of single quotations. Try next examples
% echo 'a+b >= c*d' > file
and with no quotations
% echo a+b >= c*d > file
What is the difference?
11. Check the difference of quotations. For example,
% echo "My cwd is \$cwd"
% echo 'My cwd is \$cwd'
and
% echo "Now it is 'date'"
% echo 'Now it is 'date''

Exercises on C shell basics (2)

1. Check the difference of standard output and standard error output. For example, try a non-existent command `lll`
`% lll > file.std`
The outputs (error message from shell) are displayed onto the standard error output, so they can not be redirected by simple `>`. Confirm it by using `cat` as
`% cat file.std`
`% ls -l file.std`
Next, try
`% lll >& file.err`
Then the standard error outputs are successfully redirected to a file. Confirm it by using `cat` as
`% cat file.err`
2. Try to confirm the working of `<<` on C shell. For example,
`% cat << ETD > testfile`
ABC
12345
ETD
`% cat testfile`
ABC
12345
3. Try to confirm the two different manual pages of `man`.
`% man man`
`% man 7 man`
`% man 1 man`
By the way, if you put such as
`% man 3 man`
what will happen?
4. Read the introductory pages on each manual section.
`% man 1 intro`
`% man 2 intro`
`% man 3 intro`
`% man 4 intro`
`% man 5 intro`
`% man 6 intro`
`% man 7 intro`
`% man 8 intro`

Exercises on C shell scripts

1. Check the performance of **foreach** by writing and executing the following script.

```
#!/bin/csh
foreach fname (*)
    echo "### $fname is my file ###"
    file $fname
end
```

About the meaning of **file** command, consult the on-line manuals.

2. Check the performance of **shift** by writing and executing the script of **List 1** on the attached sheets (p.3).
3. Write the script of **List 2** and **List 3** on the attached sheets (p.4, p.5). Let them run, and think how they work.

Chapter 12

Miscellaneous commands

12.1 Compressing and uncompressing files

12.1.1 compressing

One problem that is common to all UNIX systems — indeed, to nearly all computer systems of any kind — is that there is never enough disk space. UNIX comes with a programs that can alleviate this program, **compress**. They change the data in a file into a more compact form. Although you can't do anything with the file in this compact format from except expand it back to the original format, for files you don't need to refer to very often, it can be a big space saver.

compress reduces the size of the named files using adaptive Lempel–Ziv coding. Whenever possible, each file is replaced by one with the extension **.Z**, while keeping the same ownership modes, as well as access and modification times. The usage is simple like

```
% compress somefile
% ls
somefile.Z
```

Verbose option (**-v**) displays the percentage reduction for each file compressed. For example,

```
% compress -v somefile
somefile: Compression: 39.15% -- replaced with somefile.Z
```

The amount of compression obtained depends on the size of the input, the number of bits per code, and the distribution of common substrings. Typically, text such as source code or English is reduced by 50–60%. Compression is generally much better than that achieved by Huffman coding, or adaptive Huffman coding, and takes less time to compute. The bits parameter specified during compression is encoded within the compressed file, along with a magic number to ensure that neither decompression of random data nor recompression of compressed data is subsequently allowed.

NOTICE An compressed file is not a textfile, but a *binary file*, so you cannot see it by **cat** or some text editors!

12.1.2 uncompressing

To get the compressed file back to its original format, use `uncompress` command.

```
% uncompress somefile.Z
```

12.1.3 zcat

`zcat` is a compressed version of `cat` program, which sends an uncompressed version of a compressed file to the standard output, without changing the compressed files or storing the uncompressed version in a file. It is rarely useful by itself but can be quite handy with programs such as `more` or `lpr`. For example, you can read the contents of a compressed file without uncompressing it by

```
% zcat sometextfile.Z | more
```

12.2 Encoding/decoding files

12.2.1 encoding

Most mail programs are designed only for printable characters (textfiles). Unfortunately, many programs you would like to mail contain non-printable characters that may crash or confuse the file transfer program that actually send the mail. UNIX provides two utilities to turn binary files into printable textfiles and back again. `uuencode` and `uudecode`. Typical usage of `uuencode` is

```
% uuencode sourcefile filelabel > outfile.uu
```

`uuencode` converts a binary file into an ASCII-encoded representation that can be sent using `mail(1)`. It encodes the contents of `sourcefile`, or the standard input if no `sourcefile` argument is given. The `filelabel` argument is required. It is included in the encoded file's header, and becomes the filename of the binary (decoded) data. `uuencode` also includes the ownership and permission modes of `sourcefile`, so that `filelabel` is recreated with those same ownership and permission modes.

Contents of an uuencoded file is for example as follows:

```
begin 644 filelabel
MDVENDW1H:7.38VQA<W.3;V:37$ENF'1R;Y!-WF1U8W1I;VZ3=&^354Y)6"*;
M W0!872354E414.8=&AI<Y-YD+(A96%R+HZD#9F:C9$)]5U53DE8D03#:6*:
M3=YE8V]M97.6!,.S;VYEDV]FDW1H99-M;W-TDW"8;W!U;&%RDV%N9)-PF&^5
M9VYE9)8#0.%T;Y-BD$W>99-U<V5R+69R:65N9&QYDW-Y<W1E;2R1 \* 8723
....

MF%5.25B1! Z?:7.8<G5N;FEN9YAI<YAR87!I9&QYF&EN8W)E87-I;F<LD00H
MSVENF&%L;6]S=)AE=I-E<GF.H8V1"?5=<V]R='.6 Z*Y;V:38V]M<'5T97)S
M9"Z1!-RC26Z3=&AI<Y-C;&%S<RR1 Z-V=YAEDW=I;&R3<W1A<G239G)0;9-T
MDVENDV5VFK(A97)YDVMI;F1SDV]FDV-O=6Z8=')I97.3:6Z3=&AEDW>8;W)L
#W]_?
```

```
end
```

The encoded file is an ordinary ASCII text file; it can be edited by any text editor. But it is best only to change the mode or `filelabel` in the header to avoid corrupting the decoded binary. The encoded file's size is expanded by 35%, causing it to take longer to transmit than the equivalent binary.

Now you can send it by some mailer commands.

12.2.2 decoding

To decode an uuencoded file, use `uudecode`. `uudecode` reads an encoded file, strips off any leading and trailing lines added by other programs (*e.g.* mailer), and recreates the original binary data with the `filelabel` and the mode and owner specified in the header. For example, when someone sent you a binary file in the form of an uuencoded file by E-mail, at first you should save it in a textfile. Then, uudecode it by

```
% uudecode mailfile
```

And, a file named `filelabel` (which is described in the encoded file) is newly created. Since both `uuencode` and `uudecode` run with user ID set to `uucp`, `uudecode` can fail with “**Permission denied**” when attempted in a directory that does not have write permission allowed for other.

Exercises on file compressing

1. Try to compress any binary files by `compress` command. Check the degree of compression by `-v` option. For example,
% `cp /usr/ucb/vi .`
% `ls -l vi`
% `compress -v vi`
% `ls -l vi.Z`
2. Try to uncompress files which you had compressed now.
3. Try to compress some textfiles, and see the content by `zcat` command. For example,
% `compress -v textfile`
% `zcat textfile.Z | more`

Exercises on file encoding

1. Try to encode any file you like by `uuencode` command. For example,
% `uuencode filename filelabel > file.uu`
% `more file.uu`
2. Try to decode the file which you had encoded now.
3. Send someone an encoded binary file.
% `mail someone < file.uu`

Additional documents

Now we prepared three kinds of additional documents:

1. A useful list of UNIX and C books, with descriptions and some mini reviews. There are currently 167 titles on the list.
2. “The UNIX Acronym List” compiled by Wolfram Roesler (wr@bara.oche.de) which explains in detail what are the origins of the acronym form of major UNIX command names.
3. A list of services available on Internet.

Our mail addresses

Our mail addresses are as follow. If you find any question or comment when you go back to your country, please send an E-mail to us if possible. We are very willing to answer you as soon as possible.

Takashi Ito

tito@pluto.mtk.nao.ac.jp

National Astronomical Observatory, Mitaka, Tokyo 181, Japan.

Yukiko Yokoyama

yokoyama@uitec.ac.jp

The Polytechnic University, Sagamihara, Kanagawa 229, Japan.