An Introduction to UNIX in the department of physics at The University of South Florida

David A. Rabson

March 1996, most recently edited August 2006 for Physics Z-5156. *(While I have tried to root out most old references, some undoubtedly remain, for which I apologize.)*

(i) Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.

(ii) Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.

(iii) Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.

(iv) Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

from M.D. McIlroy, E.N.Pinson, and B.A. Tague *Unix Time-Sharing System Forward*, The Bell System Technical Jounal, July -Aug 1978 vol 57, number 6 part 2, p. 1902.

The author wishes to thank Christopher Myers, N. David Mermin, Todd Olson, Dan Sullivan, Stephen Langer, and William Newman for comments on earlier editions. He thanks John Wilkins for donating his lists of vi *and* jove *commands, the latter of which have been adapted to* emacs, *and especially thanks Arthur Smith, the co-author of an earlier edition, for permission to revise the manual. The author takes full responsibility for the objectionable personal opinions expressed here and for the pedagogy of inundation. This document was set in* gtroff *on a Sun Ultra 60 with parts in* **TEX**.

Note on the December 1995 edition: UNIX has become a far more complicated operating system than it was when Arthur Smith and I wrote the first two editions of this manual, then titled *An Introduction to UNIX at the Laboratory of Atomic and Solid State Physics.* Agreement on the "standard places" for files no longer exists, and manual pages, executables, and libraries are now scattered all over the system. Symbolic links, run-time libraries, **TEX**, and X Windows all add to what the beginning user must learn. Worse, a dozen subtly different versions, in two flavors, of the operating system mean that commands that work on machine X don't on machine Y. Nonetheless, the underlying simplicity of the UNIX philosophy remains. I hope to help the reader find it.

Note on June 1990 edition: several people have made useful suggestions for changes and additions to this booklet that could easily have tripled its size. I believe that 22 pages is the right length, or perhaps even too long, so I have not tried to address all the issues.

Logging In and Out

Type *username* to `login:` prompt
Type *password* to `Password:` prompt
Type `passwd` to `%` prompt to change password
Type `logout` to `%` prompt to leave

Help

`apropos` *subject* find manual pages on *subject*
`man` *subject* type manual page
`help` local help system

Create or Edit File

Type `cat >` *filename* to create a new file; type text, control-D to end.
Alternatively, type `vi` *filename* or `emacs` *filename* to invoke an editor.
Type `teachvi` or `teachemacs` to learn an editor.

Look at File

`more` *filename* or `cat` *filename*

Print File

`mpage -2 -Plp` *filename* or `lpr` *filename*
Either of these can be sent to an alternative printer with `-Plp`*n*.

Listing Directory

`ls`
or `ls -al` for more information

Copying and Moving

`cp` *source destination* to copy
`mv` *source destination* to rename or move
`rm` *filename* to remove

Changing Directory

`cd` to go to home directory
`cd` *directory* to change to subdirectory
`cd ..` to go up one level
`pwd` to find out where you are

Directories

`mkdir` *directory* to make new directory
`rmdir` *directory* to remove an empty directory
`du [`*directory*`]` to find out disk usage under directory

Wild Cards, Shell Expansions

`*` matches all files in directory (except dot-files)
`*Z` matches all filenames ending in Z in directory
`~/file` is a file in your home directory
`~davidra/file` is a file in davidra's home directory

Input/Output Redirection

command > *filename* writes output to file
command < *filename* gets input from file
command >> *filename* appends output to file
command1 `|` *command2* pipes output of first command to input of second

Searching in File

`fgrep` *string file1 file2* ... finds all occurrences of *string* in the given files

Job Control

command `&` runs command in background
control-Z stops job in preparation for `%` or `bg`
`jobs` lists background and stopped jobs; gives *n* for:
`%`*n* resumes job *n*
`bg %`*n* runs stopped job *n* in background

C Compiler

`cc -c` *prog.c* compile `prog.c` into object code `prog.o`
`cc -o` *prog prog.o* link `prog.o` into executable `prog`
option -g in each stage allows use of debugger

Mail

`mail` read mail
`mail` *user@machine* send mail

1. This manual

This manual is intended to tell the sophisticated scientific user who nonetheless is unfamiliar with UNIX[1] what he or she needs to know in order to begin writing programs. It is intended to be not inclusive nor tutorial nor eloquent, but concise. As a mostly personal introduction, it may also be considered somewhat biased. After reading this document, you will be able to use an editor (vi or emacs), to write a scientific program in your favorite language, compile it successfully, and run it. You will also know how to move around on the disk, customize your initialization files, and use mail. Most important, you will get some idea of the full capabilities of UNIX and will know where to look up anything you need to know in the on-line documentation.

If you just want to get started, read section 5 of this manual (*Commands and Information Useful to the Beginner*), saving the rest for when you need it. You may also find the condensed reference sheet (page *ii*) useful. If you forget everything else, remember "**apropos whatever**" for finding what commands do *whatever* and "**man command**" for information on *command*.

2. Conventions

By ˆX we shall mean <ctrl>-X, or the character generated by hitting the control button and while holding it down in the manner of a secondary "shift" key hitting the X key.
By <CR> we shall mean the carriage return key (usually treated the same as line feed under UNIX).
By alt-X or meta-X we will mean a keystroke generated with the "alt" or (on Sparc consoles) "diamond" key. This generates an otherwise ordinary character with the parity (7) bit set.

3. Introduction: Why UNIX? (or, "what is an operating system?")

An Operating System comprises a Kernel and auxiliary programs. The Kernel generally runs in a privileged mode from which it may access protected parts of memory. A user (or auxiliary) program routes its requests to access protected parts of memory (or I/O) through system calls.

Except for UNIX, operating systems are strongly machine dependent. Indeed, from the mid 1970's until the early 1990's, UNIX was the *only* operating system to run on more than one type of hardware. Now Macintosh, Windows, and VMS each run on two or three families of chips. In contrast, UNIX runs on everything from an Intel 80386 to the largest Cray. The kernels of most operating systems are written in the assembly languages of the machines for which they are intended, making it difficult to transport the code to other architectures. The UNIX kernel, on the other hand, is written in C, a language designed to be completely portable to any machine, but faster than FORTRAN and sufficiently flexible to code the most tortuous intricacies of a device driver.

The other part of the UNIX philosophy that enables it to run on almost any machine is its simplicity. Although the kernel has grown quite large compared to the first version, it still follows the general rule that an OS kernel should do as little, not as much, as possible. Simplicity works for, not against the user: since all files are treated in the same way under UNIX, the user need know only one I/O interface, not ten as under VMS. Moreover, with only one kind of file, the many programming tools that enable a user to sort, cut, splice, search, and edit files are much smaller and easier to use than in VMS. There are, as a direct result, more of them. As a consequence of the very simple way programs talk to the user, the output of one program may be connected to the input of another in what is called a pipe or to a file through I/O redirection (the idea has been copied by MS-DOS). Another example of the benefits of simplicity comes in the small overhead associated with new processes.

When I started with UNIX, there was rather less of it to learn. Only two flavors existed, and the one from Berkeley seemed to be taking over. Since then, dozens of vendors have released their own versions of the operating system, each with its own enhancements, incompatibilities, and bugs. There are even several free versions, which are quite good.[2] Networking has been added along with file locking, symbolic links, and windowing systems. Not all of these have been integrated as well as they might have been, and they continue to evolve. While this state of affairs might seem bewilderingly chaotic, I count the great effort that has gone into both commercial and free versions a sign of vibrancy and evidence that UNIX remains a programmer's preferred development platform.

Mark the emphasis on *development* over mere support for pre-packaged applications. The UNIX user, never satisfied with what the computer already knows how to do, manipulates data whatever their disparate sources, ties old

[1] UNIX used to be a trademark of Bell Laboratories.

[2] The server for this class runs the Red-Hat release of Linux. Among other free Unices are four important Linux releases as well as several variants of Free BSD.

programs together into new, builds objects upon objects. It is not a strategy for winning the mass Microsoft market. It is a way to do physics. The designer of an experiment needs screwdrivers, drill presses, lathes, and milling machines; someone who just wants to drive a car or store milk in a refrigerator does not. So too the computing needs of the physicist differ from those of the checkbook-balancer and video-game player. The most visible advantage of UNIX over all other operating systems is its wealth of programming tools. We will try to describe a few of these in this short manual.

4. Logging In and Out and Getting Started

When a user first logs in, a few short messages will generally appear, followed by a prompt for input. Under UNIX, these actions are controlled by a user-mode (not kernel-mode) program called a *shell*. In its simplest form, a shell repeatedly takes commands from the user and decides what programs need to be run in order to implement these commands.[3] Since the c-shell, or **csh**, is the default shell at USF physics, I will describe that shell, although the user may write a perfectly serviceable shell in ten lines of any high-level programming language.[4]

Each time the shell is invoked, it looks in the user's top-level directory[5] for a file called **.cshrc**. Note that the dot is part of the name. If **.cshrc** exists, the shell reads and executes the commands contained in it as if the user had typed them (but these commands are generally not displayed on the screen). Having read the **.cshrc**, the shell checks to see if it was invoked as a new login or a *subshell*. If it is a new login, the shell looks for a file called **.login** and executes the commands in it just as it did the **.cshrc**. Typically, the **.cshrc** contains aliases (shortened names for commands), while the **.login** contains commands to check for system notices and sets various options related to the terminal.

The commands in **.cshrc** and **.login** are critical to using UNIX; a majority of users' complaints can be traced to inadequacies in these two files.

Normally the **.cshrc** is read first, then the **.login**, but only on the login shell, and not on remote shells (see **rsh(1)**). Therefore most commands usually go in the **.cshrc**. Everything following the pound sign (#) is a comment (and ignored by the shell).

```
# Sample lines from .cshrc
#
# C-Shell options to set
set history=50         #set c-shell's memory of old commands
set noclobber          #don't overwrite files by redirection unless forced (i.e., >!)
set ignoreeof          #don't interpret ^D as ``exit'' or ``logout''
set prompt="`/bin/hostname` \!% "     #\! in prompt string replaced by command number
set mail=(20 /var/mail/$USER) #csh checks mailbox every 20 secs
#
# aliases
#
alias emacs emacs -nw          #tell emacs not to open a graphical window
alias rm "mv \!* ~/tmp"        #makes rm (remove) move file to personal temporary directory
#
# path (list of directories in which the shell looks for programs)
set path=(. ~/bin /usr/local/bin /usr/local/*/bin /usr/bin /bin)

setenv PRINTER lp7     # change the default printer
```

[3] To those from the Macintosh/Windows world, we must emphasize that UNIX is *not* graphical, although there are many good graphical programs and even interfaces. The general paradigm is that the user types commands, each one line long, which the machine obeys as soon as the user hits the return key.

[4] Actually, `csh` is the same as `tcsh` on Linux. Users may with the `chsh`(1) command change their default shells to any in the list `/etc/shells`.

[5] For those not familiar with hierarchical file systems (on which more later): this is simply a place where files are kept.

Before the login shell dies, it runs the commands in **.logout** if the file exists. There is a fuller description of the shell in section 6.

5. Commands and Information Useful to the Beginner

The first paragraph in this section contain preliminary information about special characters; the rest are about the five or six most essential commands.

5.1. Useful control characters

Subject to being reset by **stty** (in the **.login**) or usurped by programs for their own purposes, the following control keys perform useful tasks:

ˆU	erases the line you're in the middle of typing so you can start over
ˆW	erases just the word you're in the middle of typing (broken in `tcsh`)
ˆC	Kill whatever program (except the shell) you're running and return to the shell; incidentally acts like ˆU.
ˆ	Like ˆC for when ˆC doesn't work. Most programs will dump into a core file.
ˆZ	Suspend whatever program you're currently running; the shell includes facilities for continuing suspended jobs. This is useful, especially on terminals that don't support windows. Percent sign (%) resumes a stopped job.
ˆR	shows what you've typed so far
ˆS	suspend output (also known by its ASCII designation, XOFF)
ˆQ	resume output (XON)
ˆO	throw out subsequent output until input is requested; a second ˆO resumes output
ˆD	end of file — used to terminate keyboard input

The rubout key is usually either **delete** or **backspace** (**ˆH**). To change it, type `stty erase` *KEY*, hitting the key you want to be rubout in place of *KEY*. Linefeed (**ˆJ**) is the UNIX end-of-line character, although carriage return (**ˆM**) is usually mapped to the same thing.

5.2. ls (list files) and the UNIX filesystem

I assume the reader is familiar with the concept of a computer file. By *directory*, I mean a named collection of files; all postdiluvian operating systems support some sort of directory structure. Directories can contain other directories (sometimes called subdirectories) as well as files. Under UNIX, a *directory* is itself a file. See the table below for a summary comparison between UNIX's and some other operating systems' directories.

VMS	MS-DOS	Mac/Windows	UNIX	
			`/`	*the "root" of all disks*
`DISK7:[000000]`	`A:\`	picture of a disk	`/usr`	*the root of some disk*[a]
`[]`	`.`		`.`	*the current directory's name*
`FILE.EXT;`	`FILE.EXT`	icon named "file.ext"	`file.ext`	*file in current directory*
`FILE.`	`FILE.`	icon named "file"	`file`	*no dot (.) is necessary in UNIX*
`*.*;*`	`*.*`		`*`	*star (*) is interpreted by shell, not programs*
`[.SDIR]`	`SDIR`	a folder named "sdir"	`sdir`	*a subdirectory*
`DISK7:[DIR.SDIR]FILE.EXT;`	`A:\DIR\SDIR\FILE.EXT`		`/usr/dir/sdir/file.ext`	*full name (path) of file*
`[-.SDIR]`	`..\SSDIR`		`../ssdir`	*go up one level, then down one*
			`FiLE.nAm..e!`	*most characters allowed, case significant*

Note (a): Distinctions between physical disks or partitions in UNIX are nearly always invisible to the user.

When you first log in, you should find yourself in your own private directory, called your top-level (or home or login) directory. You can create subdirectories below your top level, and subdirectories can go arbitrarily deep. Files not below your top level usually belong to other people.

To see everything in a directory, give the command `ls -a`. **ls**, which stands for **list**, is the UNIX directory command. You should see a collimated list of file names in alphabetical order reading down one column and continuing at the top of the next. The file **.** (dot) is the name of the current directory, while **..** (dotdot) is the way to get from the current directory to its parent. If you issue the **ls** command without the **-a** option, these two directory names and any other files whose names that begin with a dot will not be listed. If this abbreviated form of the directory listing does not suit you, try `ls -aiglLRF`. I believe **ls** has more options (fifty in some implementations) than any other UNIX command; section 5.13 discusses UNIX command options more generally.

Files have a small number of one-bit attributes, mostly concerning protection and executability; they show up near the left-hand margin with `ls -l`. In the listing

```
-rwxr-x--x  1 username   131072 Jan 13 12:24 filename
```

the leading minus sign (-) indicates that the file is not a directory (it would have been **d** if **filename** had been a directory). The following **rwx** indicate that the owner (username) has permission, respectively, to read, to write on, and to execute it. The **r-x** that follow indicate that people in the same group as username may read or execute but not modify the file. Other people may only execute the file, as evidenced by the two minuses and x at the end.[6]

To look one directory above wherever you are, you can say "`ls ..`". You can change your current directory to **..** by saying "`cd ..`". **cd** without arguments will always return you to your login (top-level) directory. One directory above **..** lies **../..** ; slashes separate directory names. Eventually, you will find a directory whose parent is itself. This is the root directory and can always be referred to as / (slash), whatever your current directory; every directory on the system claims it as an ancestor, so a set of instructions (pathname) telling how to get from there to any file constitutes the full name of that file. To find the full name of your current directory, say `pwd` ("print working directory").

To create a directory, say `mkdir` *name* (where *name* is the name or pathname of the directory you wish to create). To remove a directory, say `rmdir` *name*. The operating system will not permit you to remove a directory that still has files (other than . and ..) in it (but see `rm -rf`).

Some Useful Directories (on our systems)

/tmp	a publicly-writable directory, good for small temporary storage (erased often)
/scratch/*username*	large storage space (not backed up)
/bin	really basic UNIX commands are here
/usr/bin	more commands
/usr/local	programs locally installed by the system manager
/usr/local/*package*	locally installed complicated software
/usr/local/doc	additional local documentation
/usr/lib	libraries (object code archives) reside here
/usr/include	header files for C and Fortran programs here and in directories underneath
~	each user's home directory: equivalent to **/home/*username***
/home/5156	assignments, supplementary material, and programs for for your course

5.3. looking at files: cat and more (and the shell's I/O redirection)

`cat file` copies a file to standard output (*i.e.*, the screen). `more file` does the same thing, but pauses at the bottom of each page, waiting for the user to hit the space bar. Typing "`h`" instead will give help on **more**. So long as the computer knows what kind of terminal you are on, **more** will be reasonably intelligent about underlining, highlighting, and size of the page.

[6] For directories (leading d), the x bit indicates permission to treat the directory as a directory. The **chmod** program changes the protection attributes of a file (an example is given in section 6).

`cat oldfile > newfile` uses the shell's output redirection facility to copy **oldfile** to **newfile**, overwriting the old contents of **newfile**. (However, if the c-shell **noclobber** variable is set, the shell will refuse to overwrite an existing file. Use "**>!**" instead of "**>**" to override **noclobber**.)

`cat > newfile` copies standard input (your terminal keyboard — end input with `^D`) to **newfile**, while `cat < oldfile > newfile` is the same as `cat oldfile > newfile`.

`cat file1 file2 file3 > file4` concatenates (hence the name of the command) the three files, placing the result in **file4**, while `cat file2 >> file1` appends file2 to the end of file1.

cat and **more** write only to "standard output," which "**>**" redirects. Some programs write also to "standard error." Use "**>&**" to redirect both standard output and standard error.[7]

5.4. moving and copying files

cp is used to copy files. `cp file1 file2` copies file1 to file2, overwriting any previously existing file2[8] and leaving file1 intact. `cp file1 file2 file3 subdir` copies the three files named to the directory **subdir**. There is an option in **cp** for copying everything under a directory to another directory.

mv (move) is like **cp**, but it leaves no old copies behind. Note that **mv** could also be called "rename." **mv** is usually more efficient than copying followed by removing.

rm (remove) erases the file. Because this is a strongly non-adiabatic process, many people have made **rm** an alias for moving the offending files to a temporary directory; see the sample **.cshrc** in section 4 . You can give **rm**'s full name, /bin/rm, if you want to be sure of actually deleting a file.

5.5. man and apropos: where to find help

The entire UNIX programming manual is on-line. If there's a section whose name you already know, use "man" to look at it:

```
man man
```

will give you information on the "man" command. More often, you may not know the name of the command you want to look up. In that case, use "apropos" and your closest guess.[9] Say you remembered there was an editor you liked but you forgot its name. Typing

```
apropos edit
```

for instance, will give a list of six or seven things the system thinks have to do with the string "edit:"

a.out (5)	- assembler and link **edit**or output
ed (1)	- text **edit**or
emacs (1)	- an interactive display-oriented text **edit**or
ex, **edit** (1)	- text **edit**or
ld (1)	- link **edit**or
teachemacs (1)	- learn how to use the EMACS **edit**or
vi (1)	- screen oriented (visual) display **edit**or based on ex

Quite obviously, only one or two of these entries is of interest to you. Note the numbers following the manual page names. The meanings of the first three sections follow.

[7] Bourne-derived shells work differently: `>stdout 2>stderr` or `>stdout 2>&1` to redirect both.

[8] Put the line `alias cp cp -i` in your **.cshrc** if this bothers you; **cp** will then ask permission before overwriting.

[9] On some systems, you may need the line `alias apropos man -k` in your **.cshrc**.

number	meaning
1	a command one can type to the shell
2	a system call (for use from C)
3	a library routine (for use from languages)

We alluded before to a great schism in the UNIX universe. Sections 4 through 9 may differ between *Berkeley*-based UNIX and *System V*, where sections can additionally have subheadings, making `man` more difficult to use.

Sometimes, there may be a name that appears in more than one section of the manual. In that case, you may give the section number as the second argument to *man*:[10]

man 2 wait

Not everything you'd ever want is in the manual pages. Other useful commands include **help** and **info**; useful directories include **/usr/doc** and **/usr/local/doc**.

Many UNIX programs offer on-line help; for most, either "h" or "?" is the "help" command.

5.6. mail

Electronic mail can be sent and received using the **mail** command. At login, the c-shell will tell you if you have received new mail, as long as the "mail" variable is set in your .cshrc file (see the example .cshrc in section 4), and it will keep checking for new mail at intervals specified by that variable. To read this mail, simply type `mail` with no arguments. The program will print the subject lines of some of the recent mail, then its prompt, the "&" sign. Typing `?` or `help` will list all recognized commands. Press `<CR>` to read the messages sequentially. Mail which you have read, but haven't saved somewhere else, is by default saved to the file "mbox" in your home (top-level) directory. You can reread this mail using the command `mail -f`. There are many other options for sophisticated users—see the man pages.

Mail with an argument is used to send mail to other users. After giving a subject line, type your message, and then end it with a `^D` (or a ".") as the first character in a line. There are various "~" escapes—for example if you type "~v" at the beginning of a line, it puts your message in a temporary file on which the editor **vi** is then invoked.[11] "~p", for example, prints the message so far. You can read in a file with "~r", while "~?" lists all recognized tilde escapes. The address to mail someone on the local machine is simply the username, but mail can be sent to remote machines with the general syntax

mail user@host

One way to find out how to send mail to somebody is to have her send mail to you first. Our host address is **physics.cas.usf.edu**.

You may set aliases and options for **mail** in the **~/.mailrc** file. It is important to set the `crt` variable to a positive integral value (*e.g.*, `set crt=5`) else **mail** will not scroll long messages.

Some users prefer alternatives to **mail**: *mh*, *elm*, or *pine*. Whatever one's choice of reader, unsolicited (and often fraudulent) commercial e-mail is a sad consequence of the open, insecure nature of the original, and still universal, mail standard (SMTP, the simple mail-transfer protocol). Fortunately, the **spamassassin(1)** mail filter works quite effectively, in my experience catching over 90% of spam with no false positives. The following two steps will set it up on `physics` (see also `help spam`):

```
mkdir ~/spam
cp /usr/local/lib/sample.procmailrc ~/.procmailrc
```

It is still advisable to check `~/spam/spambox` once a week in case of false positives; to do this in the mail program, use `mail -f ~/spam/spambox`.

[10] This assumes you are using the Berkeley version of `man`. The System-V equivalent is `man -s 2 wait`. You may also use `man -a wait` to see all manual pages on **wait**.

[11] If you have a different favorite editor, `setenv EDITOR` *favorite* in your **.login** will inform **mail** (and a host of other programs).

5.7. UNIX editor: *vi*

We support two full-screen editors: **vi**, the standard UNIX editor, and the lisp-based **emacs**. If you already know EMACS and want to use it rather than *vi*, you may wish to skip this section for now.[12] For people who like mice, **nedit** and **xemacs** are available.

Vi, or *vide infra*, is a full-screen editor designed for program and document editing. It incorporates a line-mode editor, *ex*, which is itself a superset of the old UNIX editor, *ed*.

There seem to be two types of editors in the world: those that are easy to learn and can't do anything (MacWrite, EDT, and Word are examples), and those that take some time to master but which once mastered greatly improve the user's speed and comfort. *vi* is in the latter category.

We have three levels of introduction to *vi*. The first is a file that you may edit on-line (it can't be overwritten, so don't worry). You may wish to start with this. It gives step-by-step instructions. The second (included below) is a short summary of several useful commands compiled by Professor John W. Wilkins of Ohio State. The third, the full reference for *ex* and *vi*, is in the *Users' Supplementary Documents* volume of the **Berkeley 4.4** manuals with the picture of the Daemon on the cover. Another complete reference is the **:help** command within the extended *vi* installed in Linux.

To invoke the on-line tutorial, type

teachvi

5.8. If the terminal starts acting strangely

Some programs put the terminal into a mode in which it misunderstands what's meant by a line and may refuse to echo anything. If this happens, try

```
^Jtset^J
stty sane
```

Remember that ^J means <ctrl>-J. Regular keys such as <DELETE>, <CR>, and ^U will probably NOT work until after you've succeeded with the above command, so if you mistype part of the command, start over with the first <ctrl>-J. If the terminal still misbehaves, you can try various calls to **stty** (which are probably in your **.login**, so "source ~/.login" may help).[13]

5.9. Printing

The preferred command to print a text file is `mpage -2 file | lpr`; this saves paper over the standard command (**lpr**) by printing two sheets on one. Check with College Computing about the default printer; you may specify a different printer with the `-P`*name* option (where *name* is the printer's name, for which see **/etc/printcap**) or with the **PRINTER** environment variable set in your **.cshrc** or **.login** (see section 4)).

5.10. What's Going On with the System

`who` will tell you who's on the system. `finger name` will tell you more about the named user or, in the case of a first name, about all the users sharing the given first name. `finger @theory.tifr.res.in` will tell who is logged in at the Tata Institute. On Berkeley-derived UNIX, or under Solaris if **/usr/ucb** precedes **/bin** in your search path, `ps waux` will tell you what other processes are running on your machine. The *System-V* equivalent is `ps -elf`. The `top` program is a useful alternative to **ps**. `ps` gives the process numbers needed by `kill`; `kill -9` ***process number*** will almost always destroy a process.

5.11. General Information about UNIX commands

A UNIX command consists of the command name followed optionally by some number of arguments:

```
command arg1 arg2 arg3 ...
```

Each UNIX program is responsible for reading and interpreting its command line; although there is no universal

[12] We also have several line editors and TECO. If you know TECO, you don't need this manual. Just read an octal dump of the operating system.

[13] Tilde (~) is expanded by the shell to the path of your home directory.

List of ‘vi’ commands **All commands are in single quotes (‘ ’)**

(To type continuously with 72 characters/line, create .exrc in root dir with cmd: ‘ **set noai wm=8** ’)

Start/edit file:	**‘vi filename’**	
leave editor	write modified file: ‘**ZZ**’	throw away changes: ‘**:q!**’
	write and stay in editor: ‘**:w**’	switch to *newfile*: ‘**:e** *newfile*’
insert new material	just before/after cursor: ‘**i**’ / ‘**a**’	at start/end of line: ‘**I**’ / ‘**A**’
insert a new line	below/above cursor ‘**o**’ / ‘**O**’	
To leave insert mode ‘**(esc)** ’	(push the escape key) Mac users:	‘**(esc)** ’ is upper lefthand key
Undo last deletion/change: ‘**u**’	all deletions/changes: ‘**Qu(cr) vi(cr)**’	(must be out of insert mode)
	Repeat previous command:	‘**.**’ (just a period)
Scroll screen a half screen:	‘**(ctrl)-u**’(↑) ‘**(ctrl)-d** ’(↓) a full screen:	‘**(ctrl)-b**’ (↑) ‘**(ctrl)-f**’ (↓)
To redraw screen:	‘**(ctrl)-l**’	
line no. of cursor?: ‘**(ctrl)-g**’	number lines: ‘**:set nu(cr)**’	remove linenos: ‘**:set nonu(cr)**’
Write lines *n* thru *m* to *newfile*:	‘**:***n*,*m***w** *newfile* **(cr)**’	to append: ‘**:***n*,*m***w**>> *newfile* **(cr)**’

UNIX cmds inside editor: ‘**:!***unix cmd*’ outside editor: ‘**:sh**’ return with ‘**(ctrl)-D**’

Abbreviations: ‘**:ab lt lazy typer(cr)**’ replaces typed word ‘lt” with ‘lazy typer’ everytime it is typed.

Alternately store abbreviations in .exrc file with the syntax: ‘**ab lt lazy typer**’ for each line.

Moving the cursor to position specified by ‘⟨**move**⟩’ (Prefix number to ‘⟨**move**⟩’ for multiple use of cmd.)

move cursor by character:	‘**h**’ (←) ‘**j**’ (↓) ‘**k**’ (↑) ‘**l**’ (→)	word/Word: ‘**w**’ / ‘**W**’ (→) ; ‘**b**’ / ‘**B**’(←)
move cursor to:	end of word/Word: ‘**e**’ / ‘**E**’	‘**W**’, ‘**B**’ and ‘**E**’ sees only blanks
	start/end of line ‘**0**’ / ‘**$**’	screen top/bottom ‘**H**’ / (‘**L**’)
	n^{th} line: ‘*n***G**’	start/end of file ‘**1G**’ / ‘**G**’
	matching parenthesis or brace ‘**%**’	previous/next sentence ‘**(**’ / ‘**)**’
	upto/onto character *ch*: ‘**t/f** *ch*’	back upto/onto *ch*: ‘**T/F***ch*’
Note:	‘**;**’ repeats ‘**t**’ ‘**f**’ ‘**T**’ ‘**F**’ cmd	‘**,**’ ditto in *reverse* direction
	‘previous’ cursor position: ‘ ‘ ‘ ’	‘previous’ line: ‘ ’ ’ ’
put mark *a* at cursor: ‘**m***a*’	shift to mark *a*: ‘ ‘*a* ’	or to line of mark *a*: ‘ ’*a*’
shift to *token*	below *or* above cursor: ‘/ *or* ?*token*’	‘**n**’ repeats ‘/’ or ‘**?**’ cmd

NOTE: ‘\’ must preceed ‘\’, ‘[’, ‘$’, or ‘.’ for these special characters to be recognized in a *token*.

Corrections Commands	General form: ‘**c**⟨**move**⟩ *“new text”***(esc)** ’	Any ‘⟨**move**⟩’ above works.
	This inserts *“new text”* from cursor to	position specified by ‘⟨**move**⟩’.
Additonal correction cmds	substitute some characters: ‘**s**’	a line: ‘**S**’ or ‘**cc**’
	replace from cursor to line end: ‘**C**’	break line to left of cursor: ‘**i(cr)(esc)** ’
cmds not requiring **(esc)**	replace single character ‘**r**’	interchange two characters: ‘**xp**’
join line with one below: ‘**J**’	Indent *n* lines: ‘*n*>>’	remove indentation: ‘<<’

Deletion Commands	General form: ‘**d**⟨**move**⟩’	Delete from cursor to ‘⟨**move**⟩’ position.

(Everything can be deleted into buffers *a* to *z* by appending ‘ ”*a* ’ (say) to the delete cmd.

To recover older lost lines:	‘ ”*n***p**’ recovers text from buffer *n*	From newest ‘1’ to oldest ‘9’

To shift thru buffers: start with ‘1”; cmd ‘**u.**’ removes last recovered buffer and recovers next oldest;

Additional delete cmds	a character at/before cursor: ‘**x**’ / ‘**X**’	
	a line at/above cursor: ‘**dd**’/ ‘**dk**’	rest of line: ‘**D**’

Yanking (copy text from cursor to ‘⟨**move**⟩’ into default [or *a*-*z*] buffers) **Putting** (buffer into text)

general forms yanking:	into default buffer: ‘**y**⟨**move**⟩’	putting: to right/left of cursor ‘**p**’ / ‘**P**’
	into buffer *a*: ‘ ”*a***y**⟨**move**⟩’	below/above cursor: ‘ ”*a***p/P**’
additional yank/put cmds	one/*n* lines: ‘**Y**’ / ‘*n***Y**’	below/above cursor: ‘**p**’ / ‘**P**’

Global commands

substitute:	‘**:s/***token***/***new stuff***/(cr)**’	substitutes *new stuff* for *token*
	‘**:g/***token***/ s//***new stuff***(cr)**’	does it globally
	‘**.,.+***n* **s/***token***/***new stuff***/(cr)**’	substitutes from current for *n*+1 lines

Note: ‘**g**’ at end of any of three lines above multiply substitutes “newstuff” for “token” in a line.

what file is this? ‘**:f(cr)**’	print n*th* line: ‘**:nl(cr)**’	shows all wild characters
read in *newfile*:	‘**:r** *newfile***(cr)**’	*newfile* appears below the cursor
switch to *newfile* inside editor:	‘**:e** *newfile***(cr)**’	useful for yanking buffers between files

Update: August 25, 2006 [jww]

standard, the following general guidelines usually hold.

When a command operates on a single file, the file is entered, often as the last argument. For a command that uses one file as input and another as output, the input filename generally precedes the output filename (unless the output file is specified by an option, usually called '`-o`'). Options are generally specified by a single letter preceded by a minus sign; if one wishes to specify several options at once, one often has the choice of combining them

```
command -abcd
```

or giving them separately

```
command -a -b -c -d .
```

If an option takes an argument, it is generally given as the argument following the option:

`cc -o` **filename** `foobar.c`

The option '`-`', if part of a command's syntax, usually has one of two meanings; which one is clear from the context. It can mean either "substitute standard input (the keyboard, a pipe, or redirected input) for the filename that could go in this position,"

```
ls | cat file1 - file2
```

(concatenates [in this order] file1, the listing of the current directory, and file2)

or "treat the next argument as a filename even though it looks like an option,"

```
mv - -filename1 filename2
```

(On Linux, this second, historical, meaning requires instead a double hyphen, `--`. One more easily masks a minus sign leading a file name with a construct like `mv ./-filename1 filename2`.)

6. Introduction to the Csh

In section 4 we introduced the c-shell (**csh**) as the command-line interpreter but left the reader in the dark as to its operation. Consider what happens when `csh` reads the command line

```
ls -l *.Z
```

The asterisk in the argument is a wildcard character; *.Z stands for every file in the current directory whose name ends in .Z (but does not begin with dot (.)). If the directory contains files a.Z, b.Z, and foo.bar.Z (and other files whose names do not match), the shell will treat *.Z as though the user typed in its place `a.Z b.Z foo.bar.Z`. An asterisk can appear anywhere in a name or by itself; there are fancier wildcards described in the manual. Since asterisk (*) is interpreted by the shell, not by the application program, it cannot be used for programs whose arguments are not files: for instance, `finger *` will probably give gibberish, unless you happen to be in a directory whose entries are names of users (**/home** is such a directory on our system).

Now that the shell has the line `ls -l a.Z b.Z foo.Z`, it cuts the line up at the spaces and puts the pieces in an array eventually to be passed to the program. To determine what program to invoke, `csh` looks at the first of these pieces, called the zeroth argument (the first argument in this case is the option `-l`). The csh first checks this zeroth argument (`ls` in our case) against its list of aliases. If it finds a match (for instance, if we have set `alias ls ls -F` in our **.cshrc**), it makes the substitution and starts over.[14] Next `csh` checks against its list of internal c-shell commands; these include **logout**, **set**, and **alias**. If the command is not one of these, **csh** next checks the PATH environment variable (set by the `set path=(...)` command in the example of section 4). That variable contains a list of directories; the c-shell effectively checks these in the order given for a file of the name given by the zeroth

[14] If the zeroth argument begins with a backslash (\), the backslash is stripped and the alias-checking step is skipped. This provides one way to override an alias.

argument.[15] If the zeroth argument had been a pathname beginning with `/`, `./`, or `../`, of course, this step would have been skipped and the given file executed immediately. The c-shell knows enough not to try to execute a file whose execution bit has not been set; if it comes across one, it will continue looking through the search path.[16]

An executable file may be a list of c-shell commands or a binary file created by a compiler. In the former case, it is good practice to make the first line read

```
#!/bin/csh
```

The subshell called to read the file will treat the line as a comment; it is there to tell the operating-system kernel to run **csh** and not some other shell. After creating a command script for the shell, you must make it executable with the command "`chmod 755` ***filename***"; the 7 gives the owner (you) read, write, and execution permission, while the two 5's give all other users read and execution permission only. The current **csh** can be made to read a file of commands with the `source` command. Sourced files (such as **.cshrc**) need not be executable.

Some c-shell internal commands and expansions

`pushd` *path*	like "**cd** *path*" but also pushes old directory on stack
`popd`	go back to previous directory on stack
`dirs`	show directory stack
`alias` **A B**	make **A** an alias for **B** (only for commands)
`jobs`	show a numbered list of jobs (some stopped by ˆZ, for instance)
command `&`	run *command* in background — lets you do something else
`bg %`*n*	run stopped job *n* in background
`%`*n*	resume stopped job *n*
`kill %`*n*	kill a stopped or background job
`history`	show a numbered list of recent commands
`!!`	redo the last command
`!`*n*`:s/`**a**`/`**b**`/`	redo command *n*, substituting **b** for first occurance of **a**
`!`**string**	redo the most recent command that began with **string**
`set` **a=b**	assign a (string) variable a value
command `$a`	$**a** is replaced by the value of variable **a**
`~fred/`	expands to the path of fred's home directory
`~/`	expands to the path of the user's home directory

Some users prefer an extended c-shell, such as **tcsh**. For many purposes, you may find the Bourne shell, **/bin/sh**, or one of its derivatives, more appropriate for programming than the c-shell. The manual pages describe the syntax fully. Be warned in all shell scripts that spaces may have significance: in particular, lexical elements such as parentheses must often be surrounded by spaces, while equals signs (=) often must not be.

7. Another Editor (Emacs)

In the late 1970's, the programmer Richard Stallman wrote a set of editing macros for TECO so that a user could see the text as she changed it. This version of **Emacs** was sold for several years by the Computer Corporation of America for the RSX and TSX operating systems. Stallman, founder of the Free Software Foundation, later rewrote the entire package in LISP and gave it away for free to anyone who wanted it. The current version, written mainly in C, still incorporates a fully-functional LISP interpreter. For a long time, many different versions of Emacs floated about, each with different default key bindings, something that made learning difficult. Now, however, that

[15] There is a subtlety here concerning hashing. If you create a new executable file after the invocation of the shell, and if that file is not in the current directory, you need to run `rehash` before the shell will be able to find it, even though it is in the search path.

[16] If it then fails to find an executable file (having found a non-executable one), the shell will issue the message `file: Permission denied`.

computers are all large enough to run the full GNU Emacs release, the editor has stabilized.

Emacs supports multiple editing screens and buffers in a single window, formatting, and (optionally) a mouse. Because it includes a programming language, it can perform arbitrarily complex operations on text. Some programmers, using it for e-mail, compilations, and reading news, never leave Emacs, except perhaps to eat.

Unlike *vi*, Emacs has no separate *command* and *text-entry* modes. Any printing character is entered into the text as one types it. Motion, deletion, and other commands are implemented as sequences of control or similar characters, such as `<meta>-V` to move backward one page or `^X^C` to exit. Commands, such as `<meta>-V`, that require a set parity bit can also be entered as `<escape>` followed by the required key, *e.g.*, `<escape> V`. (This has led to the somewhat deprecatory acronym Escape-Meta-Alt-Control-Shift).

The `teachemacs` command launches the Emacs tutorial. From within any Emacs session, the backspace key (or `^H`) opens a help window.

Although **emacs** supports some mouse-based commands, mouse users may prefer **xemacs**. A subset of Emacs as well as vi editing commands can often be found in other programs, such as **tcsh** and **pico**. Included below is a one-page summary of some Emacs commands.

8. Programming

We hope in this section to introduce to the reader some of the programming tools under UNIX. Those who would like to examine these tools at length will be referred to the appropriate texts.

8.1. Why C?

In this short paragraph, I will try to present a few of the best reasons for a FORTRAN-77 programmer to take the time to learn **C**. Once I have convinced the reader to learn **C**, he should refer to the source. There is only one usable text or reference for **C**:

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, second edition (Prentice Hall, 1988)

The book goes through each of the commands of **C** and many common programming tricks and practices. There are numerous examples along the way and a complete reference manual at the end. *The C Programming Language*, written by the language's authors, was considered to define **C** long before the 1989 ANSI standard came out.[17]

Like FORTRAN, **C** translates mathematical formulæ into computer code and performs calculations. Indeed, **C** programs that do nothing but calculations can look very much like FORTRAN. (*"It's possible to write FORTRAN*, i.e., bad code, *in any language*"). Because of the more efficient and flexible way in which C handles arrays and passes arguments, however, good **C** programs can usually run faster than FORTRAN, other things being equal.

C allows several things to happen on a single line, so **C** programs are generally shorter than the equivalent FORTRAN; they are also more quickly written. **C** encourages user-defined data structures called *structs*; like a named COMMON block, a *struct* combines several variables into one, but many different variables can each have the same data type (*struct*), and they can be passed, copied, modified, put into arrays, and referenced like any other variables. *structs* are used in implementing a kind of variable-length one-dimensional array called a linked list, in which new elements may be inserted anywhere without having to move the old elements. *structs* may be nested. Another example of a data structure essential for many scientific applications (but missing from FORTRAN) is the binary tree, useful for sorting items alphabetically, numerically, or by any other criterion. Since objects of interest to the scientific programmer are not only numbers, but have some structure, **C** provides a more natural environment than FORTRAN for coding simulations and calculations.

Unlike FORTRAN, **C** supports a fixed array whose dimension is determined at runtime; see footnote b to the table in the next section.

String manipulation and input/output are simpler and much faster in **C** than in FORTRAN, making the language well-suited to writing interactive programs.

[17] A second C standard was published by the ISO in 1999, adding several minor features. However, many compilers do not implement the changes, so it is still a bad idea to incorporate them into any code one might write. By now (2006), essentially all compilers *do* implement the ANSI standard, so there is no strong reason to write code in the style of the first edition (1978) of *The C Programming Language*.

List of 'emacs' commands **All commands are in single quotes (' ')**

Start/edit file: '**emacs filename**' Since **insert mode** is the default, any cmd must be prefaced by the meta characters **(ctrl)-** or **(meta)-.** The general rule is that **(ctrl)-** affects a single 'object' (e.g. letter), while **(meta)-** affects a larger one (e.g. word)

To leave editor	'**(ctrl)-x s**' saves and stays in editor	'**(ctrl)-x (ctrl)-w**' prompts to write file
	'**(ctrl)-x (ctrl)-c**' exits editor but prompts	you to save file if needed
	'**(ctrl)-z**' suspends editor	return to editor with '**fg**'
Multiply do '**cmd**'	*n* times: '**(meta)-*n* cmd**'	4 times: '**(ctrl)-u cmd**'
Abort command: '**(ctrl)-g**'	Redraw screen: '**(meta)-x redraw**'	redraw and center: '**(ctrl)-l**'
UNIX: running cmds:	'**(meta)-!unix-cmd (cr)**'	(if want shell: '**(meta)-x shell**')
compile	with make: '**(meta)-x compile'**	

HELP: '**(ctrl)-h (ctrl)-h**' enters the help menu
'**(meta)-x describe-function (cr)** *cmdname*' produces description of "*cmdname*"
'**(meta)-x describe-key (cr)** *key pattern*' produces *cmdname* for the *key pattern* if bound.
'**(meta)-x apropos (cr)** *cmdname* ' produces list of cmds containing string "*cmdname*"
'**(meta)-x describe-bindings** ' produces list of all keys bound to commands
Note: '**(space)**' or '?' will exercise command completion feature but **(cr)** still necessary.
Because **(ctrl)-h** *is bound to help, only* delete *deletes (Jove users take note!).*

Shifting the cursor *(arrow keys and scrollbar work too; some commands require* **.tex** *file or* **(meta)-x TeX-mode***)*

shift cursor 1 character:	'**(ctrl)-b**' (←) '**(ctrl)-f**' (→)	'**(ctrl)-p**' (↑) '**(ctrl)-n**' (↓)
shift cursor by a word	'**(meta)-b**' (←) '**(meta)-f**' (→)	
shifting screen	a full screen: '**(ctrl)-v**' (↓) '**(meta)-v**' (↑)	1line: '**(meta)-1 (ctrl)-v**' (↑) '**(meta)-1 (meta)-z**' (↓)
shift cursor tostart/end of	line '**(ctrl)-a**'/ '**(ctrl)-e**'	sentence '**(meta)-a**'/ '**(meta)-e**'
start/end of	s-expression: '**(meta)-(ctrl)-b**'/ '**(meta)-(ctrl)-f**'	paragraph: '**(meta)-{**'/ '**(meta)-}**'
top/bottom of	screen '**(meta)-0 (meta)-r**' / '**(meta)-- (meta)-r**'	start/end of file '**(meta)-<**'/ '**(meta)->**'
search interactive	forward '**(ctrl)-s string**'	reverse '**(ctrl)-r string**'

'Interactive': adjust token by 'deleting' and retyping; '**(cr)**' when successful; '**(ctrl)-g**' to abort
(ctrl)-s *will not work over some serial lines*

Correction commands (There is no 'undo' for deletions but there is for kills – see Yanking)

delete	next/previous character '**(ctrl)-d**'/'**(del)**'	next/previous word '**(meta)-d**'/'**(meta)-(del)**'
	white space '**(meta)-**'	blank lines '**(ctrl)-x (ctrl)-o**'
transpose	two characters: '**(ctrl)-t**'	two lines: '**(ctrl)-x (ctrl)-t**'
new above/below current	line: '**(ctrl)-o**'/'**(ctrl)-e (cr)**' paragraph	break : '**(ctrl)-m**' *or* '**(ctrl)-j**'
capitalize first letter:	'**(meta)-c**' whole word: '**(meta)-u**'	word all lower case '**(meta)-l**'
substitute globally:	'**(meta)-x replace-string (cr)** *old***(cr)** *new***(cr)**'	no query
interactive substitution:	'**(meta)-%** *current-str* **(cr)** *new-str* **(cr)**'	starts from cursor & asks

---

Yanking and putting (use '**(ctrl)-@**' to mark point; interchange mark and cursor '**(ctrl)-x (ctrl)-x**')

kill from point to cursor	'**(ctrl)-w**' into kill buffer	'**(meta)-w**' (without killing)
kill from cursor to	end of line '**(ctrl)-k**'	
" " " "	end of sentence '**(meta)-k**'	beginning of sentence '**(ctrl)-x (del)**'
kill	s-expression '**(meta)- (ctrl)-k**'	*e.g.* from "{" to matching "}"

yank last killed buffer '**(ctrl)-y**'; '**(ctrl)-y**' with repeated '**(meta)-y**'*s* yanks back successive buffers (upto 10 in a ring).

Using multiple Buffers All cmds are prefaced by '**(ctrl)-x**'

create buffer *new* '**b** *new*'	select buffer *old* '**b** *old*'	replace file in buffer '**(ctrl)-r (cr)**'
list buffers: '**(ctrl)-b**'	save some modified buffers '**s**'	kill buffer *old*: '**k** *old***(cr)**' (Be careful!)

Windowing All cmds are prefaced by '**(ctrl)-x**'

divide into 2 windows: '**2**'	switch to next window: '**o**'	
delete current window '**0**'	delete all other windows '**1**'	make window bigger: '**^**'
open 2nd window with:	file *filename*: '**4f***filename***(cr)**'	buffer *oldbuf*: '**4b***oldbuf***(cr)**'

Exception to '**(ctrl)-x**' prefix: '**(meta)-x shrink-window**'

Update: 1 Dec 95 [jww,dar]

The **C++** language builds on **C** by concentrating even more on objects (matrices, windows, databases, Feynman diagrams), providing new and more natural ways to operate on them and to isolate the code that deals with their internal aspects. A matrix-vector multiplication in **C++** might look something like `a = M*x;` .

8.2. section(3)

Section 3 of the manual deals with library routines for **C** and other programming languages. Math functions are generally included in their own library and found in section 3M. To link to the math library, it is necessary to pass the *-lm* switch to the **C** compiler, **cc**. I've compiled a list of some of the more important standard library functions, omitting the ones useful chiefly to the system programmer and those the reader would understand and expect anyway, including everything in the math library. When you wish to use a function, read the relevant manual page with the command

man 3 *function*

(Occasionally, a function will reside on a differently-named manual page, and man will sometimes be confused and not know about it. In this case, use **apropos(1)** to find the name of the correct manual page.)

function	what it does (briefly)
intro	not a function, but an introduction to section 3
fopen	open a file for buffered I/O[a]
fread	read a number of bytes from a file
fwrite	write a number of bytes to a file from a pointer
fseek	go to any location within a file
fprintf printf fscanf	These are to the FORTRAN format statement what the washing machine is to the washboard and wringer (even fortran partisans admit this).
gets fgets	read a line without formatting
getchar getc	read a character
putchar putc	write a character
ungetc	un-read a character that's been read
popen	open a pipe (simple interprocess communication)
strcat strncat	concatenate two strings
strcmp	compare two strings alphabetically
strlen	get the length of a string
malloc	dynamically allocate memory[b]
mktemp	make a unique file name, useful for keeping track of experiments, etc.
setjmp longjump	set a marker in a program and then jump to it later
perror	print an error message appropriate to the last non-fatal error
signal	specify error-handling routines for various sorts of fatal errors[c]
system	execute a shell command from a program (without leaving the program)
ncurses	not one function but forty or so: terminal-independent screen manipulation (`curses` on Solaris)
qsort	efficiently sort an array by any criterion

We summarize some of the public-domain numerical offerings later. The heavy-duty parallel programmer will appreciate multiple threads (**pthread_create(3)**) and support for communication with other machines.

[a] UNIX files do not have records, so there is no restriction on what one may write to them. This should not be viewed as a deficiency. The functions **fwrite(3)** and **fread(3)**, by their syntax, make it nearly trivial to implement files with fixed-length records, but because it is the user, not the system, who decides how to organize the file, calls to these routines may be mixed with **fprintf(3)**, **fputs(3)**, **getchar(3)** *etc.*

[b] **C** supports three strategies for storing variables. Static storage corresponds roughly to the only kind available in FORTRAN: the exact size needed must be known at compile compile-time, and if inside of a routine, there is only one version of a datum, even if the routine is called recursively many levels deep. Automatic storage (stack) exists only inside of routines; when a routine exits, its automatic storage vanishes. Dynamic memory combines the advantages of the two. It is non-volatile and global, but the user takes only what he needs. One common application of **malloc(3)** is the allocation of variable-sized arrays.

[c] This finds at least two applications in long-running simulation programs: it may be used to trap the termination signal sent to all processes just before a system shutdown, and in conjunction with **alarm(3)** or **setitimer(2)** it can run a subroutine at regular intervals to save intermediate results for interactive examination and for eventual recovery in case of a system crash.

8.3. section(2)

Section 2 of the manual deals with system calls. The strictly scientific programmer should need to refer to this section only occasionally, for example, if he wants to have two processes talk to each other; scientists sometimes have to write system programs, however, and the graduate student lucky enough to have been assigned to write a device driver for a new tape drive or a low-level graphics package will need to learn much in this section.

8.4. make

When developing a program, you will often make a series of small changes and want to recompile only the parts of the program modified by the changes. When the program is in three languages, uses four pre-processors, and lies in five directories, this can be unwieldy. UNIX provides a relatively simple utility, *make*, that determines for you what needs to be recompiled. For example, you may have several different source files that need to be recompiled every time you make changes in another file which is included in all of them (with #include in C or \input in TeX). The best way to see how this works is through the following sample make file (these are called either Makefile or makefile, normally):

```
# Sample Makefile
#
# This is the simplest non-empty form:
#
client: client.c;  cc client.c -o client
#
# The command ‘‘make client’’ effects the above compilation, but only if client.c has been changed
# since the last compilation. This would actually be done by default, even if there were no makefile.


# Slightly more complicated:
#
CFLAGS = -O
LDFLAGS = -O
LIBS = -lX11 -lXt -lsocket -lm -L/usr/local/lib -lsomelib
vec:   vec.o windows.o
       cc $(LDFLAGS) -o vec vec.o windows.o $(LIBS)    #the TAB is required
windows.o:  windowdefs.h
vec.o: windowdefs.h
#
# Here, "make vec" notices that vec depends on vec.o and windows.o, and so first tries to make
# those. It then notices that they depend on windowdefs.h, and also knows by default that they
# depend on the source files vec.c and window.c respectively, so those will be compiled using
# the CFLAGS option defined at the top. The LIBS variable collects the libraries needed for this
# particular program.
#
clean:
       rm *.o *.dvi *.log
#
# "make clean" removes all the object, dvi, and log files lying around in the current directory.
```

Make will seem indispensable once you start handling programs that depend on more than one or two source files. For more information, see the man pages, or look over other people's make files for hints.

8.5. Compiling and linking a C program

The UNIX C compiler looks at the filename suffix to determine what a file is. **.c** indicates a C program source, **.o** an object file, **.a** a library of object files (see **ar(1)** and **ranlib(1)**), and **.f** indicates a FORTRAN source. To compile a short (one-file) program, the usual procedure is

```
cc -o name name.c
```

—or—

```
cc -c name.c
cc -o name name.o
```

The first line compiles the source code **name.c** into the (binary, not-human-readable) object file, **name.o**. The second line invokes the linker to create the executable **name** from the object code. The linker is responsible for converting references to external libraries into addresses and for combining multiple object files, as in the second Makefile example earlier. It is possible to stop at various other intermediate stages, such as after preprocessing or before assembly.

The **cc** compiler on Linux is actually a link to the Free-Software-Foundation **C** compiler, **gcc**. To compile old code using Kernighan's and Richie's original **C**, use the **-traditional** option. We also have the commercial Portland-Group compiler, **pgcc**.

Include the **-g** option to incorporate debugging code; the debugger is **gdb**. See the manual page and on-line help for information. When a program has been debugged, recompile with the -O option instead of -g. This will make it run faster. (Further optimization flags may make it run even faster, but be careful of optimizer bugs.)

While the most-used functions are in the standard C library, others, including math functions, need to be linked explicitly. Give the **-lm** option *after* all source and object files to get math functions. (See the example in the section above on **make**.)

8.6. Other Languages

We have compilers for C++ (**g++(1)** and **pgCC(1)**), FORTRAN-77 (**g77(1)** and **pgf77(1)**) and FORTRAN-90 (**pgf90(1)**). The compilers recognize files containing sources for these languages by their extensions (.c, .C, or .f), and it is easy to mix languages; when creating the executable file, invoke the compiler appropriate to the language of your **main** routine. Be sure as well to understand how variables are passed between routines in each of the languages you use and whether an underscore (_) needs to be appended to symbol names. It is possible to write in assembler (**as(1)**), and we might have a **lisp** interpreter somewhere.

9. A quick preview of some of UNIX's tools

Many, although certainly not all, of UNIX's tools take the form of "filters," reading from standard input and writing to standard output (both default to the terminal). Pipes (|) may be used to connect the output of one program to the input of the next, and I/O redirection (> and <) may be used to write to and read from files (see also section 5.3). The following example uses the simple filter, `tr` (translate), to convert all lowercase letters in a file to uppercase before sending the file to a pattern matching program, fgrep, that will output all the lines that contain the word "SUBROUTINE." The output of fgrep is then piped to a program (word count) that will count how many lines were printed. The answer, a number, is then appended to the end of a file called foobar.tmp. (UNIX hackers will note that `fgrep -i` could be used to replace the `tr a-z A-Z`; this is just for demonstration purposes.)

```
tr a-z A-Z < filename | fgrep SUBROUTINE | wc -l >> foobar.tmp
```

More complicated examples of the use of filters can be found in /home/5156/examples/scripts.

9.1. grep

grep is a program that prints only those lines in a file that match a given pattern. **fgrep** is easier to to use when the string being sought contains funny characters (which **grep** might interpret as directives), and **egrep** is a more memory-intensive but faster version with more options. Other operating systems call their implementations of **grep** things like "match" or "find." Regular expressions provide a complicated pattern-matching capability described fully by the **ed(1)** documentation As an example of its power, I once used regular expression matching to translate header files between FORTRAN and an assembly language.

9.2. awk

awk is a pattern-matching and data-base manipulation program. **awk** programs can be as short and simple as one line or as long and complicated as a **C** program; in fact, the syntax of **awk** is similar to that of **C**. The manual page **awk(1)** describes the language in full, and there is a tutorial in *User's Supplementary Documents* in the BSD 4.3 documentation. The following example will add a column of numbers:

```
awk '{sum+=$1} END {print "The sum is ",sum}'
```

awk has in addition all of the pattern-matching power of **grep** and **sed**. Most systems now have an extended **awk** called **nawk** and GNU's faster implementation, **gawk**.

9.3. sed

sed is a non-interactive editor, loosely based on **ed**, designed for use as a filter. The manual page, **sed(1)**, explains the syntax, and there is a tutorial in *Users' Supplementary Documents.*

9.4. m4

m4 is an alternative to the standard **C** preprocessor, cpp. Either **m4** or **cpp** may be used in conjunction with any language to expand macros and allow for conditional compilation.

9.5. dc and bc

dc is the desk-calculator tool. It emulates an arbitrary-precision RPN calculator and is described in the manual page, **dc(1)**, and in a reference manual in *User's Supplementary Documents.* **bc(1)** is a preprocessor for **dc** with a **C**-like interpreted interface.

9.6. a small number of the others

Here's a small sampling of some of the others.

`diff`	differences	intelligent comparison of files
`sort`		sort and merge
`join`		a horizontal `cat`
`tee`	T connection	copy output to two places
`uniq`	unique lines	filters out repeated lines
`gzip`		make a file smaller by bit-packing
`gzcat`		type a compressed file without uncompressing it
`nice`		run something at lower priority
`ar`	archive	create/update/extract files in archive
`tar`	tape archive	similar to ar; not limited to tapes
`strip`		erase name table from binary
`od`	octal dump	octal, hex, decimal, and/or alpha dump
`nm`	name	print contents of name table in unstripped binary
`file`	file type	gives educated guess as to file's purpose
`date`		time and date
`factor`		factor a big integer
`find`		do searches down a directory tree
`locate`		easy-to-use flat file finder

Read manual section 1 to find out more. The only way to master UNIX is to skim through this section, cover to cover, stopping at pages that seem interesting.

9.7. some local commands

We've added a few small but useful utilities on *physics.*

`physrev`	Enter `lynx` to get a paper from *The Physical Review.* The arguments are the section (a, b, c, d, or e, or l for Letters, m for *Reviews of Modern Physics*, or blank [`""`] for pre-1970s papers), volume, and starting page or article number. For example, `physrev b 65 024201`.
`cond-mat`	Enter `lynx` to read a paper from Paul Ginsparg's preprint archive, `arxiv.org`: for example, `cond-mat 0601033`.
`google`	Enter `lynx` with a search query to `www.google.com`; note that certain characters may need to be protected from unwanted interpretation by the shell. Example: `google '"University of South Florida"' physics`

`usfinger`	Find all people at USF with the given last name, *e.g.*, `usfinger killinger`. The database includes e-mail, telephone, affiliation, and office, although the information isn't always current.
`librarian`	Find a library on the system with the given symbol: `librarian printf`. This is particularly useful when linking object files generated by different compilers results in a long list of unresolved references.
`help`	help on various topics

In addition, I've set up single-character commands `<BS>` and `<DEL>`. These `stty` shortcuts are useful in case the rubout key isn't set correctly: type the rubout key at the beginning of a line, then `<CR>`. The rubout key should now work.

10. Text Formatting

We support two similar but mutually incompatible typesetting systems, **troff** and **TEX**. Almost all scientific papers are made in **TEX**, while most documentation (including this manual) is produced in **troff**. Many journals, including those published by the American Physical Society, provide packages that sit on top of **LATEX**, a large set of macros that add some structure to **TEX**.

There are also a number of filters, such as **colcrt** and **fmt**, that are helpful for less formal text formatting. **fmt** is particularly useful when called from within an editor (*e.g.*, "`1G!Gfmt -75`" in vi).

10.1. nroff / troff

troff is the standard UNIX typesetting system; **nroff** takes **troff** input but spits out readable text instead of instructions for a phototypesetter. The **man** pages are written in **nroff**; Kernighan and Ritchie's *The C Programming Language* was set in **troff**, as was *The Physical Review* in the late 1980s. On Linux systems or other with GNU **troff**, one generally uses the **groff** command. **troff**, as a relatively low-level typesetting language, is difficult to use without macro packages. The two most commonly-used general purpose packages are **-me** and **-ms**; manual pages are prepared with **-man**. Equations and tables are handled with the preprocessing filters **eqn** and **tbl**.

While **troff** is moderately powerful and perhaps easier to learn, **TEX** (or **LATEX**)) has become the typesetting language of choice.

10.2. TEX/LATEX

TEX is a powerful typesetting language written by a well-known computer scientist as the practical expression of many of his personal views of programming. Despite this, it is possible to write papers in **TEX**. His own documentation, *The* **TEX***book*, is more of a tutorial/manifesto than a reference manual, for which I recommend

Paul W. Abrahams, **TEX** for the Impatient (Addison-Wesley, 1990)

There follows a sample bit of **TEX** with comments after the percent signs (%).

```
\nopagenumbers % note that TeX's commands are as verbose as troff's are terse

This is a short demonstration written in \TeX.  Macros and directives consist of words (letters, no digits or
special symbols) preceded by a backslash ($\backslash$) and terminated by anything that isn't a letter.

New paragraphs begin after a blank line; it is possible to change the indentation, margins, and so on.  Plain
\TeX\ contains no facility for sections with headings, subsections, and so on, but the user can write his own
macros for these.  Simple equations, such as $ E ~=~ mc^2 $, can be set inline, while more complicated
expressions require a bit more work:
% a simple math equation:
$$ \int_{-\infty}^{\infty} \delta(x) f(x) \, dx ~ = ~ f(0) $$
% a medium-size math equation:
\def\bold#1{{\bf#1}} %this defines a macro with one argument
\def\unit#1{{\bf\hat#1}}
$$                  % because vcenter is allowed only in math mode
\vcenter{%          % because halign isn't allowed in math mode
\halign{%
\hfil$#$\tabskip=2em&$#$\tabskip=0sp\hfil\cr%
{\bold k^0 = L^{-1} \unit x} & {\bold a^0 = L \unit x } \cr
{\bold k^1 = L^{-1} \unit y} & {\bold a^1 = L \unit y.} \cr
}%end halign
}%end vcenter
$$%end math mode
```

To run **TEX** on **file.tex**, give the command

```
tex file.tex
```

If the **.tex** suffix is omitted, **TEX** will assume it. The **TEX** processor is annoyingly verbose. It will stop and ask you what to do if there is an error. Error messages generally are unhelpful. Learn to find the lines beginning with a small ell, a dot, and a number: **l.10** means the error occurred on line ten. The best thing to do when **TEX** finds the error is to type **x** to leave right away; fix the error, then run **TEX** again.[18] Do not pay attention to any suggestions it makes of ways it could fix errors by itself.

Once you have been successful, **TEX** will reward you with an output file called **file.dvi**, as well as something called **file.log**, which contains its opinions of your file and the color of your window shades and how they clash with your carpet, and some thoughts on the superiority of computers over organic life forms. You may view the **.dvi** file with

```
xdvi file.dvi
```

To print the file on the laser printer, issue the command

```
dvips -f < file.dvi | lpr
```

dvips converts the **.dvi** file into PostScript, the language spoken by laser printers. It is also possible for humans and drafting programs (such as **xfig**, see below) to write in PostScript, and the **epsf** package puts PostScript pictures in the middle of a **TEX** file.

Most scientific papers are prepared in **LATEX** rather than "plain" **TEX**, although since version 2E, nearly all **TEX** commands also work inside **LATEX**. A good introduction and reference is

H. Kopka and P.W. Daly, *A Guide to* **LATEX**, 3rd ed., Addison-Wesley, 1999.

In addition, there are examples, tutorials, and two full reference books (in PDF format) in the directory `/usr/local/doc/tex` on `physics`; see especially `workshop1205/resources.dvi` under that directory.

LATEX was meant to be a "markup" language, in the sense that the user would specify logical structure, for instance sections and equations, rather than how those things would appear on the page. This is an advantage in that many things are already well thought out but adds additional layers of complexity if the user wants to do something new or isn't satisfied with the final appearance. A simple example illustrates the difficulty in writing a true markup language for typesetting: some journals specify references with superscripts, which should appear after puctuation,[19] while others use bracketed numbers, which need to go before punctuation [42]. **LATEX** does not handle this in an automated way.

11. Graphics

Most users eventually have something they want to see plotted on the screen or on the laser printer, whether it is data, the output of some big program, or just a function to be visualized. There are different plotting programs available for the various different purposes, some of which are outlined below. The plotting is usually done in two stages (sometimes more) connected by a pipe, in which the first stage converts the data (ascii format) into some standard plotting format, and the second stage consists of a "filter" which actually does the plotting to the screen or printer.

11.1. Filters

Many of our plotting packages (as well as user programs) produce output in the **plot(5)** format (say `man 5 plot` on a BSD system, not Linux, for details). All it knows about are points, lines, and labels — PostScript is much more sophisticated.

The **xplot** filter converts **plot(5)** input to a picture on an X-windows screen, although it is also possible to use the *plot -T tek* filter in conjunction with the Textronix mode of an xterm window.

The command *lpr* sends things to the laser printer, which knows how to handle PostScript nicely, but doesn't know about plot(5). The filter *laser* converts plot(5) to PostScript and sends it to your favorite printer.

11.2. Plotting programs

[18] Alternatively, type **e** to get from **TEX** directly into an editor.

[19] like this

11.2.1. axis

This is the most basic of the plotting programs. It converts two columns of data to plot(5) format and does fancy labels, too. The dvi file **/usr/local/doc/axisdoc.dvi** documents axis. As an example, if "foo" contains two columns of data,

```
axis < foo | laser
```

should print a graph on the laser printer, while

```
axis < foo | xplot
```

will do the same on the screen.

For a demonstration, look in the directory **/home/5156/demo/axis**. (The default Postscript output of laser(1) prints and displays fine but does not work with the epsf package of **TEX**, for which invoke `laser` with the `-T` flag.)

11.2.2. plot3d

Similar to axis, but *plot3d* does 3-dimensional plots. It also does contours. The labeling is not so nice, and other things don't work well. Like axis, it was written by a graduate student at Cornell many years ago. Both programs need sorely to be rewritten. See the man pages, or invoke it with the "-h" option.

11.2.3. interactive plotting programs

Some people prefer a point-and-click interface to command-line filters. The interactive plotting program **xmgrace** (formerly called xmgr/xvgr) is the favorite of the solid-state theory group at Los Alamos, while that at McMaster prefers "supermongo" (**sm**).

11.2.4. gnuplot

While it does not produce publication-quality output, **gnuplot** works very nicely for finding out what, for instance, $1/\Gamma(J_0(x) - J_1(x))$ might look like.

11.2.5. mathematica and maple

`Mathematica` is the fanciest of all the plotting programs. It does all kinds of symbol manipulation too (see below), but it is really most useful for creating plots of functions that you want to visualize. The badly-designed line interface is accessible as `math`, while the even-harder-to-use point, click, and control-carriage-return interface is `mathematica`. *mathematica* can plot 2-dimensional data, contours, 3D plots, and 2D and 3D graphics with shading and other fancy attributes. It also makes movies. Buy the book. (`Maple` does most of the same things as `Mathematica`, just differently.)

11.2.6. matlab and octave

Matlab concentrates on machine-precision arithmetic rather than symbolic manipulation, making it faster than **mathematica** or **maple**. Although originally written for classroom use, many researchers now find it the easiest way to play with problems in linear algebra, signal processing, and several other fields for which special add-on packages are available. We also have a free **Matlab** re-implementation called **octave**.

11.2.7. IDL

The Interactive Data Language (IDL) is designed for the quantitative analysis and manipulation of images and has found wide use in biophysics and astronomy; at the IDL prompt, enter `?` to bring up the help screen.

11.3. Lower-Level Graphics

For fancy applications, especially those requiring speed, it may be necessary to use the low-level X-windows programming packages *xlib*, *Athena widgets (Xaw)*, or *Motif widgets*. These are messy and ill-thought-out.

11.4. Higher-Level Graphics

We support a MacPaint-type drafting and drawing program called **xfig**. Its plain *postscript* output may work better with epsfig in **TEX** than its *encapsulated postscript*.

12. Symbolic Manipulation with Mathematica

The subtitle of *Mathematica*, the book describing Stephen Wolfram's over-arching program, is "a system for doing mathematics by computer." While this may be something of an exaggeration, problems involving nearly trivial manipulation of a large number of symbols are ideally suited to symbolic manipulation. It is often wise, however, to try to simplify an expression or problem by hand before feeding it to *mathematica*. The program is also useful for plotting (section 11.2.5) and helpful with numerical work, although slow.

Mathematica is SMP's successor. *Maple* is a functional equivalent, while I haven't seen **Macsyma** for many years.

See the section on X windows below for general information on bringing up graphical programs.

13. Numerical Work

In addition to *Mathematica*, we have the LAPACK library, combining what used to be called EISPACK and LINPACK. The routines, originally written in Algol, have been translated to FORTRAN but can be called from any language.

The book *Numerical Recipes* by Press, Flannery, Teukolsky, and Vetterling provides a good introduction to many common numerical algorithms, but I discourage the use of any of their routines. The **C** functions are poorly written (in the FORTRAN style), and some of the routines in both languages are reported to have bugs. Many of their routines are watered-down versions of LAPACK or other freely-available codes. Use the book to understand the algorithm, then implement it yourself or use the thoroughly debugged and optimized versions of LAPACK, *etc.*

The text on *Matrix Computations* by Golub and van Loan is very useful for both dense and sparse techniques.

14. Parallel Processing

The course server, *physics.cas.usf.edu*, has two CPUs; in principle, they can work in parallel through the **pthread** library, through the Message-Passing Interface, **MPI**, or through a higher-level language such as **Planguage** or **Cilk** (not yet installed here).

14.1. Condor

One of the most common, and certainly the simplest, form of parallel computation is what computer scientists refer to derisively as "embarassing parallism:" running many copies of the same code simultaneously and independently with different parameters. Condor is meant for this type of parallel work, where the different processes do not need to communicate with each other. The Linux machines in Physics Room 102, which normally act as terminals to the physics server, also serve as the nodes in a parallel Condor cluster served from `physics`. See the topic `help condor` for documentation and examples.

14.2. IRCE

USF Academic Computing runs a number of production parallel-cluster clusters through the Integrated Research Computing Environment (IRCE), duplicating the software used on current supercomputers such as the Cray XT3 and the IBM Blue Gene.[20] In addition to symmetric multiprocessing (one machine, many processors) and homogeneous clusters, **MPI** and its competitor **PVM** can also control heterogeneous clusters. See `http://rc.usf.edu` for documentation.

15. X Windows

A windowing system, at a minimum, puts several terminal screens on one very large screen and provides for moving windows over and under other windows. All current windowing systems also provide rodent support and have become bloated with hundreds of obscure computer-science-type protocols for manipulating windows and abstract graphical objects. Xerox, of all companies, seems to have come up with the idea in the **smalltalk** operating system. Apple and Microsoft later sued each other over which of them was the first to plagiarize it. Two commercial and one free windowing system appeared in the mid-1980's, Dec Windows, Suntools, and X Windows. (Sun also made something called NEWS that tried to be Suntools and X at the same time.) Happily, all but X Windows have now died.

[20] Proposed definition of supercomputer: a computer that takes the definite article. Yesterday's supercomputer is about as powerful as today's p.c., or maybe toaster oven, although much cooler.

X Windows (a name the writers of X Windows deprecate, favoring *The X Windowing System*) is distributed: you can run a program on one CPU and have it send graphics (not just text) to your host machine. If you use the **ssh** command to log in to **physics**, the `DISPLAY` environment variable, which tells the remote machine where to send the graphics, is set automatically to point to a proxy, incidentally bypassing the need to call the insecure **xhost** command (so don't). (From some systems, including Cygwin, it may be necessary to call **ssh -Y**.)

For further information on X, consult the manual pages on **xterm**, **X**, and **dwim**.

16. The Internet

Computers may still not all talk the same language, but at least they've agreed on a protocol (alphabet?) for communication. Gone and unlamented are UUNET, BITNET, DECnet, and Berknet. The "inter" in "Internet" comes from the various physical networks (*e.g.*, Sprintnet, Nysernet) it connects. Tools for remote login include **ssh**, which is to be used in preference to **telnet** when possible both because it is far more secure and because it passes terminal information. The file-transfer program, **ftp**, is useful for copying files and directories between machines. The *world-wide web* holds gems of useful information buried deep in layers of video-game dross. **Lynx** provides a fast keyboard interface, while **netscape**, **mozilla**, and **firefox** let the user point and shoot with the mouse. The amateur detective can often use **finger** and **nslookup** to advantage when seeking electronic mail addresses.

16.1. Logging in Remotely

The College computer administrators have set our computer to allow incoming logins by one protocol only: **ssh**. Other methods, such as **rlogin**, **telnet**, and **ftp**, have been disabled for security reasons (although they still work for connections *leaving* our computer). The secure shell is very easy to use from another Unix machine (including Linux): simply type `ssh physics.cas.usf.edu -l` *yourlogin*. Then enter your password. Never use an insecure channel to log in from computer A to computer B, then **ssh** to computer C: since the link from A to B is not encrypted, a snooper sitting between A and B will still be able to see your password for C. Similarly, if you use something like **telnet** to log in from our course machine to a machine D, do not use the same password on machine D as you have on the course computer. More information on **ssh** is available with the command `help ssh`.

Using **ssh** from a PC/Macintosh involves more work and depends on the specific **ssh** client's implementation.

17. Our Physical Setup; Computer Etiquette

The course computer, `physics.cas.usf.edu`, is a Dell Power-Edge 1800 with two 3.00-GHz Xeon processors, each with 2 MB of on-chip cache. Currently, the machine has 3 GB of RAM and approximately 136 GB of RAID-5 disk space (soon to be increased). For up-to-date information on system configuration and capacity, try **sysinfo(1)**. For current resource usage, **top(1)** packages the most important information in a format that updates every few seconds.

If users try to run programs requiring more memory than physically available, the machine will use swap space on the disk. This should be avoided, since random access to swap space will cause the computer to thrash back and forth between disk and RAM; a thrashing computer may run a factor of 100 more slowly than normal. If it is absolutely necessary to address swap space, there are tricks to reduce thrashing. However, none of the projects in this class should require such large amounts of memory. If **top(1)** shows your program's memory consumption steadily increasing without limit, you may have a memory leak inside a loop, meaning that you have used **malloc(3)** or a similar call and then forgotten to call **free(3)**.

The *load average* displayed in moving averages by **top(1)** and **uptime(1)** approximates current CPU usage. On a two-CPU computer, a load average of 2.00 indicates full use of both processors with no unsatisfied demand. An average of 4.00 would indicate twice as much demand as available processing capacity, under which circumstance each running process will receive only half of a CPU. On a two-processor machine, such as `physics`, **top** shows a single-threaded process using 100% of one CPU as taking up 100%.

CPU-intensive programs that will go for more than several minutes should be run at low priority: "`nice +10` *your_command*" works adequately. The more "nice," the lower the priority of your program (the maximum niceness is 19). As you can verify with **top(1)**, a nice'd program gets a smaller percentage of CPU on a busy system. However, if the load average permits it, even a nice program will get the full CPU. If you use a Bourne-derived shell, use a minus sign instead of a plus sign with **nice**.

College-of-Arts-and-Sciences Computing runs our server for the physics department; Mr. Joel Woodman is system manager.

17.1. disk space and backups

Somewhat over 30 GB of disk space is allocated for user accounts; use **du(1)** in your home directory to see how much you're using. If you need extra space for data, you may `mkdir /scratch/`*yourname* to create a directory on a file system that has more room but is not backed up. You may also wish to compress files you don't need immediately with **gzip(1)** or **bzip2(1)**: depending on the data, the program can cut 20% to 70% off the size of a file. (**Gzip** replaces **compress** which replaced **pack** which replaced **compact**). Be sure also to `rm core` if a program bombs; the **csh** command `limit coredumpsize 0` will prevent core dumps entirely.

The `/home` filesystem is part of a RAID-5 array, meaning that the failure of any one physical disk will not result in the loss of any data. CAS Computing conducts daily backups to tape, but users should not need to request them, since we also keep complete twice-daily images (taken at noon and midnight) under `/backup`. For example, the directory

```
/backup/20060815120001/home/5156
```

contains an image of `/home/5156` taken at noon on 15 August, 2006. The number of images that can be kept depends on disk-usage patterns, but the system manager will try to keep at least a week's worth (currently, backups go back one year). The `/backup` filesystem is read-only.[21]

[21] One might wonder how it's possible to keep hundreds of complete images of `/home` on a filesystem that's only about twice its size. The trick takes advantage of the fact that most user files do not change very often, so hard links, which take up negligible disk space, can be used. See **rsync(1)**, especially the `-a` and `--link-dest` flags.