SunOS for Programmers

Student Guide

Revision F
July 1, 1989

UP-210-01-F

Credits and Trademarks

Sun Workstation is a registered trademark of Sun Microsystems, Inc.

SunStation, Sun Microsystems, SunCore, SunWindows, SunView, DVMA and the combination of Sun with a numeric suffix are registered trademarks of Sun Microsystems, Inc.

UNIX, UNIX/32V, UNIX System III and UNIX System V are trademarks of AT&T Bell Laboratories.

Table of Contents

Course Description

Overview:

This course is a re-introduction. The course first presents an accelerated introduction to SunOS which includes an overview of the SunView interface, basic SunOS commands, navigation of the filesystem, and use of textedit. It then quickly moves into presenting the major concepts and tools that programmers need for maximum production efficiency while coding in the SunOS environment.

Through lecture and hands-on exercises, students learn how to compile and link C programs, the basics of program maintenance and security, and how to analyze C program performance. The course also covers debugging programs, information on SunOS process virtual address space, executable file formats, and the fundamentals of how paging, swapping and process management are implemented under SunOS.

Although the course examples and exercises are all written in C, programs used in the lab exercises are provided. Students will not have to write any C programs. The focus of the lab exercises is tool functionality. Students who are not C programmers benefit from exposure to C syntax in the examples used.

All programs used in the lab exercises are provided.

Prerequisites:

This course is for students who have a programming background.
Before attending this course, students should be able to:

* program in at least one high-level, compiled language.
* demonstrate a basic conceptual understanding of source-code configuration management.
* demonstrate an understanding of process creation and management fundamentals.
* demonstrate an understanding of the concept of virtual memory.
* demonstrate an understanding of program compilation and linking concepts.

Duration:

Five days

Objectives:

Upon completing this course, students should be able to:

* Use the SunOS windows interface.
* Use SunOS commands to create, maintain, and modify files.
* Use the C-shell features and program a basic interactive script.
* Restrict and grant access to executables and other files.
* Construct regular expressions.
* Use regular expressions to search and/or edit files with grep, sed, and awk.
* Use the basics of the awk programming language.
* Describe how a process is created in SunOS.
* Identify the virtual address space of a SunOS process.
* Describe the virtual image of a SunOS process.
* Describe how SunOS handles paging and swapping.
* Time a program's execution.
* Perform static analysis of a C program in SunOS.
* Perform dynamic analysis of a C program in SunOS to both the function and statement level.
* Describe a system in terms of dependencies and translation rules.
* Maintain and control successive versions of source code.
* Build the latest or a previous version of a program.
* Compile and link C modules utilizing compile and link options.
* Link Pascal and FORTRAN object code with C programs.
* Use the C preprocessor and its options.
* Debug programs using dbx and dbxtool.
* Perform postmortem debugging of a process.

Module 1
Introduction to SunOS®

Objectives: Upon completion of this module, the student should be able to:

- Define the four major components of SunOS.
- Describe the structure of the SunOS file system.
- Log into a system and log out.
- Utilize the *man* command to access the on-line documentation.

Evaluation:

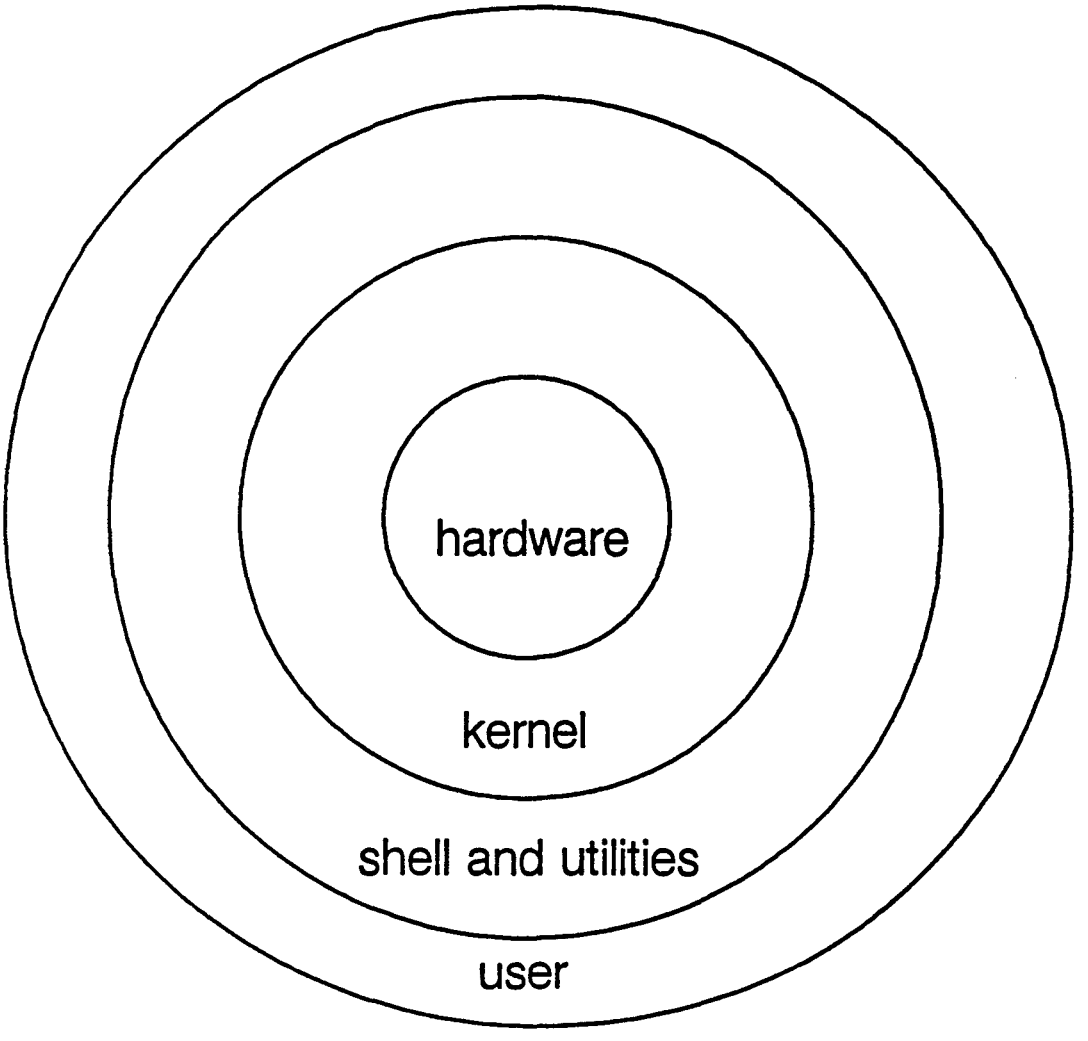
In-class review questions.

Reference information:

- *Getting Started with SunOS: Beginner's Guide* (p/n 800-1703-10), Chapters 1 - 3, 10, Appendix B.
- *Self Help with Problems: Beginner's Guide* (p/n 800-1705-10), Chapter 2.

Major Components of SunOS

- Kernel
- Shell
- Filesystem
- Files



Major Components of SunOS

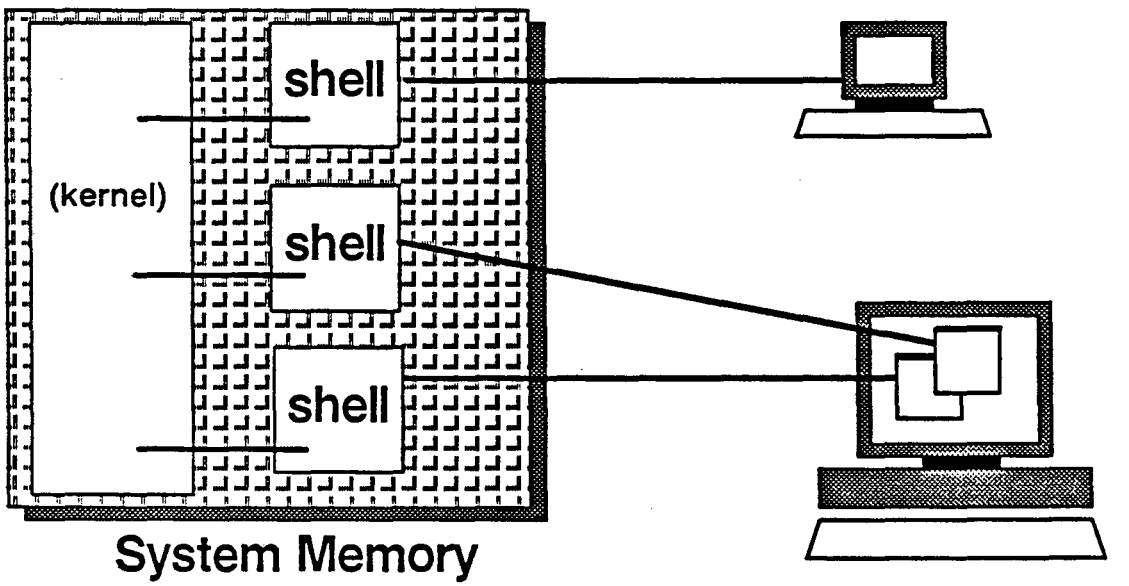
The *kernel* is the primary control program providing user–transparent service to users' applications. The kernel handles such things as device management, process scheduling, etc. It is the part of the operating system that controls the hardware. SunOS utilities interface to the kernel through *system calls.*

The *shell* is a utility and provides an interface for the user. When you enter a command, the shell interprets the command and sends it to the kernel to be executed. The shell is also an interpreter for a powerful programming language.

The *filesystem* is the organizing structure for data.

The Shell

- There are three types of shells common on UNIX systems:
  - C shell (*csh*)
  - Bourne (*sh*)
  - Korn (*ksh*)



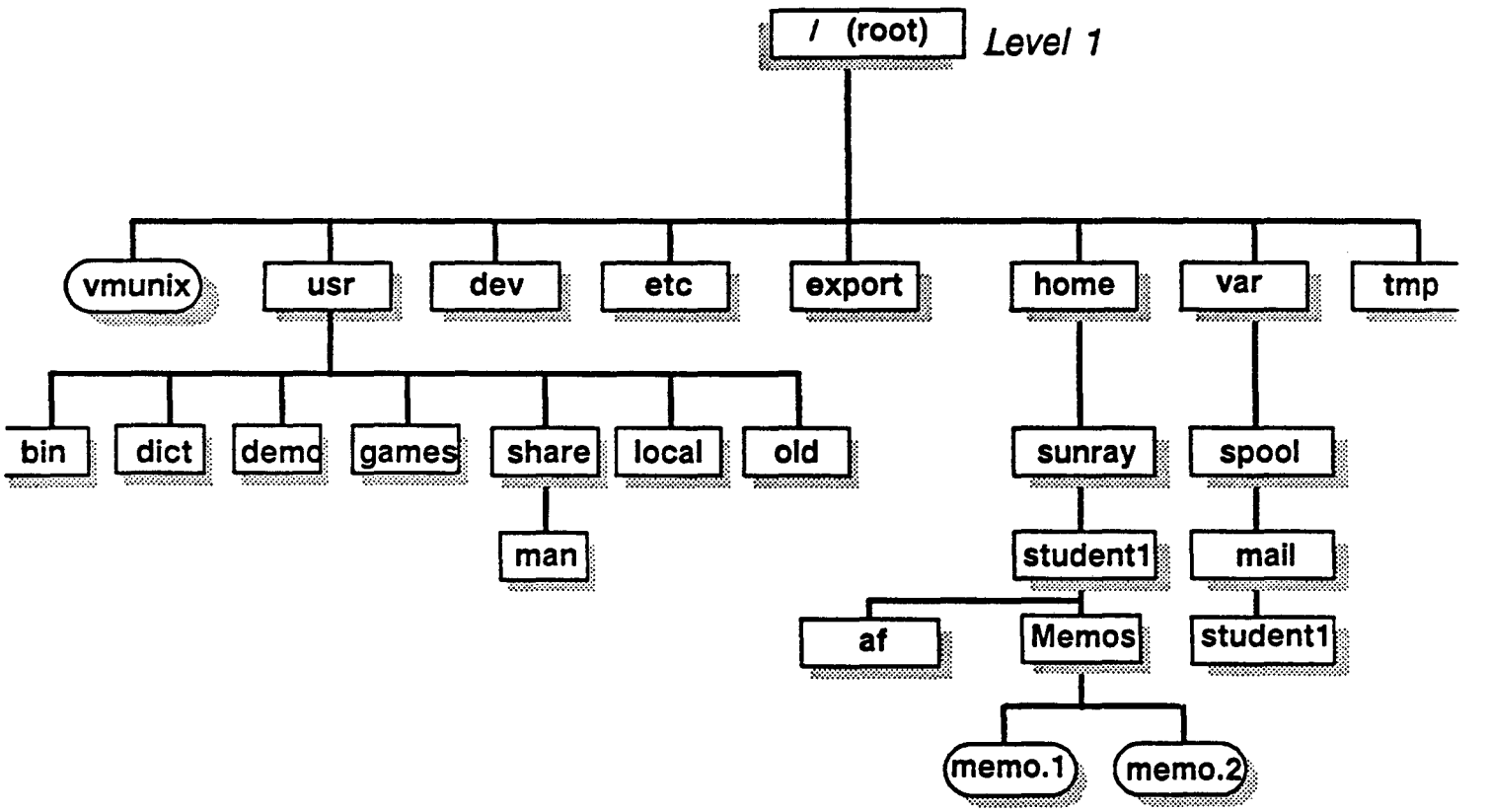
The Shell

The C shell is the primary shell used on SunOS systems. It is known for its interactive command interpreter features, such as *history*, which keeps a record of commands that the user has executed, and *aliases*, which allow the user to set up his/her own new commands. The C shell is standard with most University of California at Berkeley (BSD)–based versions of UNIX.

The Bourne shell is the standard shell for System V–based versions of UNIX. It has programming features, as does the C shell, and is often used by programmers and system administrators.

The Korn shell includes features of both of the other shells. In addition, the Korn shell has command–line editing capabilities and additional programming capabilities, such as report formatting. The Korn shell is not included in the current SunOS release.

The Filesystem

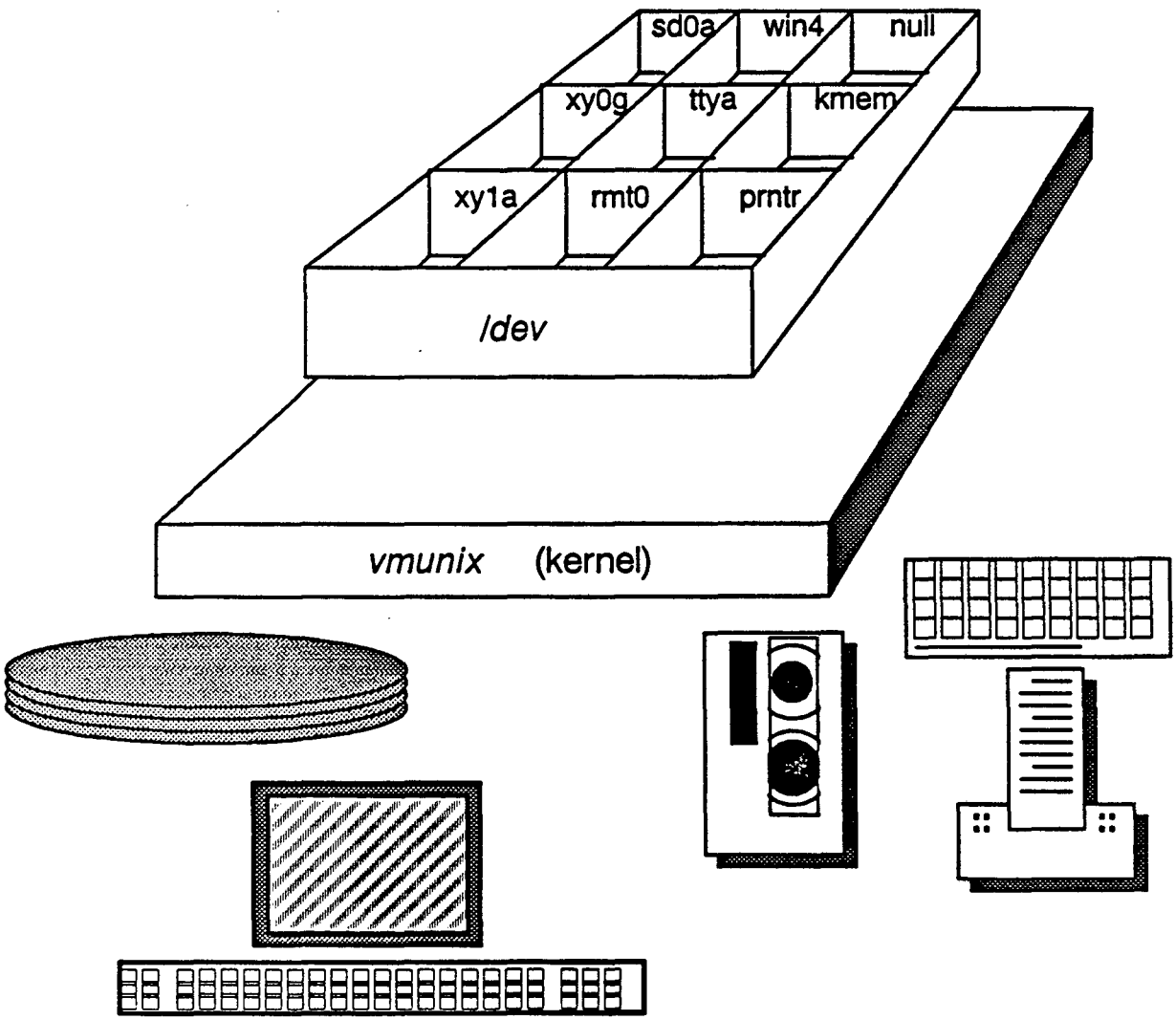


/usr	*General Purpose*
/dev, /etc, /export	*System Administration*
/home	*User directory*
/var	*Variable Length files*
/tmp	*Temporary Files*

The Filesystem

The filesystem is a virtual or logical division of a hard disk; it is a partition. SunOS uses a hierarchical filesystem. The *root* directory (also known as '/') is the top of the filesystem, or level 1. / contains additional directories and files. One important file is *vmunix*, the kernel.

All system I/O is channeled through the */dev* directory.



Files

- Compared to other operating systems, files are very simple.
- No implied structure is enforced by the system.
- Files are generic objects used to contain data, source or executable code, directories, garbage, etc.
- No overhead data, header, trailer, label or EOF character is stored with the file's data region. (The EOF character is a service provided to application by the kernel.)
- Files, (or directories), are accessed by one of two names:

absolute	*/full/path/name/from/root*
relative	*path/from/current/directory* *../path/from/parent* *~/path/from/home/directory* *./path/from/current/directory*

Files

A file is a nonstructured array of data of known location and size.

A file of *zero* length takes up zero bytes of storage.

Another way to think of an absolute pathname is as a "full" pathname. For instance, if your current directory is */home/sunray/student1/budget* and you wanted to use an absolute pathname to move to the directory *memos,* the absolute pathname would be */home/sunray/student1/memos.*

A relative pathname is a "partial" pathname; it's a pathname in relation to your current directory. Again, if the current working directory is */home/sunray/student1/budget*, then the relative pathname for *memos* is *../memos.* The '..' stands for the directory one level above your current directory in the hierarchy; in this case '..' stands for */home/sunray/student1.*
The '.' stands for the current directory and may be used in a pathname or by itself. '~' stands for your login id's home directory while '~*name*' references the home directory of the login id *name.*

Logging In and Out

- SunOS is case–sensitive; do NOT login with uppercase letters.
- SunOS will not "echo" passwords to the screen.
- There are three ways to logout: <CTRL–d>, the *logout* command and *exit* command. *exit* is normally used for terminating a subshell.
- Passwords are installed and changed with either the *passwd* or *yppasswd* commands.

Logging In and Out

patience login: *student1*
password:
Last login: Thu Jul 17 20:35:43 on ttydc
Sun UNIX 4.2 Release 4.0 (patience): Fri Jan 20 11:57:02 PDT 1988
You have mail.

patience% **^D**
patience login:

patience% **logout**
patience login:

Passwords are installed/changed using one of the commands: *passwd* or *yppasswd*. Ask your system administrator which command is in use on your system.

patience% **yppasswd**
Changing yp password for student1
Old yp password: *(password does not appear)*
New password: *(password does not appear)*
Retype new password: *(password does not appear)*
yellow pages passwd changed on sunburst

"Yellow Pages", listed in the example above, is a feature of SunOS which makes system administration easier to manage over a network. The yp allows certain system information from each workstation (such as all valid login ids and passwords), to be stored in one central place and accessed by all machines in the network. Because the information is shared in a central place, each workstation does not need its own copy.

UNIX/SunOS 4.0 Documentation

- The traditional UNIX documentation comes in three volumes.
- Sun has expanded this documentation to make it much more comprehensive.
- Most installations also have the on-line version of the manual available via the *man* command.

```
patience% man yppasswd
YPPASSWD(1)        USER COMMANDS        YPPASSWD(1)

NAME
        yppasswd – change login password in yellow pages

SYNOPSIS
        yppasswd [ name ]

DESCRIPTION
        Yppasswd changes (or installs) a network password
        associated
--More--(10%)
```

UNIX/SunOS 4.0 Documentation

Miniboxes:

Beginner's Guide set	(p/n SX-9A)
System Administration Guide set	(p/n SX-9B)
System Reference Manual Guide set	(p/n SX-9C)
Programmer's Reference Guide set	(p/n SX-9D)
SPARC Reference Manual set	(p/n SS-9A)

Beginner's Guides:

Sun System Overview	(p/n 800-1702)
Getting Started with SunOS	(p/n 800-1703)
Setting Up Your SunOS Environment	(p/n 800-1704)
Self Help with Problems	(p/n 800-1705)
SunView1	(p/n 800-1706)
Mail and Mail Messages	(p/n 800-1709)
Doing More with UNIX	(p/n 800-1710)
Using the Network	(p/n 800-1711)

SunOS 4.0 Operating System Manuals (partial list):

SunOS Reference Manual (p/n 800-1751)

- Section 1 = User Commands
- Section 2 = System Calls
- Section 3 = Library Routines
- Section 4 = Special Files (Device Driver Nodes)
- Section 5 = File Formats
- Section 6 = Games and Demos
- Section 7 = Public Information Files
- Section 8 = System Administration Commands

C Programmer's Guide	(p/n 800-1771)
Programming Debugging Tools	(p/n 800-1775)
Programming Utilities and Libraries	(p/n 800-1774)
Writing Device Drivers	(p/n 800-1780)
Editing Text Files	(p/n 800-1754)
SunView Programmer's Guide	(p/n 800-1783)
SunView 1 System Programmer's Guide	(p/n 800-1784)

Module 2
C Shell and SunOS Commands

Objectives: Upon completion of this module, the student should be able to:

- Construct shell command requests.
- Use the special shell characters: * ? [ ] & ; > >> < >& >! | && and ||.
- Use these directory commands: *cd, ls, mkdir, pwd* and *rmdir.*
- Use these file commands: *cat, cmp, diff, find, more, rm* and *sort.*
- Use these file and directory commands: *cp, mv* and *ln.*
- Use these commands to control permissions: *chgrp, chmod, chown.*
- Use the *su* command.
- Use the shell commands: *history* and *alias.*

Evaluation:

Perform Lab 1 to 90% proficiency.

Reference information:

- Appendix A, *Special Shell Characters.*
- *Getting Started with SunOS: Beginner's Guide* (p/n 800-1703-10), Chapters 4, 5, 9, and Appendix A.
- *Doing More with SunOS: Beginner's Guide* (p/n 800-1710-10), Chapters 2 - 4, and Appendix C.
- *SunOS Reference Manual* (p/n 800-1751-10).
- *Self Help with Problems: Beginner's Guide* (p/n 800-1705-10), Chapters 3 and 4.

The C Shell Interpreter

- The first token after the shell prompt must be the name of a valid command.

- A valid command is one that your shell can locate and has permission to execute.

- The shell uses your *search path* to look for commands. The search path lists pathnames of directories containing commands.

The C Shell Interpreter

patience% **ls**

Letters	df	f3	magazines
Memos	ef	f4	memofile
My.stuff	f1	f5	merger
Records	f10	f6	phone.list
af	f11	f7	yf
bf	f12	f8	zf
calendr	f13	f9	
cf	f2	ff	

patience% **/etc/vipw**
vipw: /etc/ptmp: Permission denied

patience% **fictional**
fictional: Command not found.

The *echo* command can be used to print your search path, which the C shell stores as a variable, *path*.

patience% **echo $path**
. /usr/bin /usr/ucb /bin

The dollar sign ($) tells the shell that the following word is a variable and to evaluate it as such.

Shell Wild Cards (Metacharacters)

- The asterisk matches zero or more characters in a file name.

- If the matched item is a directory when executing */s*, the contents of the directory are listed.

- The lone asterisk matches everything in the directory, except for entries whose names begin with a period.

Shell Wild Cards (Metacharacters)

```
patience% ls f*
f1      f11     f13     f3      f5      f7      f9
f10     f12     f2      f4      f6      f8      ff
patience% ls [mM]*
magazines  memofile   merger

Memos:
Memo.1          Memo.2          Memo.3          Memo.4

My.stuff:
a.file          nn.main.c       ptr.math2.c
file.empty      ptr.math1.c
patience% file *
Letters:            directory
Memos:              directory
My.stuff:           directory
Records:            directory
af:                 ascii text
bf:                 empty
calendr:            ascii text
cf:                 empty
df:                 empty
ef:                 empty
f1:                 ascii text
f10:                ascii text
.
.
.
```

Shell Wild Cards (Metacharacters)

- The question mark matches any one character in a file name.

- The square brackets allow you to match a given set or range of characters.

Shell Wild Cards (Metacharacters)

```
patience% ls f?
f1      f3      f5      f7      f9
f2      f4      f6      f8      ff

patience% ls f??
f10     f11     f12     f13

patience% ls -l f[3-8]
-rw-r--r--  1 student1      16 Jul 29 17:20 f3
-rw-r--r--  1 student1      16 Jul 29 17:20 f4
-rw-r--r--  1 student1      16 Jul 29 17:20 f5
-rw-r--r--  1 student1      16 Jul 29 17:20 f6
-rw-r--r--  1 student1      16 Jul 29 17:20 f7
-rw-r--r--  1 student1      16 Jul 29 17:20 f8

patience% ls f[2468]
f2      f4      f6      f8

patience% ls f[2-5]
f2      f3      f4      f5
```

Shell Directives

- The '&' runs a command in the background.

  A shell prompt is returned immediately and the command just issued continues asynchronously.

- The ';' is used to separate multiple commands on a single line.

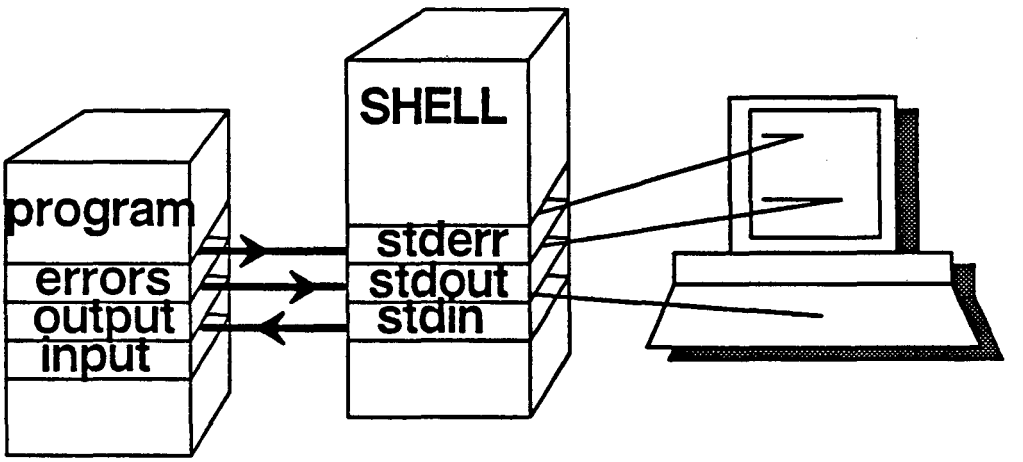
Shell Directives

```
patience% ls /usr &
[2] 1295
patience% 5bin          diag          kvm           pub
5include  dict          lib           sccs
5lib      etc           local         share
NeWS      games         lost+found    spool
adm       hosts         man           stand
bin       images.im1    mdec          sys
boot      images.im8    old           tmp
demo      include       ops           ucb

[2]    Done                    ls /usr
patience% ls; cal 6 1987
Letters         df              f3              magazines
Memos           ef              f4              memofile
My.stuff        f1              f5              merger
Records         f10             f6              phone.list
af              f11             f7              yf
bf              f12             f8              zf
calendr         f13             f9
cf              f2              ff
   June 1987
 S  M Tu  W Th  F  S
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30
patience%
```

Shell Input/Output Redirects and Pipes

- The shell works with three "files" which are normally connected to your terminal.

stdin	standard input, usually your keyboard
stdout	standard output, usually your screen
stderr	standard error, usually your screen



Shell Input/Output Redirects and Pipes

Using special shell operators, you can tell the shell to use other "*real*" disk files for *stdin*, *stdout* and *stderr*.

Shell Input/Output Redirects and Pipes

- The ">" character redirects *stdout* to a real file.
- The ">>" token appends *stdout* to a file.
- The "<" character redirects *stdin* from a real, disk–stored file.
- The "|" character redirects the *stdout* of one command into the *stdin* of another (*pipe*).

- The ">&" redirects *stdout* and *stderr*.
- The "|&" pipes *stdout* and *stderr*.
- *(command > file) >& errorfile* redirects *stdout* and *stderr* to separate files.

Shell Input/Output Redirects and Pipes

```
patience% date > date.file.1
patience% cat date.file.1
Tue Jan 31 17:38:55 PDT 1989
```

If the file does not exist, the shell creates it. If the file exists, it is cleared before writing. Toggling the special shell variable "noclobber" to "set" prevents file destruction due to redirection.

```
patience% set noclobber
patience% date > date.file.1
date.file.1: File exists.
patience% date >! date.file.1
patience% unset noclobber
patience% date > date.file.1
```

```
patience% date >> date.file.1
patience% cat date.file.1
Tue Jan 31 17:39:20 PDT 1989
Tue Jan 31 17:40:42 PDT 1989
```

```
patience% mail  student2@sunburst <  date.file.1
```

```
patience% date | mail student2@sunburst
```

```
patience% date abc
usage: date [-u] [yymmddhhmm[.ss]] [-a sss.fff]
patience% date abc >& date.file.2
patience% cat date.file.2
usage: date [-u] [yymmddhhmm[.ss]] [-a sss.fff]
```

Conditional Execution Operators

- Each command returns an exit status which may be evaluated prior to executing the next command.

- The '&&' indicates that the next command is only to be executed IF the first command is executed successfully.

- The '||' indicates that the next command is only to be executed IF the first command fails.

Conditional Execution Operators

```
patience% date && pwd
Tue Jan 31 18:25:53 PST 1989
/home/sunray/student1

patience% fictional && date
fictional: Command not found.
patience%

patience% date || pwd
Tue Jan 31 18:27:57 PST 1989

patience% fictional || date
fictional: Command not found.
Tue Jan 31 18:28:02 PST 1989
patience%
```

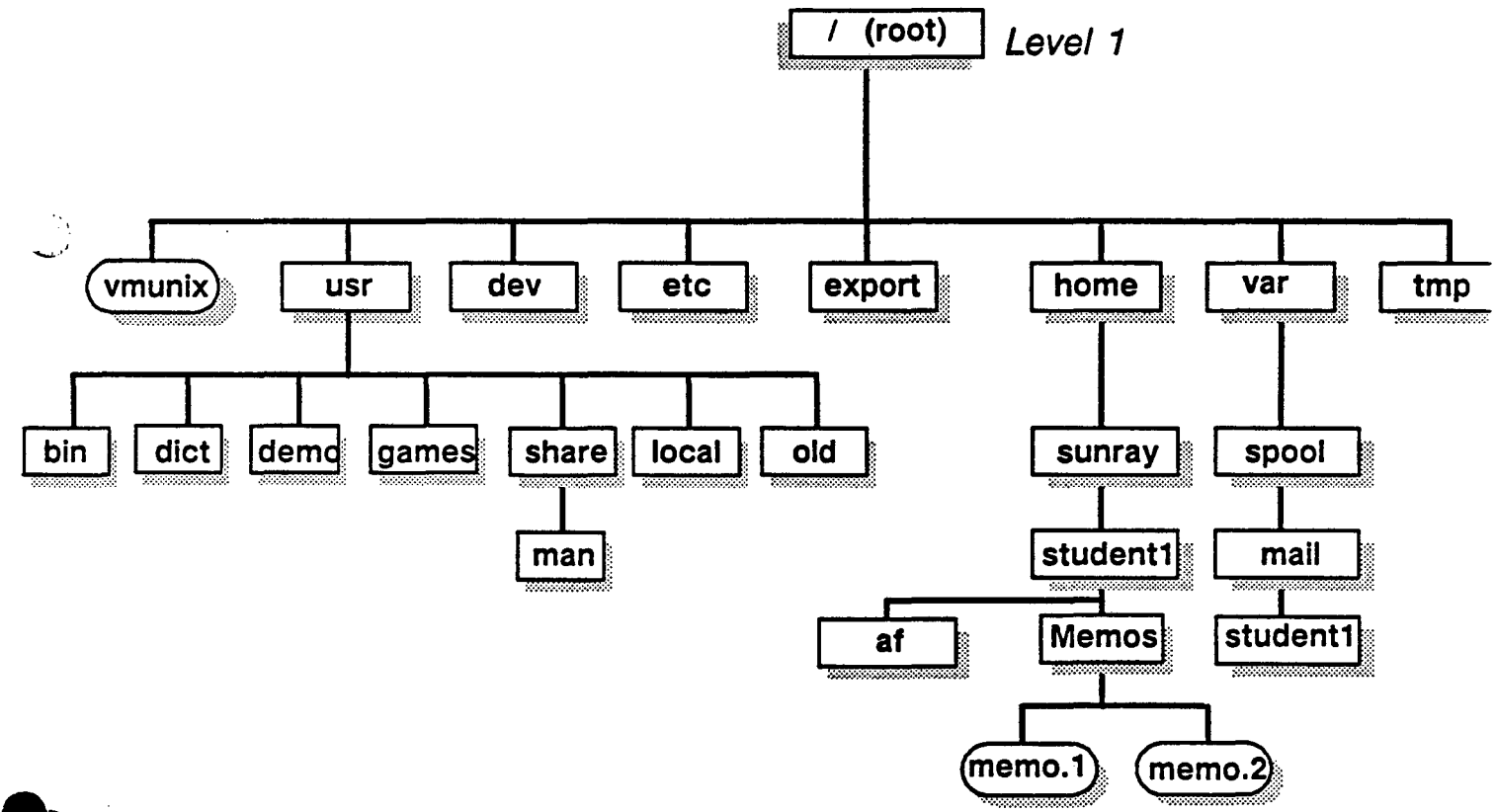
This concludes the introduction to the C shell command interpreter. Now let's look at some commonly used SunOS commands.

Useful SunOS Commands

man	show manual page for command (for more info, execute *man man* and then *man intro*)
cd	change default directory
pwd	print working (default) directory
ls	list directory contents
find	search directory hierarchy for files
file	determine file type
cat	display and/or concatenate files
more	display files, one screen at a time
diff	display differences between two files
cp	copy directory/file
mv	rename (move) directory/file
rm	remove (delete) file/directory
chmod	change access rights to file/directory
mkdir	create directory
rmdir	delete directory
echo	send a line of text to the screen
ps	process status (show jobs on the system)
lpr	print file
lpq	check printer queue
lprm	remove job from print queue

Useful SunOS Commands

Most of these commands are explained in the next few pages. This is by no means a complete list of SunOS commands; however, these commands are very useful to the new SunOS user. The examples in this section refer to the chart below, also shown on page 1-6.



/usr	*General Purpose*
/dev, /etc, /export	*System Administration*
/home	*User directory*
/var	*Variable Length files*
/tmp	*Temporary Files*

cd and *pwd*

- The *cd* command changes the current (working) directory.
- The format of the command is:

  cd [pathname]

- The tilde (~) symbol stands for your login directory.
- The "dot dot" (..) symbol stands for the parent directory.
- The *pwd* command prints the current (working) directory.
- The format of the command is:

  pwd

cd and *pwd*

Issuing the *cd* command with no pathname or with the '~' changes to the login directory, regardless of the current directory.

```
patience% cd
patience% pwd
/home/sunray/student1

patience% cd ~student2
patience% pwd
/home/sunray/student2

patience% cd ~
patience% pwd
/home/sunray/student1
```

To ascend one level in the directory hierarchy, use the *cd* command with the *absolute* pathname or with the symbol '..'.

```
patience% cd ..
patience% pwd
/home/sunray
patience% cd ..
/home
patience% cd ..
/
```

The tilde and dot dot may be used in pathnames.

```
patience% cd ~/Memos
patience% pwd
/home/sunray/student1/Memos

patience% cd ../Letters
patience% pwd
/home/sunray/student1/Letters
```

/s: List Directory Contents

- The command format is:

  /s [options] [pathname]

- If no pathname is specified, the current directory is listed.

- Commonly used options:

–a	list all files, including those that begin with '.'
–l	"long" directory format which includes file protection, owner, name, size and date.
–F	list with added flag characters, so that directories are marked with '/' and executables are marked with '*'.
–i	list each file preceded by its inode number. (An inode is a data structure that contains information which describes a file).

ls: List Directory Contents

```
patience% cd
patience% ls
Letters          date.file.1      f13              f9
Memos            date.file.2      f2               ff
My.stuff         df               f3               magazines
Records          ef               f4               memofile
af               f1               f5               merger
bf               f10              f6               phone.list
calendr          f11              f7               yf
cf               f12              f8               zf

patience% ls -a
.                cf               f13              ff
..               date.file.1      f2               magazines
Letters          date.file.2      f3               memofile
Memos            df               f4               merger
My.stuff         ef               f5               phone.list
Records          f1               f6               yf
af               f10              f7               zf
bf               f11              f8
calendr          f12              f9

patience% ls -l
total 32
drwxr-xr-x  2 student1     24  Mar 31 14:56 Letters
drwxr-xr-x  2 student1    512  Mar 31 15:09 Memos
drwxr-xr-x  2 student1    512  Mar 31 16:42 My.stuff
drwxr-xr-x  2 student1     24  Mar 31 14:56 Records
-rw-r--r--  1 student1     12  Mar 31 15:02 af
-rw-r--r--  1 student1      0  Mar 31 15:02 bf
-rw-r--r--  1 student1     17  Mar 31 15:03 calendr
-rw-r--r--  1 student1      0  Mar 31 15:02 cf
-rw-r--r--  1 student1     58  Mar 31 15:37 date.file.1
.......
patience% ls -F
Letters/         date.file.1      f1               f5
Memos/           date.file.2      f2               ff
My.stuff/        df               f3               magazines
Records/         ef               f4               memofile
.......
```

Options may be combined on the command line; for instance *ls -al* is valid.

mkdir **and** *rmdir*

- The *mkdir* command format is:

  mkdir dirname [dirname] [dirname] .....

- The *rmdir* command format is:

  rmdir dirname [dirname] [dirname] .....

- *rmdir* only removes empty directories.
  To remove non–empty directories, use *rm –r.*

mkdir and *rmdir*

The *–d* option for the *ls* command means that if the filename is a directory, do not list the contents, just the name.

```
patience% cd
patience% mkdir Testing
patience% mkdir Test.1 Test.2 Test.3
patience% ls -d T*
Test.1   Test.2   Test.3   Testing

patience% mkdir ~/Memos/Oldmemos
patience% ls ~/Memos
Memos:
Memo.1          Memo.3          Oldmemos
Memo.2          Memo.4

patience% cd
patience% rmdir Testing
patience% ls -d T*
Test.1  Test.2  Test.3
patience% rmdir ~/Memos/Oldmemos
patience% ls ~/Memos
Memo.1  Memo.2  Memo.3  Memo.4

patience% rmdir ~/Memos
rmdir: Memos: Directory not empty
patience% rm -r ~/Memos
patience% ls -d M*
My.stuff
```

File Commands: *cat* and *more*

- *cat* concatenates and displays files.
- The command format for *cat* is:

  cat [options] [filename] [filename].....

- Useful *cat* options include *–n,* which numbers the lines and *–v* which displays non–printing characters (except tabs and new–lines).

- *more* displays files, one screenful at a time.
- The command format for *more* is:

  more [options] [filename] [filename].....

- *more* commands:

<CR>	displays next line in file
<space>	displays next screen of lines
q or *Q*	quit
h	display *more* command summary
/	search
n	search again
b	back one page

File Commands: *cat* and *more*

```
patience% cd My.stuff
patience% cat nn.main.c
/*   nn.main.c       9.11.86         */
main()
{       int index, rsp;         rsp = fun(index);
        printf("%d\n", rsp);          rsp = fun(index);
        printf("%d\n", rsp);
}
patience%
```

```
patience% cat ~/f1 ~/f5  > f14
patience% cat f14
This is file 1.
This is file 5.
```

```
patience% more nn.main.c
/*   nn.main.c       9.11.86         */
main()
{       int index, rsp;         rsp = fun(index);
        printf("%d\n", rsp);          rsp = fun(index);
--More--(50%)
```

find

- *find* finds files in the directory hierarchy.

- The command format is:

  find pathname–list expression

- *pathname–list* is one or more pathnames (separated by spaces) in which *find* will start looking for files.

- *expression* may be one or more of:

–name filename	true if filename matches *filename.*
–user uname	true if file belongs to *uname.*
–inum n	true if file has inode number *n.*
–print	always true; the current pathname is printed.

find

patience% **find /usr -name words -print**
/usr/dict/words

patience% **cd**
patience% **find . -name file.empty -print**
./My.stuff/file.empty

patience% **find . -name '*empty*' -print**
./My.stuff/file.empty

patience% **find . -inum 178 -print**
./zero

An *inode* is an data structure which contains information that describes a file. It contains such information as the access rights on the file and pointers to the data blocks on the disk where the file is stored.

cmp **and** *diff*

- *cmp* compares two files byte by byte until the files end or a difference is found.
- The command format is:

  cmp [options] file1 file2

- *diff* shows the differences between the contents of two files or directories.
- The command format is:

  diff [options] file_or_dir1 file_or_dir2

cmp and *diff*

```
patience%  cd ~/My.stuff
patience% cmp ptr.math1.c ptr.math2.c
ptr.math1.c ptr.math2.c differ: char 168, line 9

patience% diff ptr.math1.c ptr.math2.c
9c9
<                 putchar( *(ptr++) );
---
>                 putchar( *ptr++);
```

Both *cmp* and *diff* also work on binary files. In the examples below, *ptr.math1.c* and *ptr.math2.c* have been compiled. The resulting files are *ptr.math1* and *ptr.math2.*

```
patience% cmp ptr.math1 ptr.math2
ptr.math1, ptr.math2 differ: char 17504, line 34
patience% diff ptr.math1 ptr.math2
Binary files ptr.math1 and ptr.math2 differ
```

sort

- *sort* sorts or merges files.
- The command format is:

  sort [options] filename [filename].....

- Useful options:

–r	reverse order sort
–n	sort by arithmetic value of numeric string
+x	*x* denotes the number of fields to skip from the beginning of the line before starting the sort.
–o filename	store sorted output in *filename*
–t'ch'	names 'ch' as a field separating character

sort

```
patience% sort a.file
demos   1589    mayfield
doc     1409    cutter
doc     1409    cutter
eirc    1558    toner

patience% sort -r a.file
eirc    1558    toner
doc     1409    cutter
doc     1409    cutter
demos   1589    mayfield

patience% sort +1 a.file
doc     1409    cutter
doc     1409    cutter
eirc    1558    toner
demos   1589    mayfield

patience% ls -l *.c | sort -rn +3
-rw-r--r--  1 student1      299 Sep 11 14:38 ptr.math1.c
-rw-r--r--  1 student1      297 Sep 11 14:38 ptr.math2.c
-rw-r--r--  1 student1      187 Sep 15 17:58 nn.main.c

patience% sort -t: +2n /etc/passwd
nobody:*:-2:-2::/:
+::0:0:::
+student1::0:0::/home/sunray/student1:
root:AJxDEWOEfRMzQ:0:1:Operator:/:/bin/csh
sysdiag:*:0:1:System Diagnostic:/usr/diag/sysdiag:/usr/diag/sysdiag/sysdiag
daemon:*:1:1::/:
. . .
```

Directory and File Commands: *cp* and *mv*

- *cp* copies files and directories. The source file/directory remains unchanged.
- The command format is:

  cp [options] file1 file2

- *mv* moves (renames) files and directories. The source file/directory is deleted.
- The command format is:

  mv [options] file1 file2

Directory and File Commands: *cp* and *mv*

```
patience% cp nn.main.c temp

patience% ls -l nn.main.c temp
-rw-r--r--  1 student1      187 Sep 15 17:58 nn.main.c
-rw-r--r--  1 student1      136 Feb 1 16:32 temp

patience% cp -r ~/Letters ~/Records/Letters.backup
patience% ls ~/Records
Letters.backup

patience% mv temp temp2
patience% ls -l nn.main.c temp temp2
temp not found
-rw-r--r--  1 student1      187 Sep 15 17:58 nn.main.c
-rw-r--r--  1 student1      136 Feb 1 16:32   temp2
```

If the last item in a list of arguments is a directory, both *cp* and *mv* willl locate the new files in that directory. For example:

```
patience% cd
patience% mv f1 f2 f3 f4  calendar  My.stuff
patience% ls My*
a.file          f14             f4              ptr.math1.c
calendr         f2              file.empty      ptr.math2.c
f1              f3              nn.main.c       temp2
```

The *ln* Command

- A link is a directory entry that references another file or directory.

- Two types of links:
  - hard (default)
  - symbolic

- Command format:

ln [–fs] filename [linkname]
pathname... directory

The *ln* Command

A file or directory may have any number of links (references) to it. A hard link creates a standard directory entry with the same inode number as the file it is being linked to, which means that hard links can only be made to existing files (not directories). Execute the *ls –li* command to compare the inode numbers. In addition, hard links cannot span file systems.

Symbolic links also create a directory entry but actually point to files and/or directories. However, it is not necessary for the file or directory being linked to to exist prior to executing the *ln* command. The file or directory may even be removed after the symbolic link is created, but will not affect the link itself. Trying to access a nonexistent file or directory will cause an error message. Execute *ls –l* to view information about the link. In contrast to hard links, symbolic links may span file systems.

The *–f* option is only available to the superuser and tells the system to force a hard link to a directory. The *–s* option tells the system that you are creating a symbolic link.

Executing *ln*

```
patience% cd /tmp

patience% man clear > clear.mp

patience% ls -li clear.mp
  96 -rw-r--r--  1 student1        494 Feb 15 13:18 clear.mp

patience% ln clear.mp clear.hlink

patience% ls -li clear*
  96 -rw-r--r--  2 student1        494 Feb 15 13:18 clear.hlink
  96 -rw-r--r--  2 student1        494 Feb 15 13:18 clear.mp

patience% cd /other_filesystem

patience% ln /tmp/clear.mp clear.link
clear.link: Cross-device link

patience% ln -s /tmp/clear.mp clear.slink
patience% ls -li
18530 lrwxrwxrwx  1 student1         13 Feb 15 13:24 clear.slink -> /tmp/clear

patience%
```

Executing *ln*

One of the arguments which UNIX passes to a newly created process is the name of the calling command. Therefore, we can do the following:

```
main(argc, argv)
char  *argv[];
{
      if (!strcmp(argv[0], "cmd1"))
         set flags. . .;
      else
         set other_flags;
      run_app(flags);
}


run_app(op_params)
int op_params;
{
      :
      :
}
```

In this manner we can have one executable acting like many different (but related) utilities.

su: Switching User Id

- *su* allows the user to temporarily switch his/her effective user id.
- The command format is:

  su [–] [username]

  If no *username* is entered, 'root' (superuser) is assumed.

- 'root' has special privileges. Certain commands can only be run by superuser.

su: Switching User Id

patience% **su**
Password:
patience#

To return to previous user id, enter *logout* or <CTRL-d>.

patience# **CTRL-d**
patience%

The dash ("-") option performs a complete login such that all variables are removed from the environment except for TERM, USER, and SHELL. The shell will also read the username's *.login* or *.profile* file.

Permissions: *chmod*

- Permissions are the access rights to files and directories.

- When a file or directory is created, it has permissions assigned to it based on the current value of 'umask'. To change the permissions, use the *chmod* command.

- The command format is:

  chmod mode file_or_dir [file_or_dir]....

- *mode* may be specified in absolute notation:

  400 read by owner only
  200 write by owner only
  100 execute (search in directory) by owner only
  The second digit indicates group access and the third, access by all others (040 is group read access).

- *mode* may be specified in symbolic notation:

  u = user, g = group, o = others, a = all (ugo)
  + = add permission, – = remove permission
  r = read, w = write, x = execute

Permissions: *chmod*

```
patience% cd
patience% ls -l
total 29
drwxr-xr-x  2 student1          24  Mar 31 14:56 Letters
drwxr-xr-x  2 student1         512 Mar 31 16:42 My.stuff
drwxr-xr-x  2 student1          24  Mar 31 14:56 Records
drwxr-xr-x  2 student1          24  Mar 31 16:02 Test.1
drwxr-xr-x  2 student1          24  Mar 31 16:02 Test.2
drwxr-xr-x  2 student1          24  Mar 31 16:02 Test.3
-rw-r--r--  1 student1          12  Mar 31 15:02 af
.....

patience% chmod 777 af
patience% ls -l af
-rwxrwxrwx  1 student1          12  Mar 31 15:02 af

patience% chmod go-wx af
patience% ls -l af
-rwxr--r--  1 student1          12  Mar 31 15:02 af
```

The *-R* option changes permissions recursively, that is, throughout the directory hierarchy.

```
patience% chmod -R 777 *
patience% ls -l
total 29
drwxrwxrwx  2 student1          24  Mar 31 14:56 Letters
drwxrwxrwx  2 student1         512 Mar 31 16:42 My.stuff
drwxrwxrwx  2 student1          24  Mar 31 14:56 Records
drwxrwxrwx  2 student1          24  Mar 31 16:02 Test.1
drwxrwxrwx  2 student1          24  Mar 31 16:02 Test.2
-rwxrwxrwx  2 student1          24  Mar 31 16:02 Test.3
-rwxrwxrwx  1 student1          12  Mar 31 15:02 af
.....
patience% ls -l ~/Records
-rwxrwxrwx  1 student1          12  Mar 31 17:11 Letters.backup
```

Permissions: *chown* and *chgrp*

- *chown* changes the owner of files.
- The command format is:

  /etc/chown [–f] [–R] owner [.group] file [file].....

- *–R* recursively changes ownership.
- Only super–user can execute *chown*.

- *chgrp* changes the group–id of files.
- The command format is:

  chgrp [–f] [–R] group file [file].....

- To execute *chgrp* successfully you must belong to the specified group and be the owner of the file, or be super–user.

Permissions: *chown* and *chgrp*

```
patience% su
Password:
patience# cd /home/sunray/student1
patience# /etc/chown student2 af
patience# ls -l af
-rwxrwxrwx  1 student2          12  Mar 31 15:02 af
patience# CTRL-d
patience%
```

The *-g* option of the *ls* command lists group information.

```
patience% ls -lg af
-rwxrwxrwx  1 student2  educatio12  Mar 31 15:02 af
```

```
patience% chgrp staff af
chgrp: You are not a member of the staff group
patience% groups
education students source
patience% chgrp source af
patience% ls -lg af
-rwxrwxrwx  1 student2  source  12  Mar 31 15:02 af
```

The Shell *history* Command

- The C Shell has the ability to remember commands and make them available for reuse or modification and reuse.

- The option is enabled by the command:

  patience% **set history = 23**

- *history* uses the operators:

  !
  ^
  :

The Shell *history* Command

In the example, *set history = 23*, the number specifies the number of lines you wish to save in the history list, in this case 23. If you do not set *history*, the C Shell will remember only the most recent command.

Executing *history*

- Command format:

  history [–hr] [n]

Executing *history*

The *–r* option prints the history list in reverse order; with most recent command first. The *–h* option removes the history number when displaying the history list. If you do not wish to view the entire history list, you may specify the number of lines you wish to see with the command: *history n,* where *n* is the number of lines.

```
patience% history
  98  set history = 23
  99  pwd
 100  ls
 101  date
 102  mv f1 f2
 103  ls
 104  cat f2
 105  chmod 777 f2
 106  history

patience% history 4
 104  cat f2
 105  chmod 777 f2
 106  history
 107  history 4
```

Accessing a Past Event

- *!!* executes the previous command.
- *!* followed by a number or a pattern re–executes the most recent past event corresponding to the specified history number or pattern.
- *!?* followed by a pattern will re–execute the most recent command containing the pattern.

Accessing a Past Event

```
patience%!!
history 4
 105  chmod 777 f2
 106  history
 107  history 4
 108  history 4

patience% !99
pwd
/home/sunray/student1


patience% !da
Mon Feb 13 14:53:48 PST 1989


patience% !?f2
chmod 777 f2
patience%
```

Accessing a Past Event (con't.)

- The ^ operator is the delimeter used when making simple corrections to the previous command.

- *!$* substitutes the last argument of the previous command.

- *!m:s/old/new* allows you to recall a previous command and make a change to it.

Accessing a Past Event (con't.)

```
patience% pwt
pwt:  Command not found.
patience% ^t^d
pwd
/home/sunray/student1
patience%
```

```
patience%  ls My.stuff
a.file          f14          f4           ptr.math1.c
calendr         f2           file.empty   ptr.math2.c
f1              f3           nn.main.c    temp2

patience% cd !$
cd My.stuff
patience%  cat f14
This is file 1.
This is file 5.

patience% cp !$ new.file
cp f14 new.file
patience%

patience% !105:s/f2/new.file
chmod 777 new.file
patience%
```

Aliases

- Available only in the C shell.
- Aliases allow users to customize their command set by adding new commands.
- Aliases are used to:
  - Assign simple names to frequently used commands.
  - Simplify a long and/or complex command line.
- Aliases are created via the *alias* command and are removed with the *unalias* command.
- Command formats:

  alias [name [def]]
  unalias pattern

- If the *def* (the command(s) to be executed when *name* is entered as a command by itself) contains shell directives or metacharacters, then *def* must be enclosed in quotes.

Aliases

```
patience% alias h history
patience% h
  98  set history = 23
  99  pwd
 100  ls
 101  date
 102  mv f1 f2
 103  ls
 104  cat f2
 105  chmod 777 f2
 106  history
 107  history
 108  pwd
 109  date
 110  chmod 777 f2
 111  pwt
 112  pwd
 113  ls My.stuff
 114  cd My.stuff
 115  cat f14
 116  cp f14 new.file
 117  chmod 777 new.file
 118  alias h history
 119  h
patience%
```

```
patience% alias  stuff   'cd ~/My.stuff; ls -l'
patience% alias  cl      'clear; pwd'
```

Summary of Directory and File Commands

Directory commands:

cd	changes current directory
ls	lists directory contents
mkdir	creates new directory
pwd	print working (current) directory
rmdir	deletes empty directory

File commands:

cat	displays and concatenates files
cmp	compares files (character by character)
diff	prints differences between files
find	searches directory hierarchy for files
more	displays files, one screen at a time
rm	removes file
sort	sorts files
lpq	check printer queue
lprm	remove job from print queue

Directory and file commands:

chgrp	changes group owner of file/directory
chmod	changes access rights to file/directory
chown	changes owner of file/directory
cp	copies directory/file
mv	moves directory/file to new location
ln	creates links to files and/or directories

Miscellaneous:

man	show manual page for command (for more info, execute *man man* and then *man intro*)
echo	send a line of text to the screen
ps	process status (show jobs on the system)
lpr	print file

Summary of Directory and File Commands

Module 3
SunView

Objectives: Upon completion of this module, the student should be able to:

- Demonstrate how to start and stop SunView.
- Demonstrate how to use the basic window tools.
- Design and implement a custom window setup.
- Use *textedit* to create and change files.

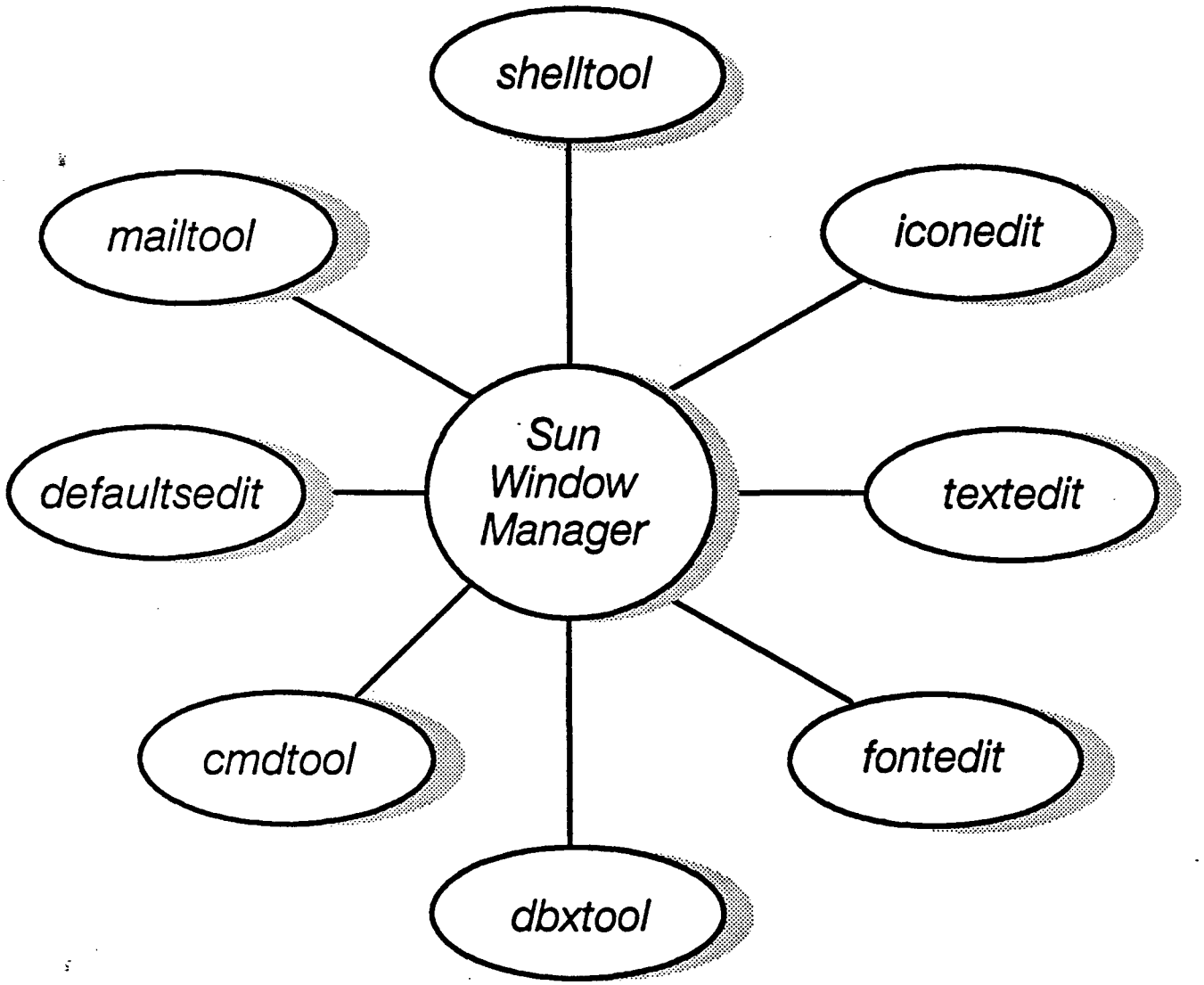
Evaluation:

Perform Lab 2 to 90% proficiency.

Reference information:

- Appendix B, *vi Quick Reference*
- *SunView 1 Beginner's Guide* (p/n 800–1706–10).
- *Mail and Messages: Beginner's Guide* (p/n 800–1709–10), Chapters 1, 2, and 3.
- *Getting Started with SunOS: Beginner's Guide* (p/n 800–1703–10), Chapter 6.
- *Self Help with Problems: Beginner's Guide* (p/n 800–1705–10), Chapter 5.

SunView


shelltool
mailtool
iconedit
Sun
Window
Manager
defaultsedit
textedit
cmdtool
fontedit
dbxtool

SunView

Program	Function
shelltool	SunOS shell in a window
cmdtool	Shelltool and SunView Text facility
textedit	Window and mouse based text editing program
defaultsedit	Program to change SunView's window defaults
mailtool	Window and mouse based mail program
perfmeter	Meter displaying system performance statistics
lockscreen	Screen protection program
dbxtool	Window-based debugger
fontedit	Tool for creating and revising character fonts
iconedit	Tool for creating and revising icon and cursor images

Starting Up Windows *(sunview)*

- *sunview* — Activates the Sun Window System

- Options:

–i	Reverses video screen image to white–on–black.
–b r b g	Specifies the values of the red, green and blue components of the background color (color monitors only). Default is 255 (white).
–f r b g	Specifies the values of the red, green and blue components of the foreground color (color monitors only). Default is 0 (black).
–background	Specifies a raster file to run in the background.

Starting Up Windows *(sunview)*

patience% **sunview**

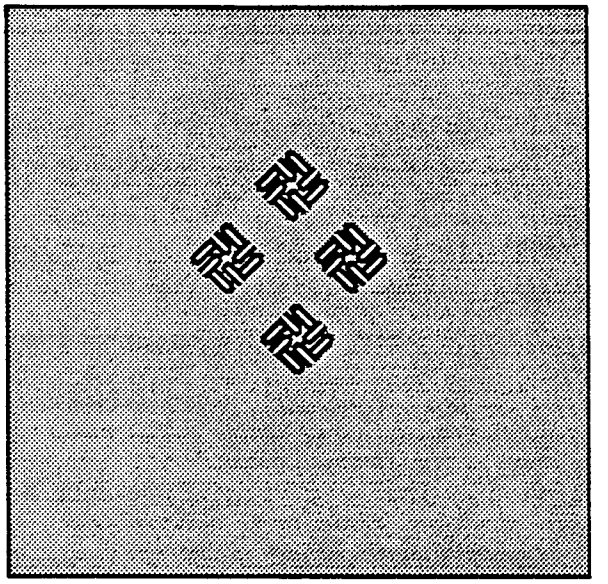
Note: In SunOS 3.5 and earlier, the command for starting SunView is *suntools* instead of *sunview.*

patience% **sunview -background /usr/images/threefounders**

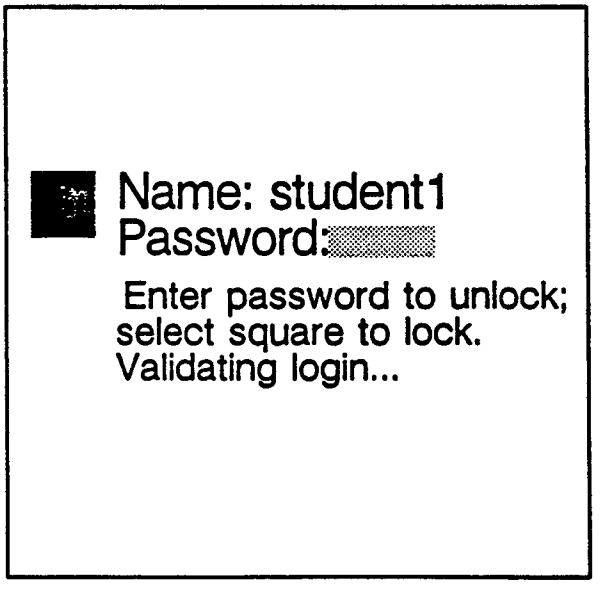
Each window is controlled by a separate shell (or other process). Mouse input is controlled by the sunview process.

Exiting SunView

- Lock Screen



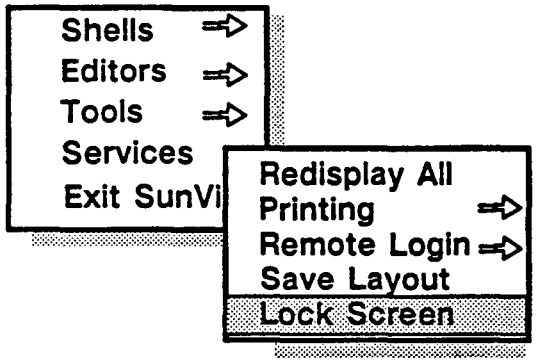
- Re–entering the system



- From the root menu select "Exit SunView" and confirm

Exiting SunView

The *Lock Screen* program allows a user to secure his/her workstation without logging out. By locking the screen, a user can leave processes running and prevent unauthorized workstation access.



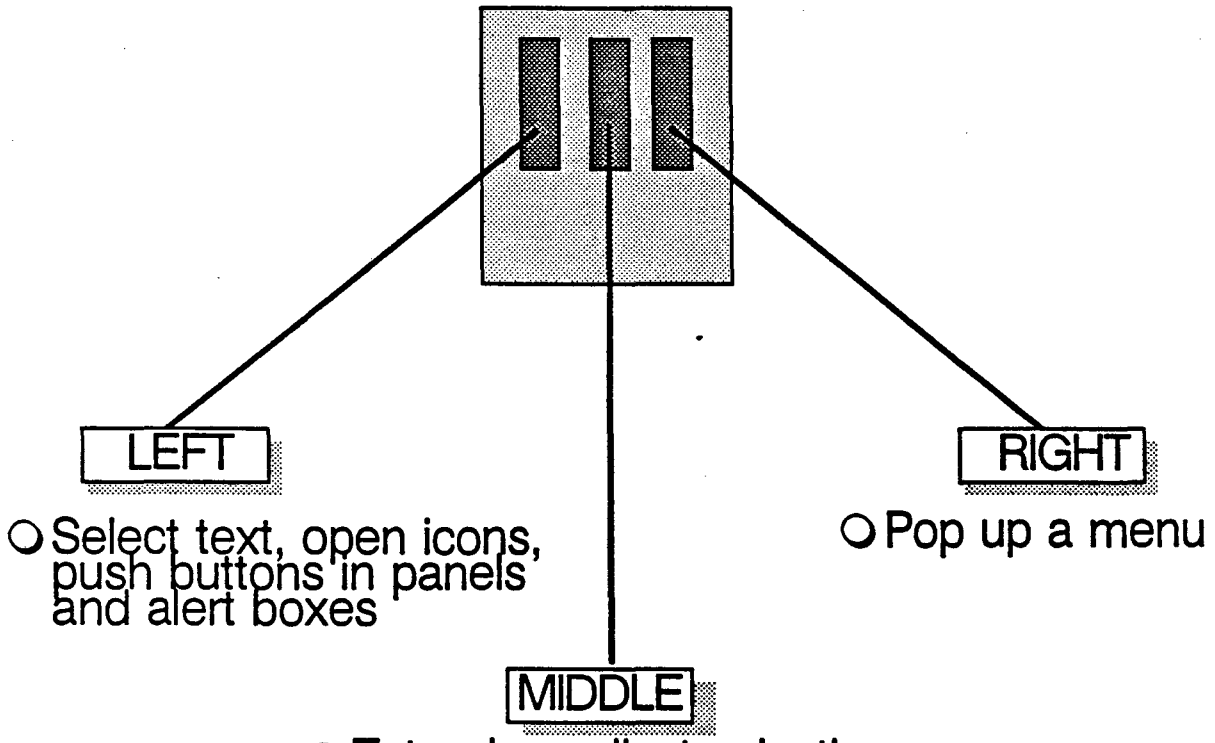
Depress the right mouse button and highlight the text *lockscreen* to activate the screen locking mechanism. Another method to lock the screen is to enter the *lockscreen* command from any *shelltool.*

Depress any mouse button *or* keyboard character when the *lock screen* image is displayed. This will bring up the screen which displays *login–id* and prompts for a *password.* Enter the password and enter a <CR>. If a *login–id* and *password* is not entered within several seconds, *the lockscreen* program automatically resumes. *Note:* Type a <CR> in the event a password does not exist for the user.

To cancel the login process and re–activate *lockscreen*, position the system pointer in the shaded square then depress the left mouse button.

Mouse Buttons

Mouse Actions: *click* or *press and hold*



Mouse Buttons

Mouse	Action
Text Selections:	
1 Left Click	Select a character
2 Left Clicks	Select a word
3 Left Clicks	Select a line
4 Left Clicks	Select everything in a window
Click left; Move; Click Middle	Select range of text
Window Control:	
Click Left on Icon	Open
Click Left on Window Frame	Expose
Hold Middle on Icon/Window Frame	Move
Control–Left Click on Window Frame	Zoom/Unzoom
Control–Hold Middle on Window Frame	Resize

Menus

- Editors Menu
- Tools Menu
- Services Menu
- Shells Menu

Menus

The 'Editors ⇨' *Pull-Right* Menu

This menu contains the names of tools for creating and revising documents (*Text Editor*), icons (*Icon Editor*), fonts (*Font Editor*) plus a tool that allow users to change SunView defaults (*Defaults Editor*).

The 'Tools ⇨' *Pull-Right Menu*

This menu contains the names of additional important applications which allow users to utilize electronic mail (*Mail Tool*), debug program code (*Dbx Tool*), monitor system performance (*Performance Meter*), and display the time and date (*Clock*).

The 'Services ⇨' *Pull-Right Menu*

This menu contains a list of useful options. *Redisplay All* refreshes the SunView screen; '*Printing* ⇨' displays the status of print jobs and prints a selection; '*Remote Login* ⇨' opens a new Shell or Command Tool and connects to a remote host; '*Save Layout* ⇨' saves the current SunView screen layout in the ~/.sunview file; '*Lock Screen*' displays a random Sun logo on a black background.

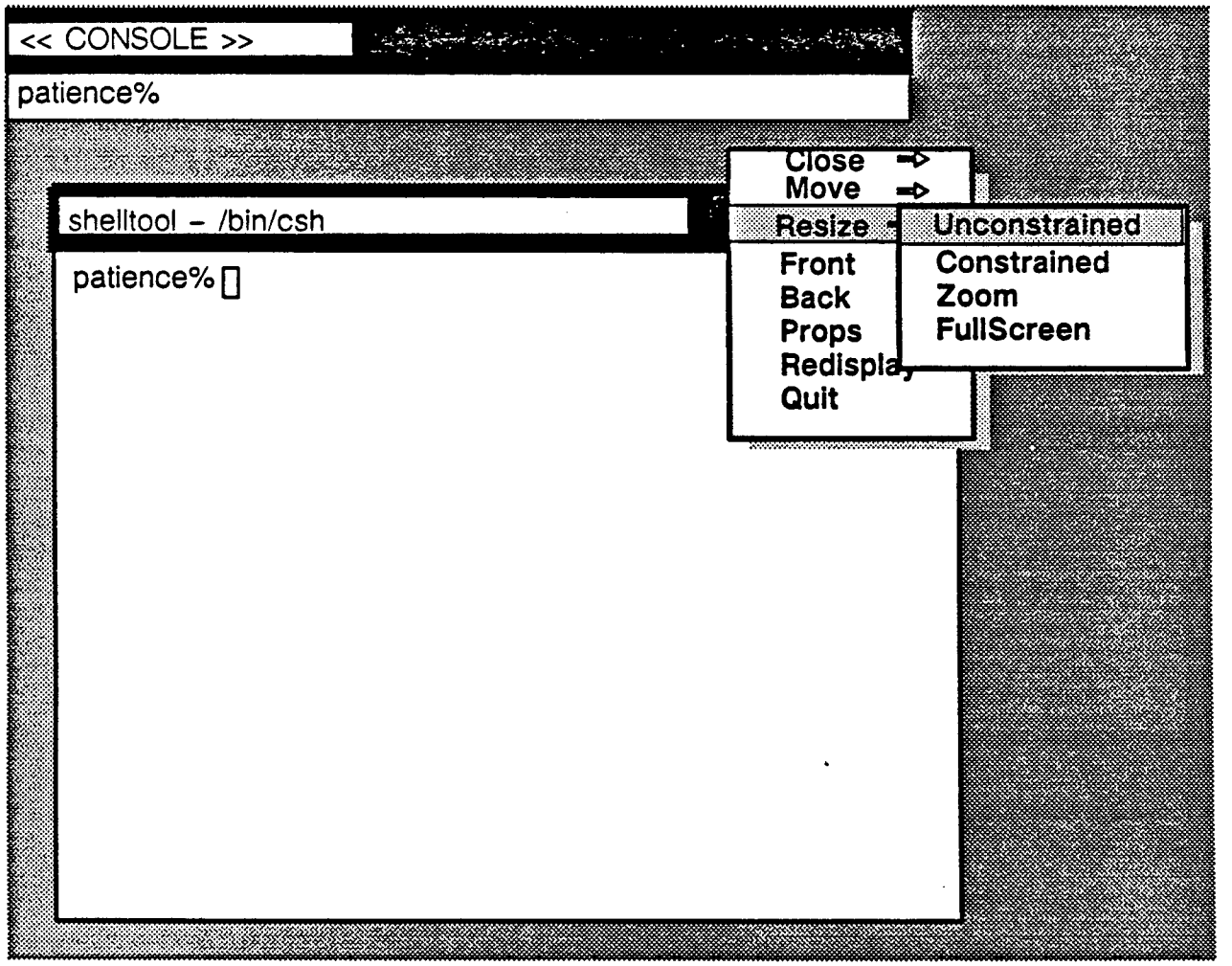
The 'Shells ⇨' *Pull-Right Menu*

This menu allows you to select which type of shell you wish to open: *Command Tool, Shell Tool, Graphics Tool,* or *Console.*

'Exit SunView' exits SunView.

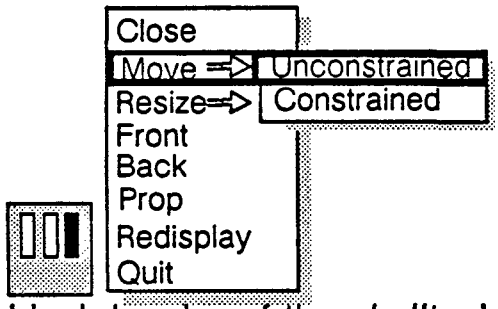
Frame Menu

- Controls characteristics of a given window, invoked with right mouse button on border.

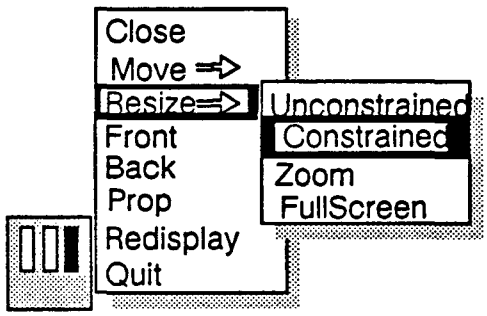


Frame Menu

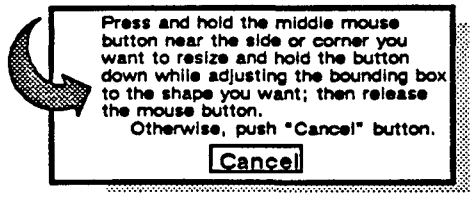
In the case of the *move* and *resize* operations, the frame menu allows the user to move or resize the window in either horizontal (constrained), vertical (constrained) or both directions (unconstrained).



Position the ↑ (system locator) within the black border of the *shelltool* window. Press, hold and highlight the *shelltool* entry *Move* with the right mouse button. Move the mouse to the right, following the direction of the arrow and then release it when the appropriate selection is highlighted. An *alert box* is displayed; position the ↑ as instructed and drag the window across the screen.



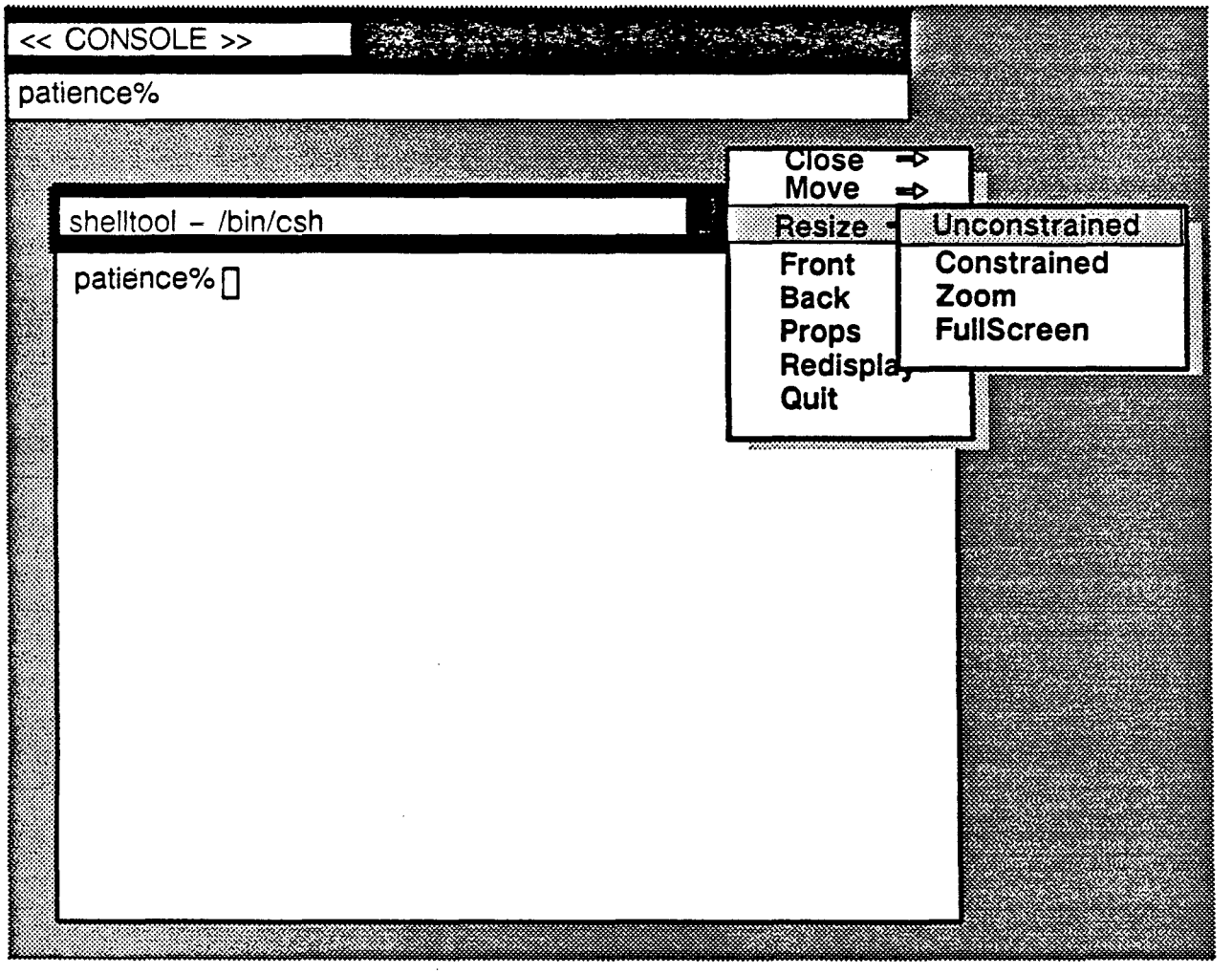
Click and highlight the *shelltool* entry *Resize* with the right mouse button. Move the mouse to the right following the direction of the arrow and then release it when the appropriate selection is highlighted. An alert box



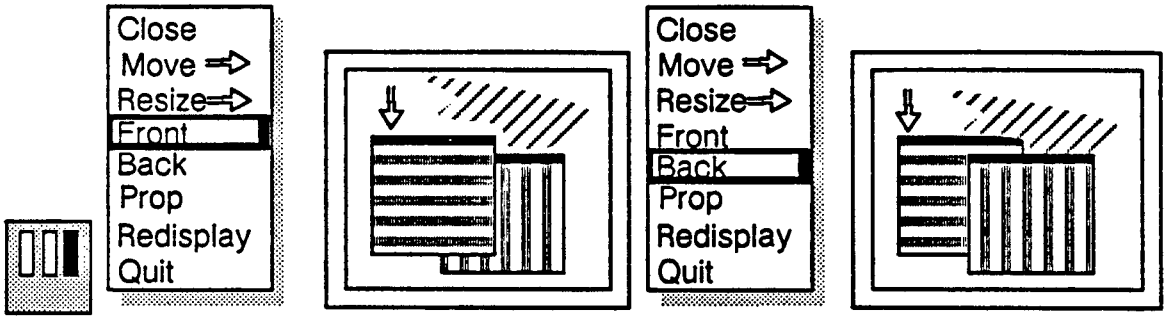
is displayed; position the ↑ as instructed using the middle mouse button and reshape the window accordingly. The *Zoom* and *FullScreen* pull-right menu options automatically readjust the size of the windows.

Frame Menu (con't.)

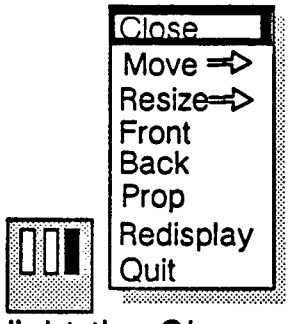
- Controls characteristics of a given window, invoked with right mouse button on border.



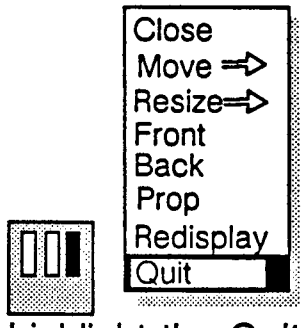
Frame Menu (con't.)



Selecting the *Front* option brings the window into the foreground (nearest to the user), thereby exposing the window. Selecting the *Back* option places the window into the background (farthest from the user), thereby hiding the window from view. Press, hold and highlight the *Front* or *Back* entry on the frame menu with the right mouse button and then release it. Another method to move windows from *Front* to *Back* is to click the *left* mouse button anywhere within the black border of the window.



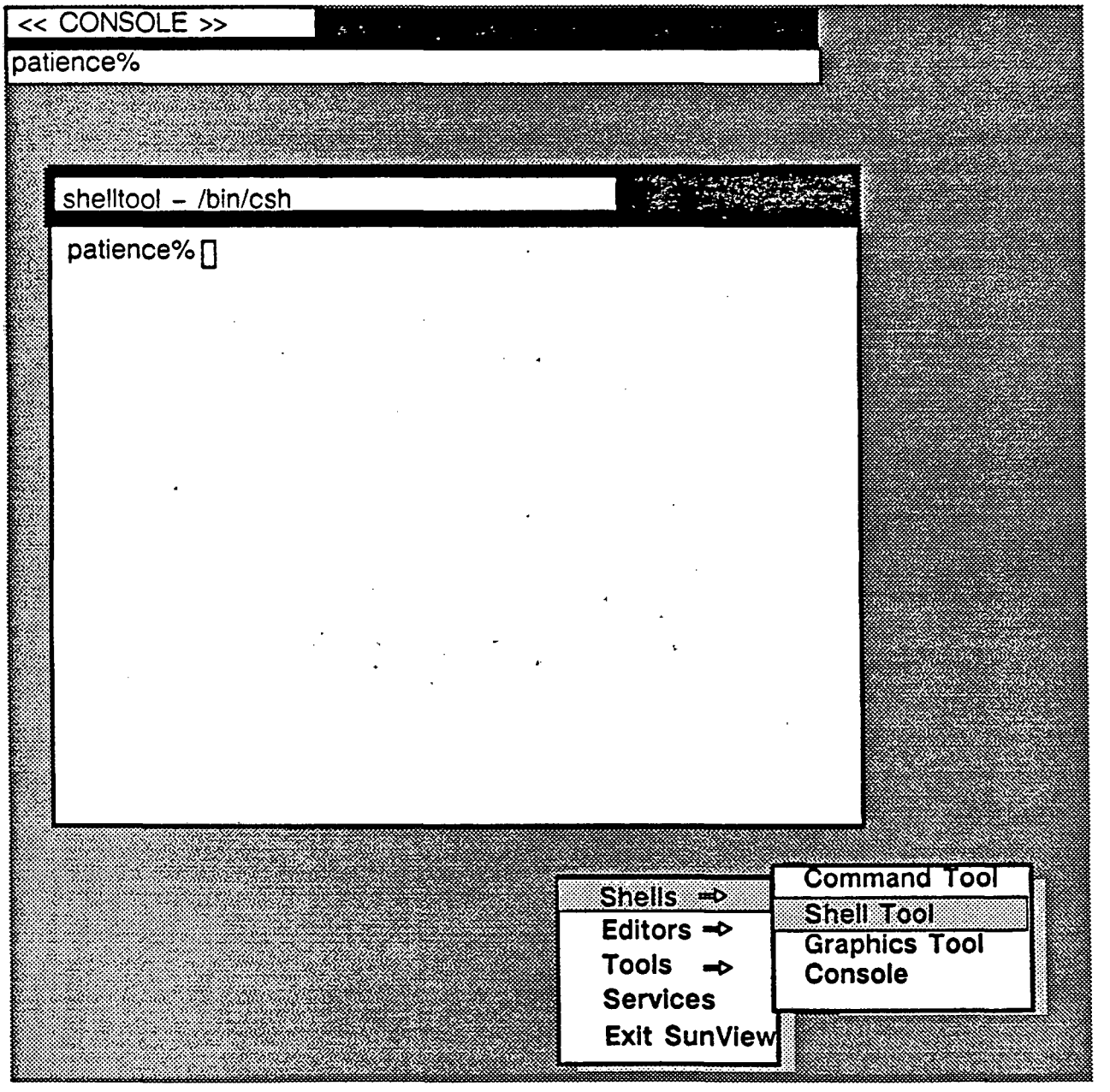
To temporarily close the window, press, hold and highlight the *Close* entry on the frame menu with the right mouse button, and release it. The closed window is represented by an icon which can be re-opened at a future time.



To permanently disable this *shelltool*, press, hold and highlight the *Quit* entry on the frame menu with the right mouse button, and release it. The window disappears from the SunView screen.

Shell Tool

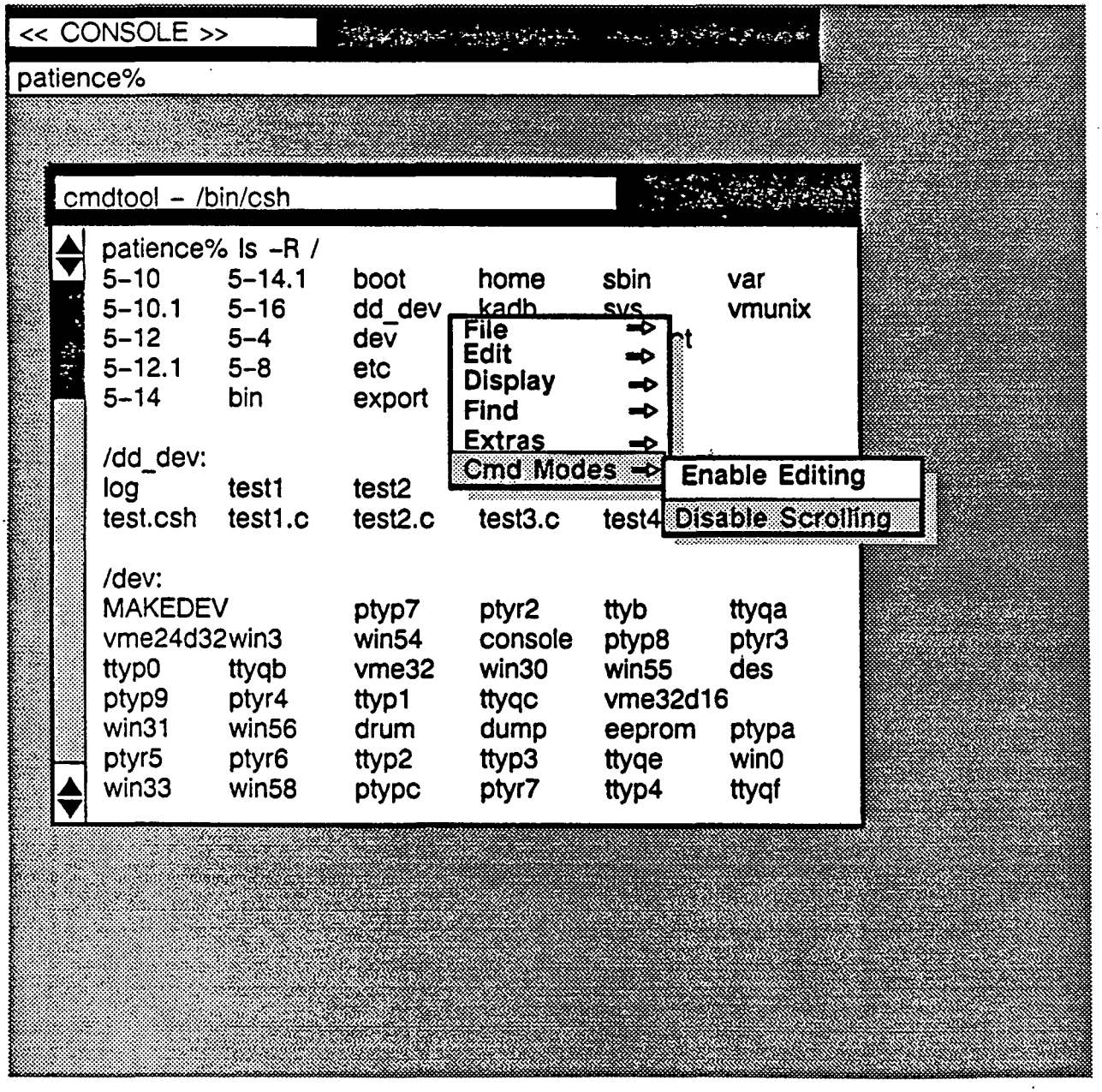
- A simple terminal emulator
- Each shell is treated as a separate login session (although there is no "logging into" a shelltool).



Shell Tool

Command Tool

- Saves display in a scrollbar–controlled log which allows simple editing and saving
- Used as the console window



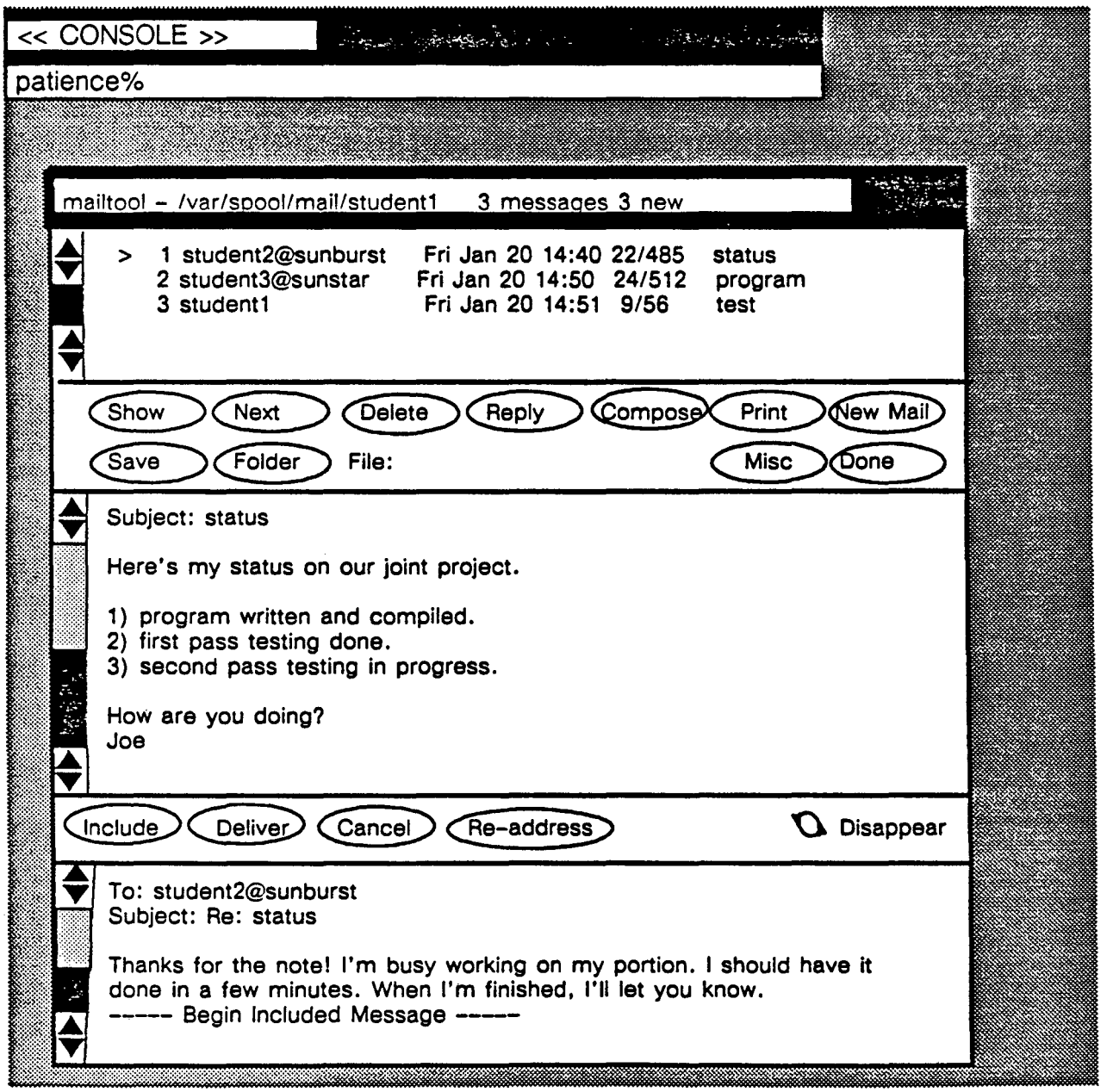
Command Tool

Light gray area = whole log
Dark gray area = portion currently visible
Select "File =>Empty document" to clear log

If scrolling is enabled, place mouse pointer in scroll bar to scroll. The pointer's appearance changes to '=>'. Using the left mouse button moves text up visually (move forward through display); using the right mouse button moves text down. Using the middle mouse button selects a relative position in the file, based on the pointer's relative vertical placement in the scroll bar.

Mail Tool

- A window interface to SunOS *mail* utility
- Various subwindows, each with its own scroll bar



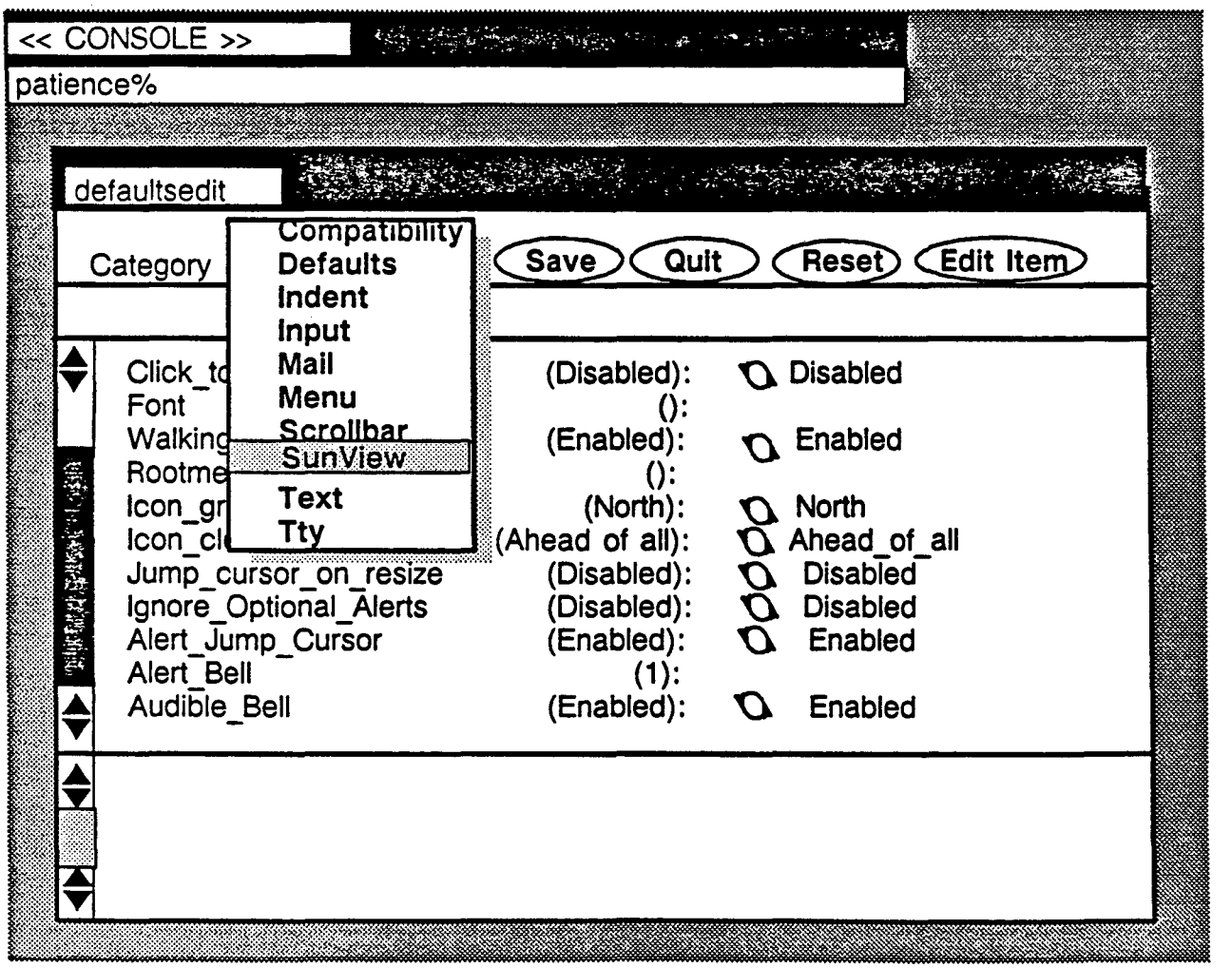
Mail Tool

- Subwindows:
  Header List Window: Contents of Mailbox
  Command Panel: Mail Operations
  Message Window: Display Messages
  Composition Window: Writing/Responding to Mail
- Respond to mail with "reply"
- Generate new messages with "compose"
- Lower screen splits, four new soft buttons appear, then enter your message in new lower portion
- "deliver" or "cancel" when ready

Defaults Editor

- A general purpose mouse driven default parameter editor currently setup for: Indent, Mail, and SunView as well as parameters for: Menus, Scrollbars, Text and Tty Subwindows

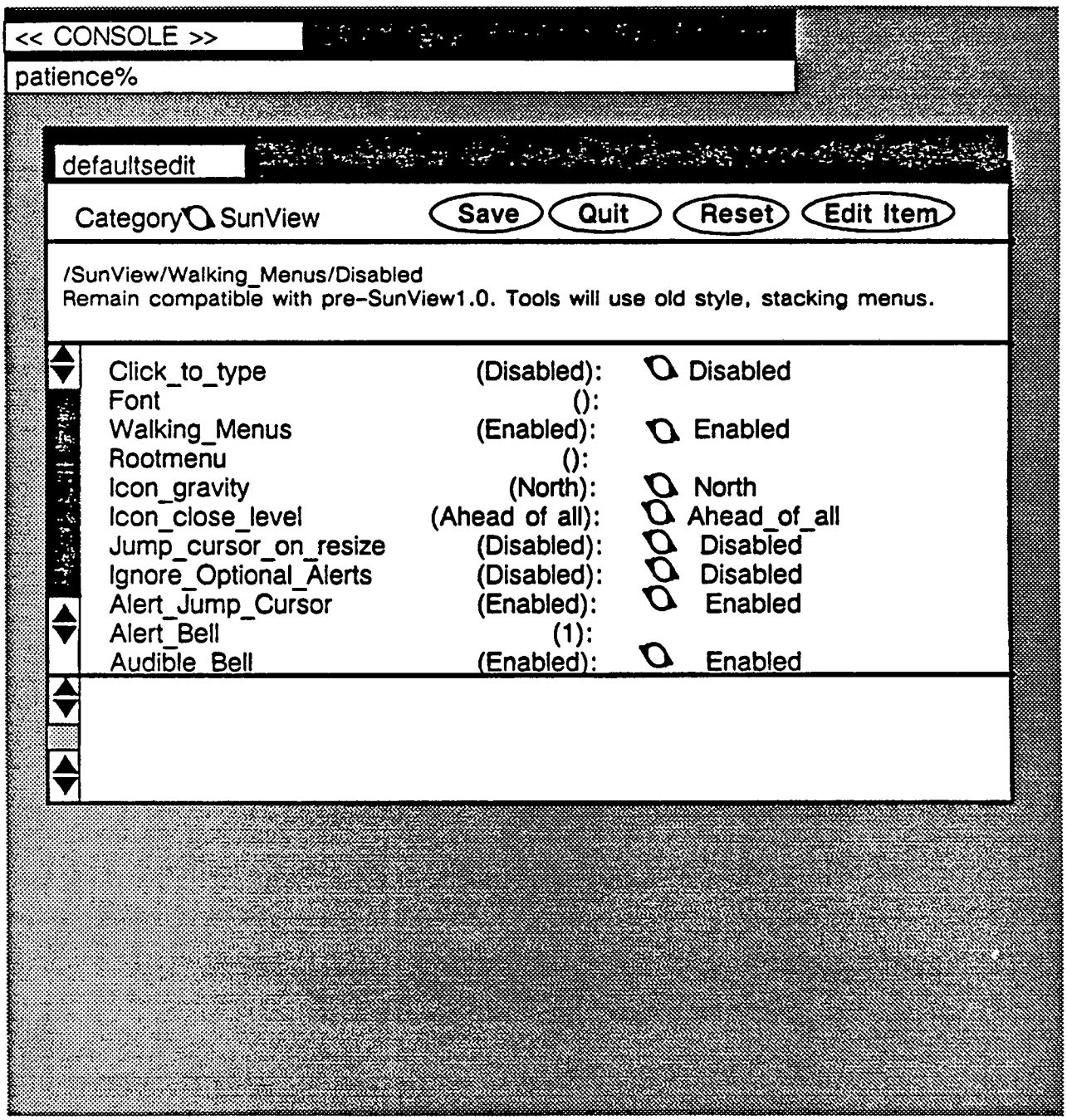
- Select a category, edit and save. If you change SunView parameters, you must exit SunView and restart for the new settings to take effect.



Defaults Editor

Help on each item is available by clicking the left mouse button on the parameter.

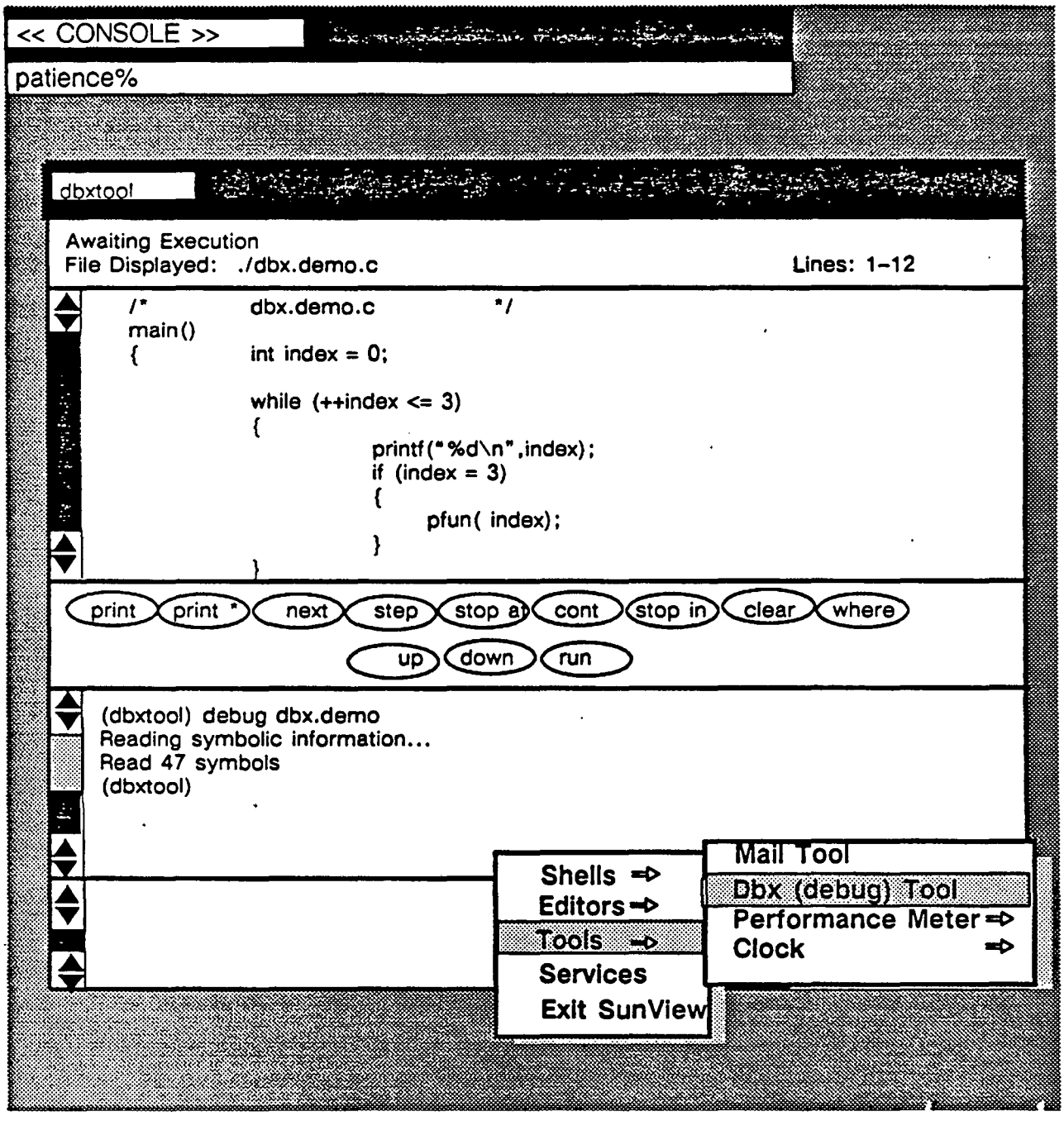
Many of the options are toggles and/or may have built-in menus for easy selection.



Dbx Tool

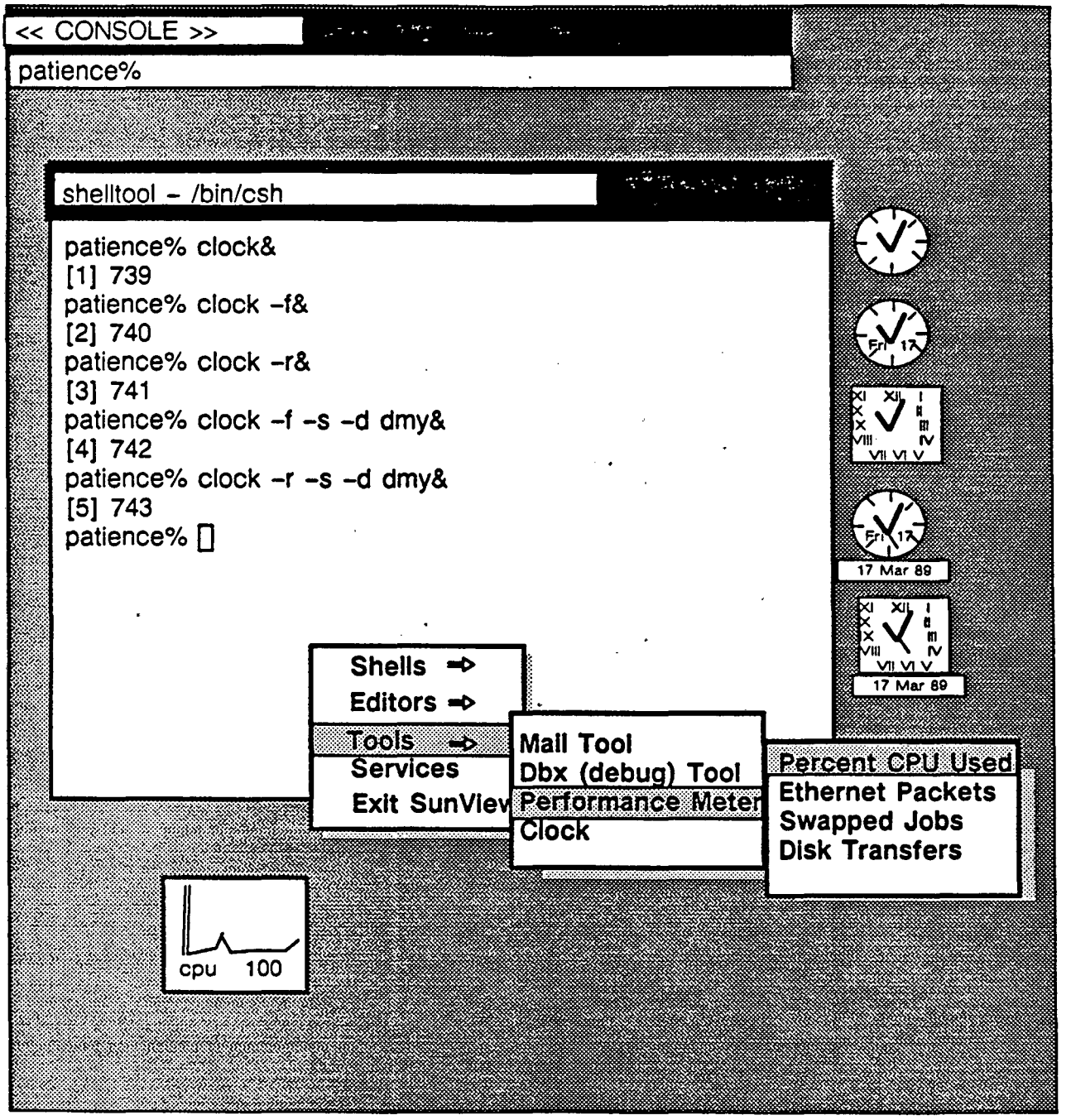
- Highly configurable symbolic debugger for C, FORTRAN 77, and Pascal
- Super set of normal *dbx* commands with built–in help messages on each
- Interactively displays source during runs

Dbx Tool



dbx and *Dbx Tool* are discussed in more detail in the Debugging module of this course.

Clock and Performance Meter



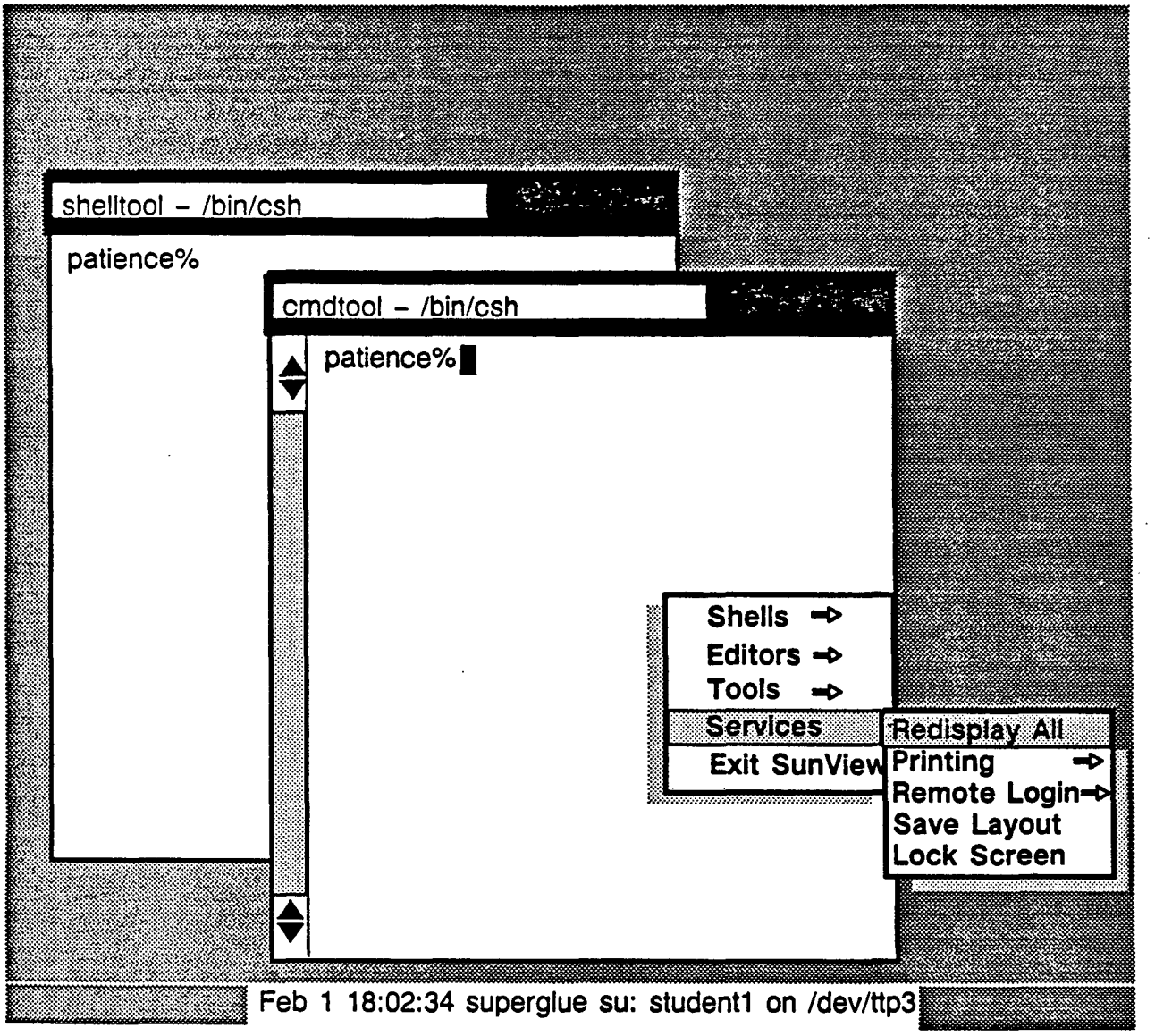
Clock and Performance Meter

clock is distributed with a variety of popular options:

–f	displays date in the face of the clock
–s	displays clock with seconds turned on.
–r	displays clock with square face and roman numerals.
–d [mdywa]	displays date in area below clock face; you can specify any three items to be displayed (month, date, year, day of week, a.m. or p.m.). For example, *clock –d mdw,* activates a clock with today's month, date and day of the week displayed below.

Perfmeter displays one system statistic. To display information about another system in the network, you may also type in *perfmeter systemname* in another shelltool window.

Console Window

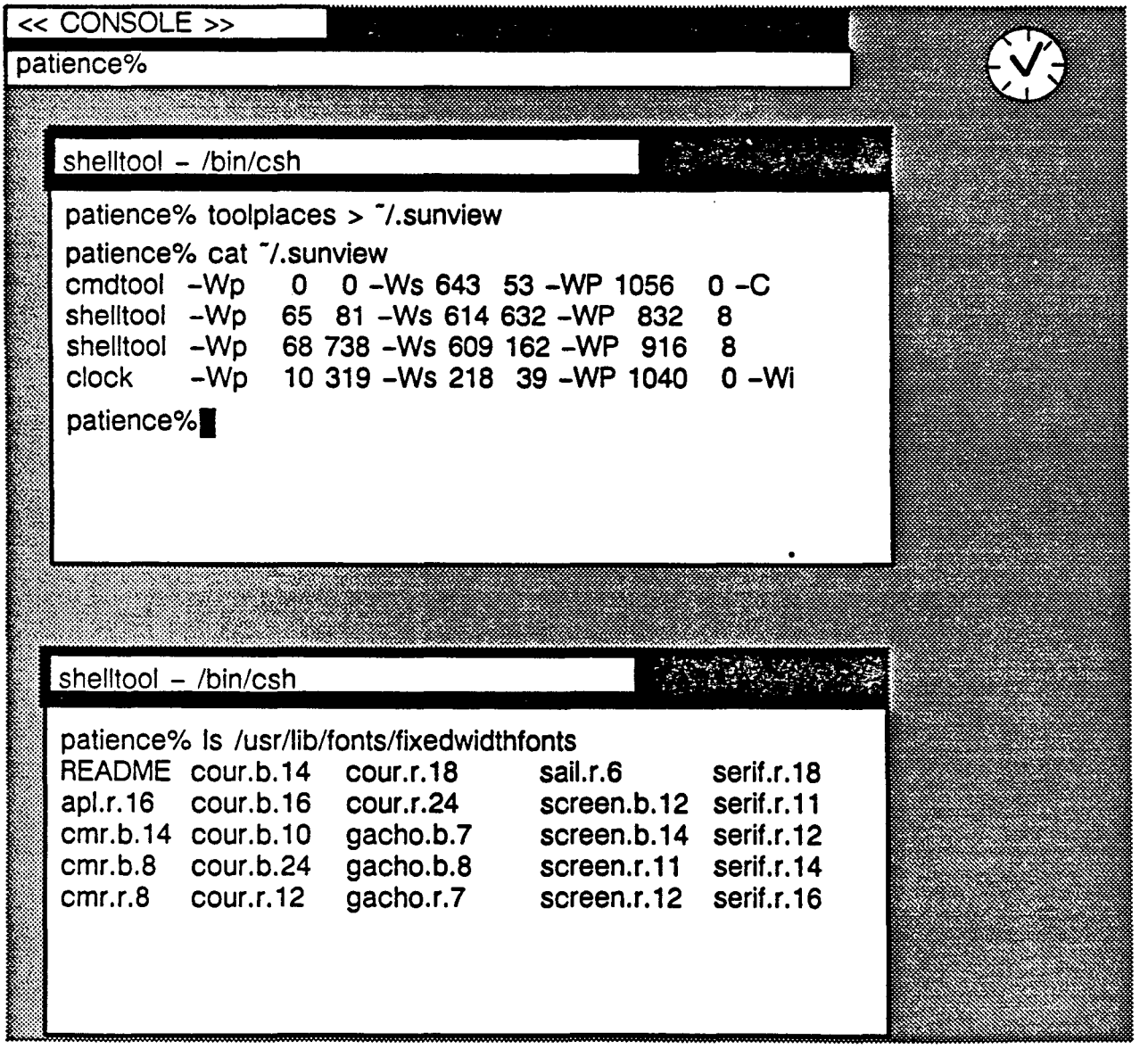


Console Window

If SunOS encounters an error or status message and a console window does not exist, the message corrupts the screen display. You can recover with 'Services ⇨Redisplay All' and 'Shells ⇨Console' from the SunView menu.

Adjusting Fonts

- The directory */usr/lib/fonts/fixedwidthfonts* contains all of the available fonts.
- Use the utility *toolplaces* or select "*Save Layout*" on the SunView menu and either the *–Wt* or *–font* option flags.



Adjusting Fonts

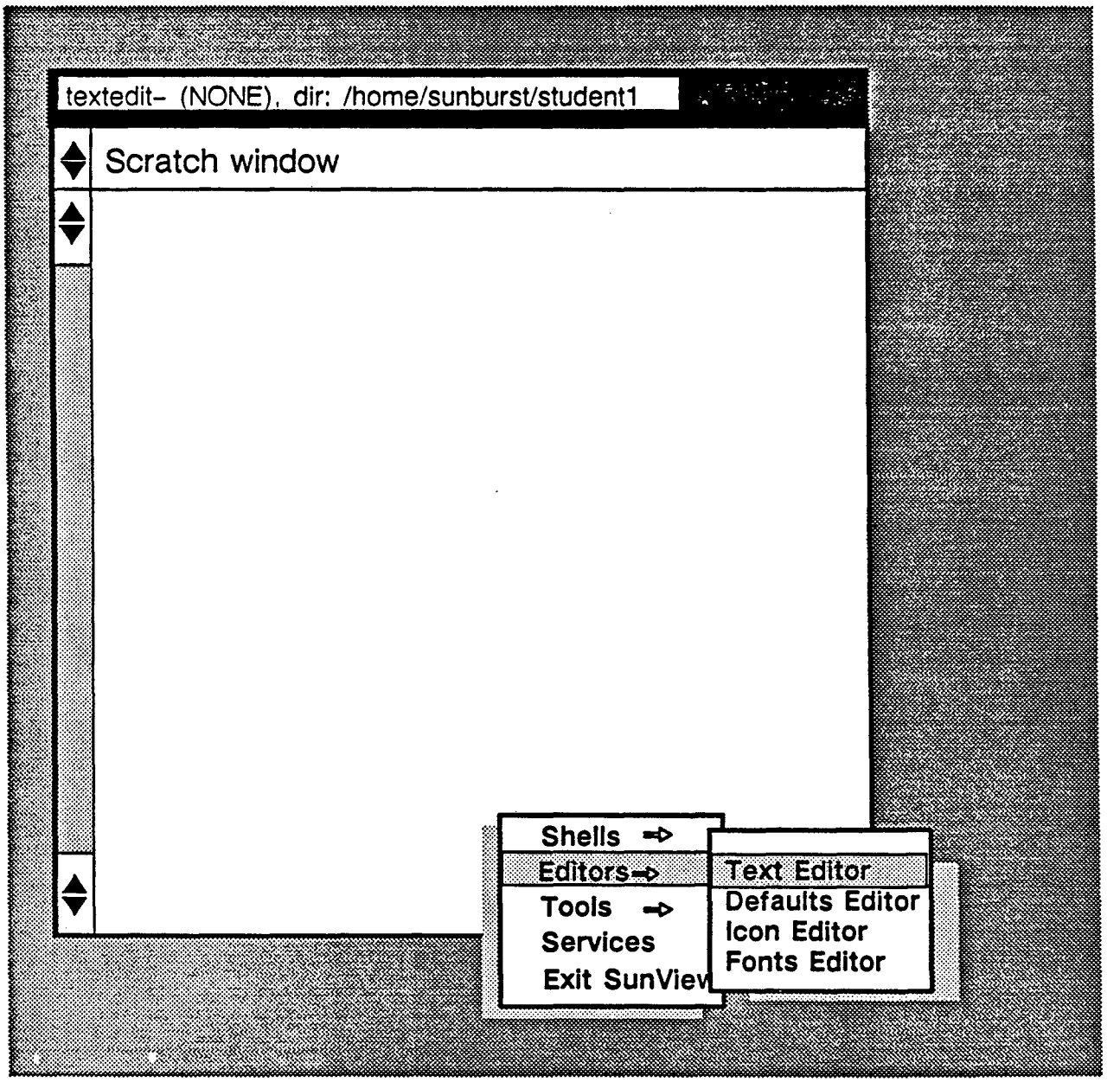
To change the font for the first shelltool window in the *~/.sunview* file, edit the file and change the line to:

shelltool -Wp 65 81 -Ws 614 632 -WP 832 8 -font /usr/lib/fonts/fixedwidthfonts/cour.r.24

where *cour.r.24* is the name of a file (font) in the */usr/lib/fonts/fixedwidthfonts* directory.

Text Editor

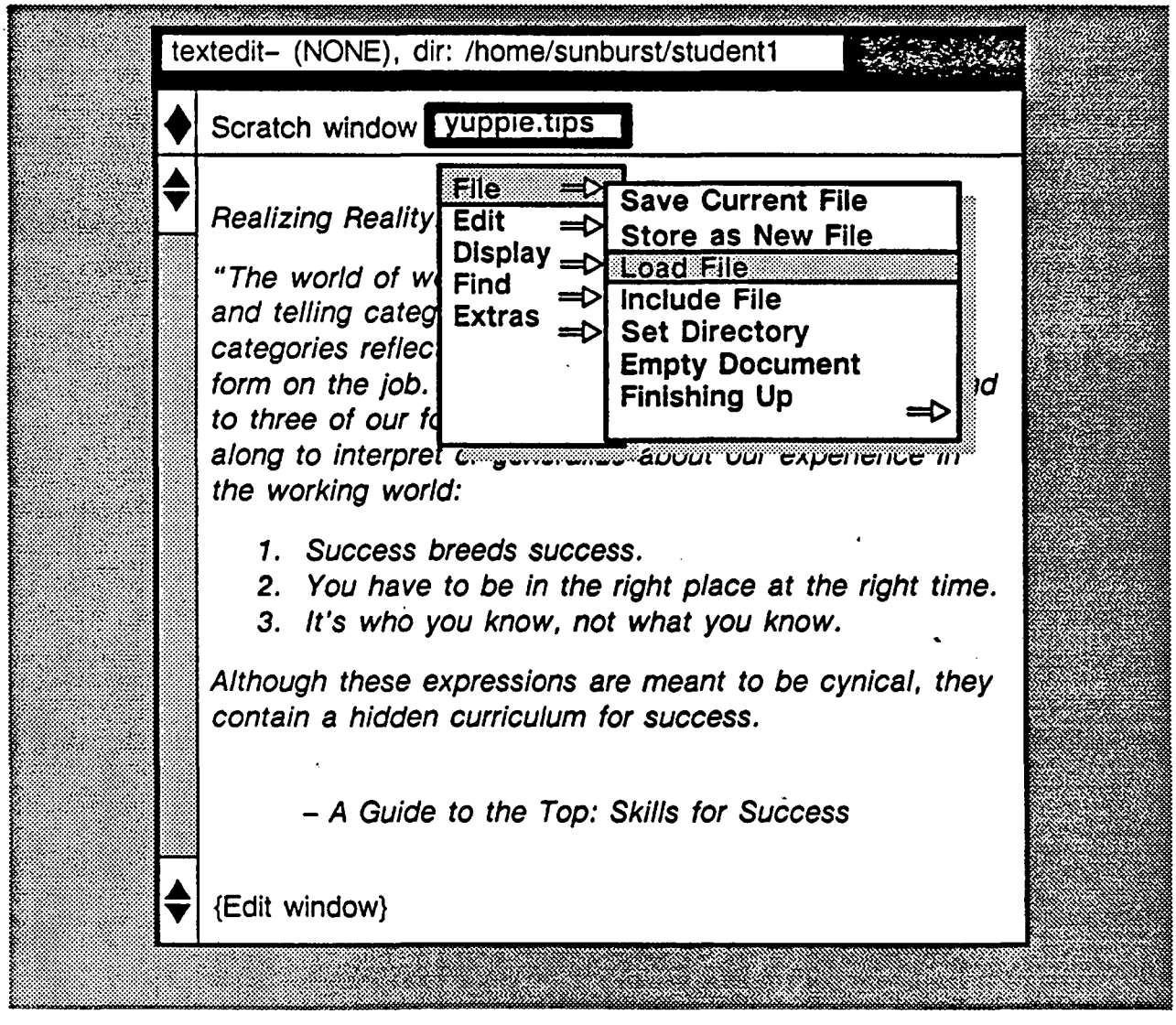
- A mouse–driven editor, which includes all of the editing capabilities. *mailtool* and *cmdtool* use *textedit*.



Text Editor

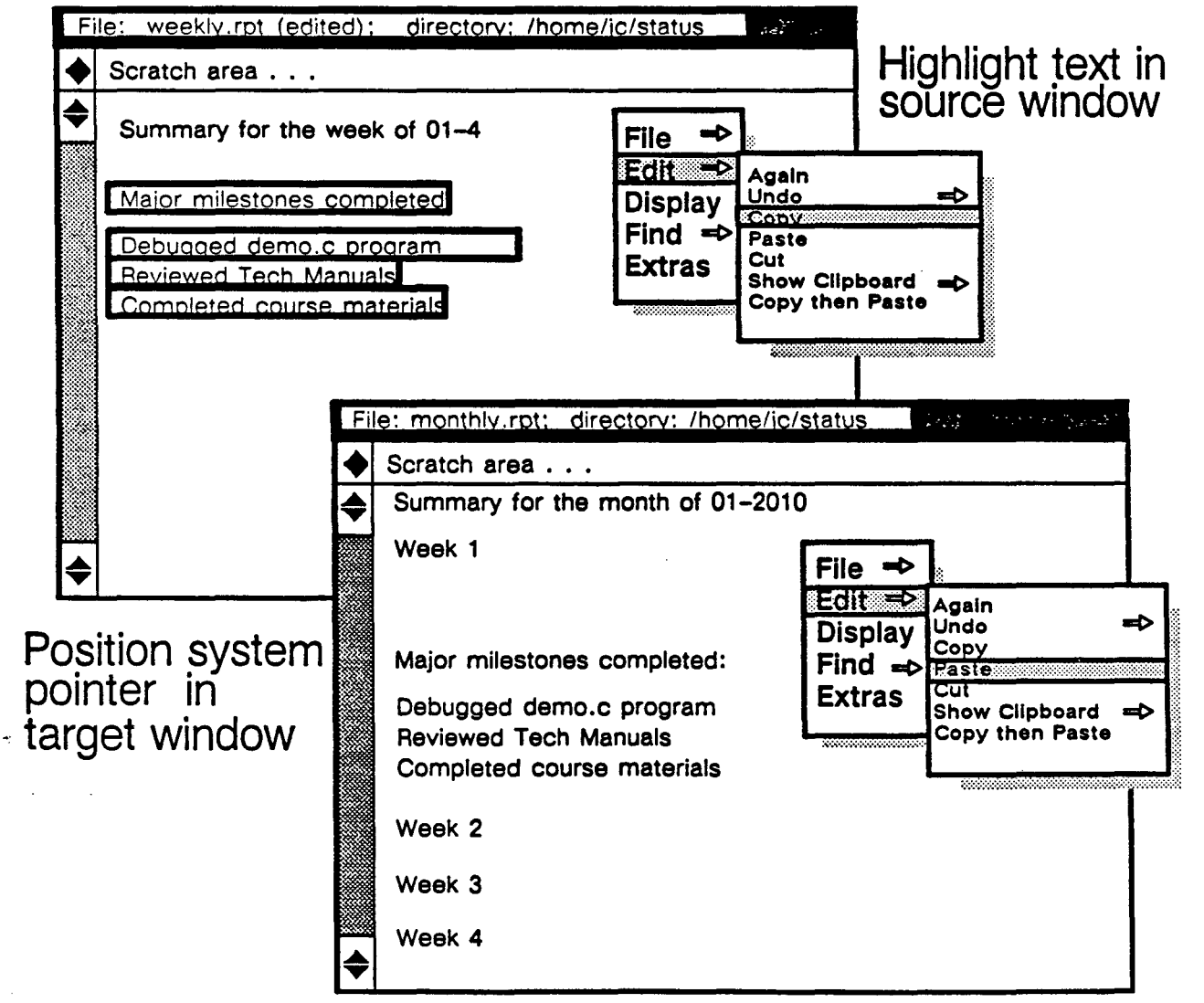
Start up a *textedit* window by selecting "Text Editor" from the "Editors" menu, or enter *textedit &* in a *shelltool* window.

Edit an existing file by entering the name of the file in the scratch window and selecting "load file". Alternately, simply enter the file name in the edit region and hit <ESC>. To create a new file, simply start inserting text into the empty *textedit* text subwindow.



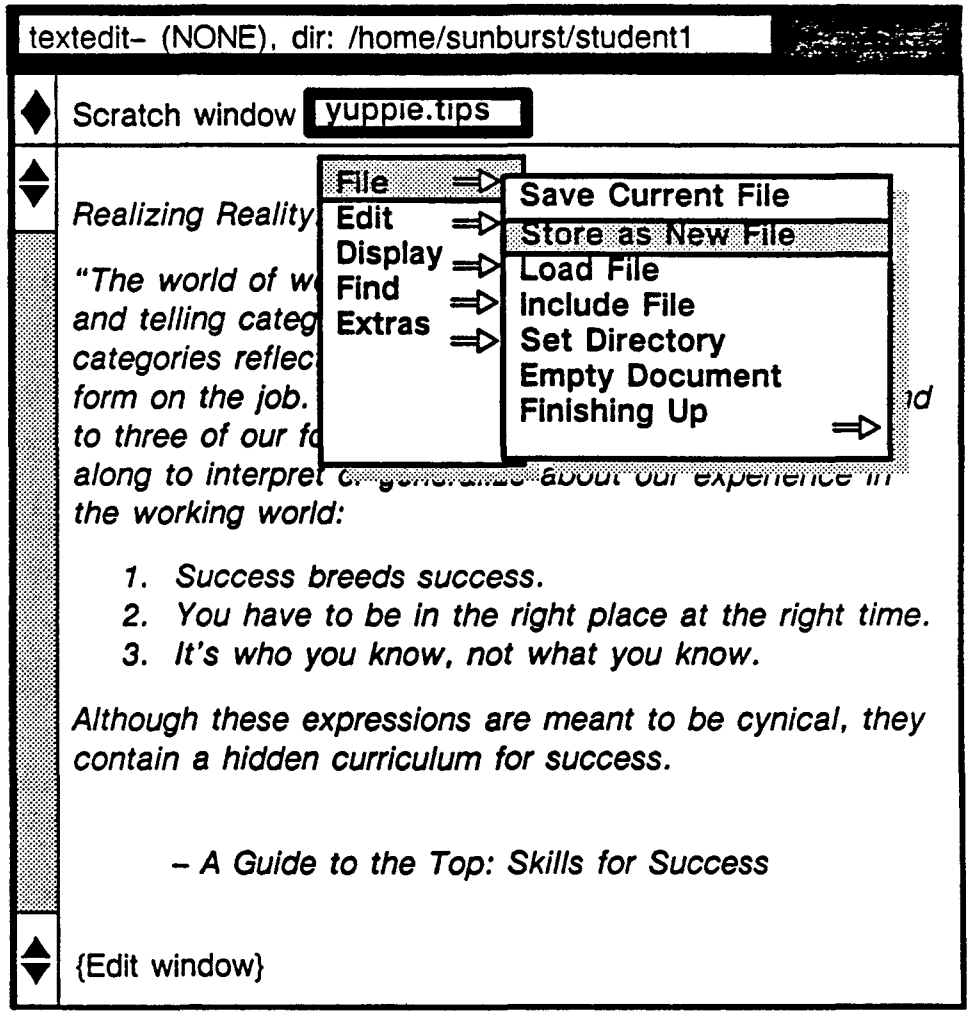
Text Editor (con't.)

- A powerful feature of *textedit* is the ability to copy and paste between *textedit* windows.



Text Editor (con't.)

Save the results of the edit session by entering the name of the file in the scratch window and selecting "Store as New File".



Text Editor Summary

- *Inserting* character or block of text:
  move the cursor to the correct position, click left,
  and enter the new text.

- *Deleting* a single character:
  use the rub-out character (<backspace> commonly)
  to delete character to the left of the caret.
  use *<SHIFT>rub-out character* to delete character
  to right of caret.

- *Deleting* block of text:
  click left on the first character to delete, move the
  cursor to the last character to delete and click middle,
  then press Meta-X or the delete function key
  (usually <L10>).

- *Inserting* deleted text:
  press Meta-V or the get function key (usually <L8>).

- *Moving* text:
  Delete block of text, click left on new location and
  insert deleted text.

- *Copying* text:
  click left on the first character to copy, move the cursor
  to the last character to copy and click middle, then
  press the put function key (usually <L6>).
  Reposition the cursor, click left and press Meta-V.

- *Undoing* changes:
  Press the undo function key (usually <L4>).

- *Scrolling* in file:
  Click middle in a scroll box to go to next screen.
  <SHIFT>click middle displays previous screen.

Text Editor Summary

Note that the "Meta" keys are the diamond (◆) keys on either side of the space bar. On some keyboards these keys may be labeled "Left" and "Right". Use whichever key is more convenient.

Module 4
Regular Expressions

Objectives: Upon completion of this module, the student should be able to:

- Construct regular expressions using defined regular expression characters.
- Use the commands *grep*, *egrep* and *fgrep* with regular expressions to display text patterns in files.
- Use the stream editor *sed.*
- Develop the constructs */RE/ {ACTION}*, *printf()*, *BEGIN*, *END* and built-in variables using *awk*.

Evaluation:

Perform Lab 3 to 90% proficiency.

Reference information:

- Appendix A, *Special Shell Characters.*
- *Getting Started with SunOS: Beginner's Guide* (p/n 800-1703-10), Chapter 8.
- *Doing More with SunOS: Beginner's Guide* (p/n 800-1710-10), Appendix C.
- *The UNIX System: A Sun Technical Report,* Sun Microsystems Inc. (p/n 800-1419-02).
- *The AWK Programming Language,* Alfred Aho, Brian Kernighan and Peter Weinberger, Addison-Wesley Publishing.

Regular Expressions

- Wild cards to match text patterns
- Can include literal characters
- Regular Expression Characters:

*****

.

[abc012]

[a–z]

[^ a–z]

^

$

Regular Expressions

Regular expressions are templates which define character patterns. Most editors and filters use them. They are often incorporated into applications programs, too, so library functions are provided with SunOS to expedite applications development. With its strong orientation toward ASCII files, SunOS data processing and programming relies heavily on these techniques.

A great deal of a computer's time is spent sorting and searching. Programmers also perform these activities extensively. Particularly when working on large software projects, you must often search for things and organize them. Tools that support these activities are extremely valuable.

A set of techniques applicable to source code, documentation, and ASCII data is based on patterns of characters. A *regular expression* is a formal description of such a pattern, and might include literal characters to be matched, and metacharacters such as wild cards, repetition indicators, character classes, and replacement rules.

You use regular expressions to find text patterns in files and perhaps to modify or count them. Most text editors include some kind of regular expression capability for searching within edited files. In particular, the UNIX editors *ed, ex,* and *vi* use regular expressions of the kind described here. So, of course, do *grep, sed*, and *expr*. To a limited extent, the utility *more* (sometimes called *page*) also uses them, as do *tr, cut, lex,* and others.

Take care when using the characters $, *, [,], ^, |, (,), and \ in the regular expression, as these characters are also meaningful to the Shell. It is safest to enclose the entire regular expression argument in single quotes '...', if it contains spaces or characters which have a special meaning to the shell.

Regular Expressions and *grep*

Global *r*egular *e*xpression *p*rinter

- Used to search for text patterns
- Used with regular expressions

Command format:

grep [options] [regular expression] [filename]

Useful options:

–c	count lines where matches occurred
–l	name files
–n	add line numbers
–v	inverse search
–s	work silently
–i	case insensitive

Regular Expressions and *grep*

grep is the SunOS pattern finder. It searches a file or group of files for lines that contain text patterns matching a symbolic regular expression and then prints those lines. *grep* is extremely useful. With it you can find a subroutine lost in one of the fifty modules within the complex program you're working on. You can find that misspelled name that your linker says is an undeclared function. *grep* can number all the lines in a file or do innumerable other things. The power of *grep* lies in its use of regular expressions as pattern templates. There are two other varieties of *grep, fgrep* and *egrep* which are discussed later in this module.

Command Format

The simplest form of *grep* uses the command syntax:

grep [options] [regular expression] [filename]

Using various command line switches when you invoke the program you can get a count of the number of procedures found *–c*, name the files in which a pattern is present *–l*, have the matching line numbers printed with the lines *–n*, list the lines which don't have the requested pattern *–v*, only have exit status set *–s*, and more.

Regular Expressions and *grep*

RE	*Meaning*
c	a single character that matches that character.
.	matches any single character (wild card).[1]
[...]	matches a character class.
[–]	matches a range of characters
^	matches the beginning of a line.
$	matches the end of the line.
*	0 or more occurrences of the preceding character
\	Escapes meaning of an RE character
\(...\)	A group within an RE which can be repeated
\<...\>	Beginning–of–word, End–of–word anchors[2]

1 Except for new–line (By convention, regular expressions are used to match text patterns within lines, not spanning lines.)

2 Understood by *grep* only. May be used separately or together.

Regular Expressions and *grep*

Regular Expression Terminology

Character Classes

A string enclosed in square brackets specifies a *character class*. Any single character in the string will be matched.

For example the command, *grep '[abc]' frisbee* matches every line which contains an *'a'*, *'b'*, or *'c'* in the file *frisbee*. Ranges of ASCII character codes may be abbreviated as in *[a-z0-9]*. If the first symbol following the *[* is a ^ then a *negative character class* is specified. In this case, the string matches all characters *except* those enclosed in the brackets and NEWLINE (e.g. *[^a-z]* matches everything except lower case letters and NEWLINE). Note that a *negative character* class must match something, even though that something cannot be any of the characters listed.

Anchors

^ and *$* are termed *anchors*. These are symbols which match a character's position on the line. Specifically, they match text patterns which are positioned at the beginning ^ or ending *$* of a line of text.

The regular expression *^$* is not the same as *^[^z]$*. The first example matches an empty line (beginning of line followed by end of line); the second example matches a beginning of line followed by any character except a *z* followed by end of line. In the second example, a character must be present on the line, but that character can't be a *z*. Note that *, . and $ are not special characters when inside a character class ([]).

Regular Expressions and *grep*

RE	*Meaning*
c	a single character that matches that character.
.	matches any single character (wild card).[1]
[...]	matches a character class.
[-]	matches a range of characters
^	matches the beginning of a line.
$	matches the end of the line.
*	0 or more occurrences of the preceding character
\	Escapes meaning of an RE character
\(...\)	A group within an RE which can be repeated
\<...\>	Beginning–of–word, End–of–word anchors[2]

Regular Expressions and *grep*

Regular Expression Terminology

Closures (*)

C shell (filename generation) metacharacters have slightly different meanings than the regular expression metacharacters. Specifically, * means *zero or more of any character* to the C shell. The same symbol is termed a *closure* when used in a regular expression. The closure symbol * matches the *preceding* symbol or character any number of times.

Escaping a Regular Expression

A '\' (backslash) followed by a single character matches that character. Thus, a '*' will match an asterisk, a '\.' matches a period, etc. Its purpose is to divest a metacharacter of its special meaning.

\(. . .\)

Backslashed parentheses delimit a group of characters within a regular expression which can be repeated later in the same regular expression such as *\1* (for the first group), *\2* (for the second), and so on. For example the command, *grep '\(abc\).*\1' frisbee* matches lines with at least two occurrences of the string *abc* in the file *frisbee.*

\<

A backslashed left angle bracket is a beginning–of–word anchor. It will match blanks, tabs, punctuation, and the beginning of a line.

\>

A backslashed right angle bracket is a end–of–word anchor. It will match blanks, tabs, punctuation, and the end of a line.

Quoting Regular Expressions

If you are logged in as student1:

Example 1:

patience% **grep "^$user" /etc/passwd**

will be interpreted as:

grep ^student1 /etc/passwd

Example 2:

patience% **grep '^$user' /etc/passwd**

will be interpreted as:

grep ^$user /etc/passwd

Quoting Regular Expressions

Quote Characters

- Single quotes

  '...' disable all metacharacters

- Double quotes

  "..." disable all metacharacters but '$' (variable substitution) and ` ` (command substitution)

- \ (Backslash)

  \ disable the immediately following shell metacharacter.

The shell has three kinds of quoting (i.e. disabling of metacharacters on the command line) single quote–quoting, double quote–quoting, and back-slash–quoting. Please note that single or double quotes do not disable the ! or the \ (backslash). These quoting mechanisms are used to suppress and/or disable shell metacharacters. The shell strips off the quoting mechanisms and then processes the resulting command line.

Regular Expressions and *grep*

- patience% **grep –n '.*' testfile**
- patience% **grep 'a.d' testfile**
- patience% **grep '^a.d' testfile**
- patience% **grep 'a.d$' testfile**
- patience% **grep '^$' testfile**

Regular Expressions and *grep*

The command line, *grep –n '.*' testfile* can be used to output line numbers preceding every line in the file *testfile.*

Regular expressions may be used in combination. For example:

'a.d'

matches any line containing an *'a'* followed by any character followed by a 'd'. This expression will match the strings *"and"* and *"aardvark"* but not *"labrador"*.

'^a.d'

matches the same strings, but only if they're at the beginning of the line. No white space is allowed in front of the *'a.'*

'a.d$'

matches the string at the end of the line (no white space can follow the *'d'* either).

'^$'

matches a beginning of line followed by an end of line. That is, it matches blank lines (lines containing nothing but a new-line character).

Regular Expressions and *grep*

- patience% **grep '^[A–Z]' testfile**

- patience% **grep '0x[0–9a–fA–F][0–9a–fA–F]*' testfile.c**

- patience% **grep 'f[^o]' testfile**

Regular Expressions and *grep*

Pattern Explanations:

"^[A-Z]'

matches a line that begins with an uppercase letter.

'0x[0-9a-fA-F][0-9a-fA-F]'*

matches all hexadecimal numbers in a C program. This last regular expression matches the characters *0x* followed by any of the characters *0123456789abcdefABCDEF* repeated one or more times.

'f[^o]'

finds all occurrences of an 'f' followed by a character that is not an 'o'. Be careful here with patterns at the end of line. "[^o]" matches any character except 'o'. Consequently, a single 'f' at the end of line will not match the pattern because the end-of-line isn't a character in regular expressions, it is a position. 'f[^o]*$' will find 'f's at the end of a line as well as any lines that have an 'f' followed by one or more characters that are not an 'o', followed by end-of-line.

Regular Expressions and *grep*

- patience% **grep 'an*d' testfile**

- patience% **grep 'aa*' testfile**

- patience% **grep '[abc]x' testfile**

- patience% **grep 'who[ls]e' testfile**

Regular Expressions and *grep*

*'an*d'*

matches an *'a'* followed by an *'n'* repeated zero or more times, followed by a *'d'*. So it will match *"pad"* as well as *"panda".*

'aa'*

matches one or more occurrences of the letter *'a'* (as compared to zero or more).

'[abc]x'

employs a character class. A character class matches any one of the characters surrounded by the square brackets. This particular character class matches an 'a,' 'b,' or 'c' followed by an *'x'.*

'who[ls]e'

matches either "whole" and "whose."

The *grep* Family

Varieties of *grep*:

- *fgrep* Fixed *grep*.
  RE metacharacters **are not** supported.

- *egrep* Extended *grep*.

  Supports additional RE metacharacters:

RE	*Meaning using egrep*
*	zero or more occurrence of the regular expression.
+	matches one or more occurrences of the RE
?	matches zero or one occurrences of a one character RE
|	A pair of regular expressions separated by a '|' matches either.
( )	Groups REs together

The *grep* Family

fgrep

fgrep patterns are fixed strings i.e. *lampoon*, *fork* etc. Regular expression metacharacters are *not* supported.

egrep

The command *egrep* stands for extended *grep*. In general, *egrep* is the fastest member of the *grep* family. *egrep* accepts regular expressions of the same sort *grep* does, except for *\(*, *\)*, *\n*, *\<*, and *\>*.

*

A regular expression followed by an asterisk (*) matches zero or more occurrences of that regular expression.

+

A regular expression followed by a plus sign (+) is a regular expression that matches one or more occurrences of the one-character regular expression.

?

A regular expression followed by a question mark (?) is a regular expression that matches zero or one occurrences of the one-character regular expression.

NOTE: *, +, and ? originally applied to just single-character regular expressions. Now they can also be applied to groups (longer expressions within parentheses).

The *grep* Family

- *egrep* Extended *grep.*

  Supports additional RE characters:

RE	*Meaning using egrep*
*	zero or more occurrence of the regular expression.
+	matches one or more occurrences of the RE
?	matches zero or one occurrences of a one character RE
|	A pair of regular expressions separated by a '|' matches either.
( )	Groups REs together

Order of precedence (within same parenthesis level):

```
[]
+  *  ?
concatenation
|
new–line
```

The *grep* Family

|

This symbol is called an *alternation*. When two regular expressions are separated by | or a new-line character (within a script), a match can come from either the first or the second regular expression. The expression *dog|cat* matches the string *"dog"* or the string *"cat"*.

( )

Parentheses (. . .) can be used to group together an expression so that '*', '?', and '+' can be applied to the entire expression.

Use "\(" and "\)" to search for open or close parentheses. *egrep* does not have *grep's* backslashed parentheses for matching repeated groups.

Order of Precedence of Operators

The order of precedence of operators at the same parenthesis level is *[ ]* (character classes), then * + ? (closures), then concatenation (patterns of plain characters), then | (alternation) and new-line.

The *grep* Family

- patience% **egrep '^(ab)+$' whiz**

  matches lines containing only one or more occurrences of the string *'ab'*.

- patience% **egrep 'f[^a–zA–Z]|f$' gee**

  matches all lines containing a word ending in 'f'.

- patience% **egrep '^#define[ ]+[a–zA–Z_]+\(' doody.c**

- patience% **egrep 'Carol (Briggs | Prochnow)' golly**

The *grep* Family

'^(ab)+$'

matches lines containing only one or more occurrences of the string *'ab'*.

'f[^a-zA-Z]|f$'

matches all lines containing a word ending in *'f'*.

You can find subroutine-like macros with:

'^#define[ ]+[a-zA-Z_]+\('

The *[ ]* contains a tab and a space. This expression is interpreted as beginning of line ('^') followed by the string "#define" followed by at least one space or tab ("[ ]+") followed by at least one letter or underscore ("[a-zA-Z_]+") followed immediately (no intervening space) by an open parenthesis ('(').

Finally, you can combine grouping with alternation:

'Carol (Briggs | Prochnow)'

This will search for lines containing either "Carol Briggs" or "Carol Prochnow".

Regular Expressions: Counting Comments in C Programs

```
patience% cd /usr/src/fortune

patience% ls
Makefile  README  obscene  scene     strfile.h
Notes     fortune.c  rnd.c     strfile.c  unstr.c

patience% cat README
This is the Berkeley fortune database program...

patience% grep "/ \*" *.[ch]
fortune.c:/* $Header: fortune.c,v 1.18 87/05/08 13:26:02
arnold Exp $ */
fortune.c:# define MINW  6  /* minimum wait if desired */
fortune.c:# define CPERS 20 /* # of chars for each sec */
...
rnd.c:/*
rnd.c:/*
rnd.c:/*
rnd.c:/*
...
strfile.h:/* @(#)strfile.h        1.2 (Berkeley) 5/14/81 */
...
```

Regular Expressions: Counting Comments in C Programs

This example and the examples which follow, use the *fortune* game program written by Ken Arnold of UCB. The purpose is to demonstrate techniques of regular expressions, shell and *awk* programming.

This first attempt at comment counting is a simplistic approach using *grep* and regular expressions. We will become more sophisticated in our techniques to count source code comments and *"switch"* statements. We will also demonstrate a way to compute the percentage of comments versus non-code and non-blank lines using *sed* and *awk*.

First, we position ourselves in the directory in which *fortune*'s source code resides. Then we look to see what files are there and we read the *README* file.

Next, we wish to study *fortune*'s comments. We retrieve from the source code files (those with .c and .h suffixes) all the comment lines. Comment lines are those containing the character pair *"/*"*.

Note that the command line with which we accomplish this contains two asterisks, each with a different meaning. The first asterisk would have its regular expression meaning ("zero or more of the preceding character"), since it's the second argument to a *grep* command, except that it's quoted by a backslash, which causes it to be treated simply as an asterisk character, not as a metacharacter.

The second asterisk has its filename generation meaning. Note also that the entire regular expression is surrounded by double quotes. This prevents the shell from doing any filename generation, variable substitution, or other processing on the characters within the quotes. We want these characters conveyed to *grep* unaltered. When *grep* is told to search more than one file, it prepends the filename to the line it found.

Regular Expressions: Counting Commands in C Programs

```
patience% grep "/ \*" *.[ch] | wc -l
197
patience% grep switch *.[ch]
fortune.c:                      switch (*sp) {
rnd.c:/* bytes.  The state can be switched by
rnd.c:      switch( type ) {
strfile.c:                      switch (*sp) {
unstr.c:                        switch (*sp) {

patience% grep switch *.[ch] | wc -l
5

patience% grep -c 'switch[^a-z]' *.[ch]
4
```

Regular Expressions: Counting Commands in C Programs

Finally, we repeat the *grep* command and pipe its output to *wc –l* which tells us how many lines *grep* found. This is an approximation of the number of comments in the source code.

Later in this module we shall refine our techniques for counting comments.

Next, we search for the *switch* statement in the C source code, and count how many times it's found by piping *grep*'s output to *wc*. In our first attempt, we mistakenly include in our count a comment which happens to contain the word "switched."

Our second attempt, though not foolproof, refines the search with an improved regular expression. We no longer include the comment in the tally, so the line count accordingly drops from five to four.

sed: A Stream Editor

sed

- Non–interactive *s*treaming *ed*itor, not buffered.
- Uses Regular Expressions

Command Format

sed [options] '[commands/arguments]' [file(s)] > newfile

sed: A Stream Editor

sed stands for stream editor. It does not require direct interaction with users, and hence has also been termed a *batch* editor. This is in contrast to such editors as *vi* and *ed* which are interactive and *WYSIWYG* (what you see is what you get). *sed* takes its input from a pipe or from a file and applies a transformation which has been specified by means of regular expressions, which are combined into commands with syntax much like that of *ed*.

sed is capable of performing text pattern substitutions and text pattern deletions using regular expression syntax. *sed* commands can also be stored in a script file. The script file can then be called and run against the data file to perform repetitive editing operations.

For example,

sed "s/[Mm]allomar/Snickers/" hungry > stuffed

creates a file called *stuffed* which is a copy of *hungry* with all the Mallomars (even those spelled with a lowercase 'm') replaced by Snickers.

In fact, *sed*'s command language is a powerful programming language unto itself. However, its most popular use is in one-liners like this one.

Using *sed* to Substitute Text

```
patience% cat data.file
northwest  NW  Joel Craig      3.0  .98  3  4
western    WE  Sharon Kelly    5.3  .97  5  23
southwest  SW  Chris Foster    2.7  .8   2  18
southern   SO  May Chin        5.1  .95  4  15
southeast  SE  Derek Johnson   4.0  .7   4  17
eastern    EA  Susan Beal      4.4  .84  5  20
northeast  NE  TJ Nichols      5.1  .94  3  13
north      NO  Val Shultz      4.5  .89  5  9
central    CT  Sheri Watson    5.7  .94  5  13

patience% sed 's/Kelly/Haskel/' data.file
northwest  NW  Joel Craig      3.0  .98  3  4
western    WE  Sharon Haskel   5.3  .97  5  23
southwest  SW  Chris Foster    2.7  .8   2  18
southern   SO  May Chin        5.1  .95  4  15
southeast  SE  Derek Johnson   4.0  .7   4  17
eastern    EA  Susan Beal      4.4  .84  5  20
northeast  NE  TJ Nichols      5.1  .94  3  13
north      NO  Val Shultz      4.5  .89  5  9
central    CT  Sheri Watson    5.7  .94  5  13

patience% sed 's/3/9/g' data.file
northwest  NW  Joel Craig      9.0  .98  9  4
western    WE  Sharon Haskel   5.9  .97  5  29
southwest  SW  Chris Foster    2.7  .8   2  18
southern   SO  May Chin        5.1  .95  4  15
southeast  SE  Derek Johnson   4.0  .7   4  17
eastern    EA  Susan Beal      4.4  .84  5  20
northeast  NE  TJ Nichols      5.1  .94  9  19
north      NO  Val Shultz      4.5  .89  5  9
central    CT  Sheri Watson    5.7  .94  5  19
```

Using *sed* to Substitute Text

In this first example, the 's' (substitute) command is used to replace the first occurrence of the characters 'Kelly' with the characters 'Haskel' in *data.file.* The change occurs on line 2.

In the second example, every occurrence ('*g*' means global) of the character '3' is replaced with the character '9'.

Using *sed* to Substitute Text (con't.)

```
patience% cat data.file
northwest  NW  Joel Craig      3.0  .98    3  4
western    WE  Sharon Kelly    5.3  .97    5  23
southwest  SW  Chris Foster    2.7  .8     2  18
southern   SO  May Chin        5.1  .95    4  15
southeast  SE  Derek Johnson   4.0  .7     4  17
eastern    EA  Susan Beal      4.4  .84    5  20
northeast  NE  TJ Nichols      5.1  .94    3  13
north      NO  Val Shultz      4.5  .89    5  9
central    CT  Sheri Watson    5.7  .94    5  13
```

```
patience% sed 's/      \.9./& */' data.file
northwest  NW  Joel Craig      3.0  .98 *  3  4
western    WE  Sharon Kelly    5.3  .97 *  5  23
southwest  SW  Chris Foster    2.7  .8     2  18
southern   SO  May Chin        5.1  .95 *  4  15
southeast  SE  Derek Johnson   4.0  .7     4  17
eastern    EA  Susan Beal      4.4  .84    5  20
northeast  NE  TJ Nichols      5.1  .94 *  3  13
north      NO  Val Shultz      4.5  .89    5  9
central    CT  Sheri Watson    5.7  .94 *  5  13
```

```
patience%sed  '/west/,/east/s/.$/&  stuff/' data.file
northwest  NW  Joel Craig      3.0  .98    3  4 stuff
western    WE  Sharon Kelly    5.3  .97    5  23 stuff
southwest  SW  Chris Foster    2.7  .8     2  18 stuff
southern   SO  May Chin        5.1  .95    4  15 stuff
southeast  SE  Derek Johnson   4.0  .7     4  17 stuff
eastern    EA  Susan Beal      4.4  .84    5  20 stuff
northeast  NE  TJ Nichols      5.1  .94    3  13 stuff
north      NO  Val Shultz      4.5  .89    5  9
central    CT  Sheri Watson    5.7  .94    5  13
```

(Folder)

Using *sed* to Substitute Text (con't.)

The first example illustrates the use of *sed* and regular expressions to substitute text.

The second example uses pattern to pattern matching and line number addressing.

Notice the special characters which are used:

\	use as an escape character
&	insert pattern found into the replacement string
.	any single character
*	0 or more occurrences of preceding character
$	end of line anchor
^	beginning of line anchor
[ ]	set

Using *sed* to Delete Text

```
patience% cat data.file
northwest  NW  Joel Craig      3.0  .98  3  4
western    WE  Sharon Kelly    5.3  .97  5  23
southwest  SW  Chris Foster    2.7  .8   2  18
southern   SO  May Chin        5.1  .95  4  15
southeast  SE  Derek Johnson   4.0  .7   4  17
eastern    EA  Susan Beal      4.4  .84  5  20
northeast  NE  TJ Nichols      5.1  .94  3  13
north      NO  Val Shultz      4.5  .89  5  9
central    CT  Sheri Watson    5.7  .94  5  13

patience% sed /west/d data.file
southern   SO  May Chin        5.1  .95  4  15
southeast  SE  Derek Johnson   4.0  .7   4  17
eastern    EA  Susan Beal      4.4  .84  5  20
northeast  NE  TJ Nichols      5.1  .94  3  13
north      NO  Val Shultz      4.5  .89  5  9
central    CT  Sheri Watson    5.7  .94  5  13

patience% sed 2d data.file
northwest  NW  Joel Craig      3.0  .98  3  4
southwest  SW  Chris Foster    2.7  .8   2  18
southern   SO  May Chin        5.1  .95  4  15
southeast  SE  Derek Johnson   4.0  .7   4  17
eastern    EA  Susan Beal      4.4  .84  5  20
northeast  NE  TJ Nichols      5.1  .94  3  13
north      NO  Val Shultz      4.5  .89  5  9
central    CT  Sheri Watson    5.7  .94  5  13
```

Using *sed* to Delete Text

The 'd' (delete) command is used to delete all lines which contain the characters 'west'.

The 'd' command can also be used with line number(s) to delete a particular line. If you wanted to delete lines 2 through 5, the command format would be:

```
sed 2,5d data.file
```

Another use of the 'd' command is to delete blank (or apparently blank) lines from a file. In the command format below, please note that the brackets contain a space and a tab.

```
sed  '/^[          ]*$/d' filename
```

Using *sed* to Read in Data from Another File

```
patience% cat enclosure
**********************************
*                                *
*                                *
*                                *
**********************************
```

```
patience% sed ' /Beal/r enclosure' datafile
northwest   NW   Joel Craig      3.0  .98   3   4
western     WE   Sharon Kelly    5.3  .97   5   23
southwest   SW   Chris Foster    2.7  .8    2   18
southern    SO   May Chin        5.1  .95   4   15
southeast   SE   Derek Johnson   4.0  .7    4   17
eastern     EA   Susan Beal      4.4  .84   5   20
**********************************
*                                *
*                                *
*                                *
**********************************
northeast   NE   TJ Nichols      5.1  .94   3   13
north       NO   Val Shultz      4.5  .89   5   9
central     CT   Sheri Watson    5.7  .94   5   13
```

Using *sed* to Read in Data from Another File

Another useful *sed* command is 'r' (read). It allows you to read in the contents of another file. If you wanted to save the new file, simply send the *sed* output to a new file. For instance,

sed ' /Beal/r enclosure' data.file > new.output.file

sed Options

```
patience% sed -e /west/d -e s/CT/CE/ data.file
southern    SO    May Chin       5.1  .95    4    15
southeast   SE    Derek Johnson  4.0  .7     4    17
eastern     EA    Susan Beal     4.4  .84    5    20
northeast   NE    TJ Nichols     5.1  .94    3    13
north       NO    Val Shultz     4.5  .89    5    9
central     CE    Sheri Watson   5.7  .94    5    13

patience% sed -n 4,8p data.file
southern    SO    May Chin       5.1  .95    4    15
southeast   SE    Derek Johnson  4.0  .7     4    17
eastern     EA    Susan Beal     4.4  .84    5    20
northeast   NE    TJ Nichols     5.1  .94    3    13
north       NO    Val Shultz     4.5  .89    5    9

patience% sed -n '/west/w newdata' data.file
patience% cat newdata
northwest   NW    Joel Craig     3.0  .98    3    4
western     WE    Sharon Kelly   5.3  .97    5    23
southwest   SW    Chris Foster   2.7  .8     2    18
```

sed Options

sed has several options. The *–e* option is the "edit" flag. If only one edit is to be done on a file, the flag is not necessary.

The *–n* option suppresses normal (full file) output. In the second example, the *–n* flag is used with the *p* (print) command to only print lines 4 through 8.

The next example combines the *–n* flag with the *w* (write) command to print lines containing the characters 'west' to a new file called *newdata.*

sed Scripts

```
patience% cat data.file
northwest   NW   Joel Craig      3.0  .98   3   4
western     WE   Sharon Kelly    5.3  .97   5   23
southwest   SW   Chris Foster    2.7  .8    2   18
southern    SO   May Chin        5.1  .95   4   15
southeast   SE   Derek Johnson   4.0  .7    4   17
eastern     EA   Susan Beal      4.4  .84   5   20
northeast   NE   TJ Nichols      5.1  .94   3   13
north       NO   Val Shultz      4.5  .89   5   9
central     CT   Sheri Watson    5.7  .94   5   13
```

```
patience% cat sed.sc1
s/De.*son/Frank Marcus/
/west/d
s/ [0-9] / /
```

```
patience% sed -f  sed.sc1  data.file
southern    SO   May Chin        5.1  .95   15
southeast   SE   Frank Marcus    4.0  .7    17
eastern     EA   Susan Beal      4.4  .84   20
northeast   NE   TJ Nichols      5.1  .94   13
north       NO   Val Shultz      4.5  .89   9
central     CT   Sheri Watson    5.7  .94   13
```

sed Scripts

The *–f* option takes *sed* input from a specified file, instead of on the command line. This makes it easy to perform several *sed* operations on a file at once.

sed Scripts (con't.)

```
patience% cat sed.sc2
/Craig/a\
-->    Mr. Craig has been ill approx. 1 wk
/Nichols/i\
\
-->    Ms. Nichols is a new agent \
-->    promoted from NE dist
/NO/c\
-=-=- The north dist office is closed until further notice -=-=-

patience% sed -f sed.sc2 data.file
northwest  NW    Joel Craig      3.0  .98    3    4
-->    Mr. Craig has been ill approx. 1 wk
western    WE    Sharon Kelly    5.3  .97    5    23
southwest  SW    Chris Foster    2.7  .8     2    18
southern   SO    May Chin        5.1  .95    4    15
southeast  SE    Derek Johnson   4.0  .7     4    17
eastern    EA    Susan Beal      4.4  .84    5    20

-->    Ms. Nichols is a new agent
-->    promoted from NE dist
northeast  NE    TJ Nichols      5.1  .94    3    13
-=-=- The north dist office is closed until further notice -=-=-
central    CT    Sheri Watson    5.7  .94    5    13
```

sed Scripts (con't.)

This script uses three additional *sed* commands.

a\	
text	appends text to output before reading next input
i\	
text	inserts text on standard output
c\	
text	change the pattern space

For more details on other *sed* commands, refer to the *man* page.

Regular Expressions and *sed*

```
patience% more rnd.c
...
/* seed for R. N. G. */      unsigned seed;
/* pointer to state array */ char     *arg_state;
/* # bytes of state info */  int      n;
...
patience% sed  -f com-fix.sed  rnd.c > /tmp/rnd.c
patience% more /tmp/rnd.c
...
     unsigned seed;     /* seed for R. N. G. */
   char     *arg_state;  /* pointer to state array */
   int      n;           /* # bytes of state info */
...
patience% mv !$ .
patience% cat com-fix.sed
s,^\([    ]*[^ ]*.*\*[^)]\(.*\),\2   \1,
```

Regular Expressions and *sed*

While reading *rnd.c* (the subroutine that generates random numbers) with the utility *more*, we discover lines in which the comments precede the code. Here we create a script which exchanges code and comment within such lines.

We name it *com-fix.sed.* The shell attributes no special meaning to minus signs or periods, so it's proper syntax to use them in filenames.
com-fix.sed is a script to drive *sed,* the stream editor. We run it on *rnd.c* and preserve its output in the temporary storage area */tmp*.

Next, we read the temporary file (again with *more)* to make sure *sed* performed what we expected.

We replace the current version with the modified version.

Finally, we look at *com-fix.sed* itself. This is a *sed* script consisting of a single *sed* command. The command says to substitute one regular expression for another. The substitution occurs on lines that contain white space followed by a C comment, followed by anything else (i.e. source code). Source and comment sub-patterns are then swapped.

The script makes use of *sed*'s ability to use any character for a substitution delimiter. The convention is to use '/', but here we use ',' instead. The '/' is part of the search string, and we wish to avoid the effort of quoting it.

The *awk* Programming Language

Features of *awk:*

- Uses Regular Expressions
- Uses numeric and text variables and functions
- Fields and Records

awk Applications:

- Filters
- Numerical processing on rows and columns of data
- Text processing to perform repetitive editing tasks

The *awk* Programming Language

awk is a record oriented language which is named for its authors, Aho, Weinberger and Kernighan of Bell Labs.

awk grew out of the recognition that many data processing problems are simply specialized applications of the concept of filtering, where the data is structured into records to which transformations are repetitively applied.

Unlike *sed, awk* looks at data by records (lines) and fields. By default, records are delimited by new-line characters, and the fields within them by spaces or tabs, but these can be reset to whatever delimiters are built-in to your data.

awk Programming Constructs

Summary of Constructs discussed:

- */RE/ {ACTION}*
- *{ACTION}*
- *BEGIN* {statements}
- *END* {statements}
- */RE/*

- Execute *awk* scripts using this format:

  *awk [–***f** *scriptfile] [input.file]*

awk Programming Constructs

awk commands can be combined in an *awk* scripts which consist of one or more lines of the form:

PATTERN {*ACTION*}

PATTERN controls when an *ACTION* is to be done. *ACTION* is one or more statements written in the *awk* language. One of the most common *PATTERN*s is a regular expression bounded/delimited by slashes.

The *awk* language is very much like C. One major difference is that in *awk*, variables are not explicitly declared. Unlike in C, *awk* variables are dynamically allocated and initialized the first time they are referenced. Another difference is that arrays in *awk* can be indexed by strings (making them associative, a very powerful technique).

With the first construct, *awk* performs the *ACTION* on every input record (usually simply a line of text) which matches the */RE/*. The second construct performs the *ACTION* on all lines of the file. *BEGIN* and *END* mean that the corresponding *ACTIONs* are to be performed before and after, respectively, the processing of the data. And, finally, if /RE/ stands by itself, with no action, all lines containing a match are printed to the screen (like *grep*).

For example, *END* might be used to cause an accumulated total to be printed.

Using *awk* to Print Selected Fields

```
patience% cat data.file
northwest  NW  Joel Craig      3.0  .98   3  4
western    WE  Sharon Kelly    5.3  .97   5  23
southwest  SW  Chris Foster    2.7  .8    2  18
southern   SO  May Chin        5.1  .95   4  15
southeast  SE  Derek Johnson   4.0  .7    4  17
eastern    EA  Susan Beal      4.4  .84   5  20
northeast  NE  TJ Nichols      5.1  .94   3  13
north      NO  Val Shultz      4.5  .89   5  9
central    CT  Sheri Watson    5.7  .94   5  13

patience% awk '{print $3, $4, $2}' data.file
Joel Craig NW
Sharon Kelly WE
Chris Foster SW
May Chin SO
Derek Johnson SE
Susan Beal EA
TJ Nichols NE
Val Shultz NO
Sheri Watson CT

patience% awk '{print $3, $4 "    " $2}' data.file
Joel Craig      NW
Sharon Kelly    WE
Chris Foster    SW
May Chin        SO
Derek Johnson   SE
Susan Beal      EA
TJ Nichols      NE
Val Shultz      NO
Sheri Watson    CT
```

Using *awk* to Print Selected Fields

All *awk* commands are enclosed in single quotes and curly braces *'{ }'*.

The *print* command is used to output data from the file.

All input records in the file are separated with a tab or a space. *'$0'* is the entire line. *'$1'* is the first field, *'$2'* is the second, and so on. Fields are separated in the command line by commas.

In this example, fields 3 (first name), 4 (last name) and 2 (office) are printed out.

By adding tabs or other text, inside of double quotes, you can present the output more neatly.

Using *awk* to Print Pattern Matches

```
patience% cat data.file
northwest   NW   Joel Craig      3.0  .98   3   4
western     WE   Sharon Kelly    5.3  .97   5   23
southwest   SW   Chris Foster    2.7  .8    2   18
southern    SO   May Chin        5.1  .95   4   15
southeast   SE   Derek Johnson   4.0  .7    4   17
eastern     EA   Susan Beal      4.4  .84   5   20
northeast   NE   TJ Nichols      5.1  .94   3   13
north       NO   Val Shultz      4.5  .89   5   9
central     CT   Sheri Watson    5.7  .94   5   13

patience% awk '/east/ {print $5,$4}' data.file
4.0 Johnson
4.4 Beal
5.1 Nichols

patience% awk '/     [0-9]$/ {print $4 "\t" $8}' data.file
Craig     4
Shultz    9
```

Using *awk* to Print Pattern Matches

The first example prints only fields 5 and 4 from lines containing the characters 'east'.

The second example prints fields 4 and 8 from only those lines that end with a single digit. The *'$'* is the end of line anchor.

Notice the special characters which are used:

[ ]	set or range

Additional special characters include:

^	beginning of line anchor
.	any single character
*	0 or more occurrences of preceding character
$	end of line anchor

Mathematical Operations with *awk*

```
patience% awk '{print $5/$6, $1}' data.file
3.06122 northwest
5.46392 western
3.375 southwest
5.36842 southern
5.71429 southeast
5.2381 eastern
5.42553 northeast
5.05618 north
6.06383 central
```

```
patience% awk '{print $2, "Real Margin " , $5*(1000/$6)}'\
data.file
NW Real Margin 3061.22
WE Real Margin 5463.92
SW Real Margin 3375
SO Real Margin 5368.42
SE Real Margin 5714.29
EA Real Margin 5238.1
NE Real Margin 5425.53
NO Real Margin 5056.18
CT Real Margin 6063.83
```

```
patience% grep '^s' data.file | awk '{print $5/$6, $1}'
3.375 southwest
5.36842 southern
5.71429 southeast
```

Mathematical Operations with *awk*

Operators:

```
>
<
/
+
-
*
~
```

Notice that in the second example *awk* interprets the characters enclosed in double quotes as a text string to be printed as part of the output.

The third example illustrates combining *awk* with *grep* using a pipe to create a powerful filter.

Conditional Printing with *awk*

```
patience% awk '{if ($6 > .92) print $2, $6, $4}' data.file | sort
CT  .94  Watson
NE  .94  Nichols
NW  .98  Craig
SO  .95  Chin
WE  .97  Kelly

patience% awk '{if ($2 ~ /N./) print $2, $1}' data.file
NW  northwest
NE  northeast
NO  north

patience% awk '$2 !~ /N./' data.file
western     WE   Sharon Kelly   5.3  .97   5   23
southwest   SW   Chris Foster   2.7  .8    2   18
southern    SO   May Chin       5.1  .95   4   15
southeast   SE   Derek Johnson  4.0  .7    4   17
eastern     EA   Susan Beal     4.4  .84   5   20
central     CT   Sheri Watson   5.7  .94   5   13
```

Conditional Printing with *awk*

Notice that in the third example no command was specified. In this case, *awk* prints out the entire line.

awk Variables

```
patience% awk '{print $0 RS}' data.file
northwest  NW  Joel Craig      3.0  .98   3  4
western    WE  Sharon Kelly    5.3  .97   5  23
southwest  SW  Chris Foster    2.7  .8    2  18
southern   SO  May Chin        5.1  .95   4  15
southeast  SE  Derek Johnson   4.0  .7    4  17
eastern    EA  Susan Beal      4.4  .84   5  20
northeast  NE  TJ Nichols      5.1  .94   3  13
north      NO  Val Shultz      4.5  .89   5  9
central    CT  Sheri Watson    5.7  .94   5  13
```

```
patience% awk '{if ($6 > .92) print NR, $2, $6, $4}' data.file | sort +2n
7 NE .94 Nichols
9 CT .94 Watson
4 SO .95 Chin
2 WE .97 Kelly
1 NW .98 Craig
```

```
patience% awk 'BEGIN {OFS = "/"}{print $1,$2,$3,$4,$5,$6,$7,$8}' data.file
northwest/NW/Joel/Craig/3.0/.98/3/4
western/WE/Sharon/Kelly/5.3/.97/5/23
southwest/SW/Chris/Foster/2.7/.8/2/18
southern/SO/May/Chin/5.1/.95/4/15
southeast/SE/Derek/Johnson/4.0/.7/4/17
eastern/EA/Susan/Beal/4.4/.84/5/20
northeast/NE/TJ/Nichols/5.1/.94/3/13
north/NO/Val/Shultz/4.5/.89/5/9
central/CT/Sheri/Watson/5.7/.94/5/13
```

awk Variables

awk variables:

FILENAME	filename
FS	input field separator
RS	input record (line) separator
OFS	output field separator
ORS	output record (line) separator
NF	number of fields in record (line)
NR	number of current record (line number)

Writing *awk* Reports

```
patience% awk '{print $2, $5; total += $5} END {print "total = " total} ' data
  : :
  : :
NO 4.5
CT 5.7
total = 39.8

patience% awk 'BEGIN {OFS = "\t"; print "==== report ====" RS}{print $2,
 total += $5} END {print "total = " total} ' data.file
==== report ====

NW      3.0
 :      :
 :      :
NO      4.5
CT      5.7
total = 39.8
```

Writing *awk* Reports

awk Programming

```
patience% cat awk.sc
BEGIN {OFS = "\t"; print "====== report =======" RS}
   {
   print $2, $5, $5 * $6
   total += $5
   margin_t += $5 * $6; margin_cnt ++
   }
END {print  "----------------------" RS "total = " total, margin_t /
margin_cnt}

patience% awk -f awk.sc data.file
====== report =======

NW      3.0     2.94
 :  :   :
 :  :   :
NO      4.5     4.005
CT      5.7     5.358
----------------------
total = 39.8    3.971
```

awk Programming

awk commands do not have to be enclosed in single quotes when they are part of a script.

As with *sed*, the *–f* option indicates that *awk* is to take its input from a file and not from the command line.

awk Programming

A shell script, *dir*, which uses *awk*:

```
#! /bin/csh –f
# The objective of the program dir is to
# display the output of ls in a format
# similar to VMS or MS–DOS
#
#
ls –lA $argv | awk '{ printf ("%–14s%8s  %s %–2s %s\n", $8,
$4, $5, $6, $7) }'
```

awk Programming

The *"%s"* notation means to print the argument as a string. If a number is present – e.g., *"%14s"* – it means to display the string in a field at least 14 characters wide, blank filling as needed. If the number is positive (and the length of the string is less than the number), the string is right justified in the field.

A negative number indicates that the string should be left justified in the field. Hence, *"%–14s%8s"* (which are associated here with *$8* – the file name, and *$4* – the file size in bytes) means, "Display the file name, left justified, in a field 14 characters wide and the file size, right justified, in a field 8 characters wide."

If either string is greater than its allotted size, it overflows its field and disturbs the alignment of the output. If users have files with really long names or gross quantities of data, they might want to use larger values in order to avoid overflowing fields.

awk Programming: Counting Comments

```
patience% awk –f commentcounter1.awk *.[ch]
108 comments
```

```
patience% cat commentcounter1.awk
# comment counter for C
 /\*\// { count++ }
 END { printf ("%d comments\n", count) }
```

awk Programming: Counting Comments

It searches each line in the file using the regular expression:

```
/\*\//
```

This is less complicated than it appears. What it is actually looking for is the string "*/". However, since both "*" and "/" are special characters to awk, they have to be escaped by preceding each of them with "\".

Although it is not a recommended coding practice, it is legal in C to write:

```
/*  this is a comment
/*  that continues over
/*  several lines
*/
```

so we search for the end–of–comment delimiter rather than the start–of–comment delimiter.

awk Programming: Counting Comments (con't.)

patience% **awk –f commentcounter2.awk *.[ch]**
105 comments

patience% **cat commentcounter2.awk**

```
# smarter comment counter for C
/\*\//  { count++ }
/".*\*\/.*"/ { count-- }
END { printf ("%d comments\n", count) }
```

awk Programming: Counting Comments (con't.)

There is one small problem with our first comment counter. It is legal in C to say something like:

```
char *c = "comment terminator = */ ";
```

and our counter will erroneously count this as a comment.

One way to minimize the effects of this kind of error is to subtract one from our count every time an end-of-comment delimiter is found in a quoted string.

The regular expression in the second line means:

Search for an occurrence of a double quote	(")
followed by zero or more characters of any kind	(.*)
followed by "*/"	(*\/)
followed by zero or more characters of any kind	(.*)
followed by a double quote	(\")

Note: Since *awk* only searches for the first occurrence of a pattern on each line, none of these scripts will successfully count multiple comments on a single line.

awk Programming: Counting Comments a Fancier Way

```
patience% cat countcode.awk
/^[    ]*\/\*.*\*\/[    ]*$/ { comments++ }
/^[    ]*$/ { empties++ }
END    { code = NR – (comments + empties);
        printf("%5d code     (%5.2f%%)\n",
        code,    100.00 * code     / NR);
        printf("%5d empty    (%5.2f%%)\n",
        empties,  100.00 * empties  / NR);
        printf("%5d comments (%5.2f%%)\n",
        comments, 100.00 * comments / NR);
        printf("------       ----------\n");
        printf("%5d lines    (100.00%%)\n", NR);
}
```

```
patience% awk –f countcode.awk *.[ch]
 1061  code         (76.39%)
  163  empty        (11.74%)
  165  comments     (11.88%)
------              --------
 1389  lines        (100.00%)
```

awk Programming: Counting Comments a Fancier Way

Building on earlier *awk* scripts we have created a new script that computes percentages of types of lines in C source code.

awk syntax is quite liberal; it can often infer the ends of statements even without semicolons. *awk* also has a number of built-in variables, including "*NR*" which is the number of lines (input records) read so far. By the time the special pattern "*END*" is satisfied, the entire input has been processed, hence *NR* contains the total number of lines of input.

awk Programming: *for* Loops and Arrays

```
patience% awk '{ for (c = 1; c <= NF; c++) print $c}' data.file
northwest
NW
Joel
  :
  :
  :
9.4
5
13

patience% awk '{ a[NR] = $0 } \
                END { for (c = NR; c > 0 ; c--) print a[c] }' \
                data.file

central     CT   Sheri Watson    5.7  .94   5   13
north       NO   Val Shultz      4.5  .89   5   9
northeast   NE   TJ Nichols      5.1  .94   3   13
eastern     EA   Susan Beal      4.4  .84   5   20
southeast   SE   Derek Johnson   4.0  .7    4   17
southern    SO   May Chin        5.1  .95   4   15
southwest   SW   Chris Foster    2.7  .8    2   18
western     WE   Sharon Kelly    5.3  .97   5   23
northwest   NW   Joel Craig      3.0  .98   3   4
```

awk Programming: *for* Loops and Arrays

In general, *awk* uses the same *for* loop syntax as the C language. The parentheses following the *for* hold three elements separated by semicolons: an initialization statement; a condition; and a re–initialization statement. Like C, the loop continues executing as long as the condition is true.

The first example on the page shows how to use *awk* to break a file with multiple words per record into single word records. In previous examples we used $1, $2, $3, et cetera, to refer to various fields in a record. Here, we set a loop counter, *c*, to each of the values in the range 1 through NF, the number of fields in the current record. In the print statement the dollar sign before the *c* means the same thing as it did in front of a number, so each field from the original file is printed as a separate record.

The second program is a little more complicated. Here we see our first example of *awk* arrays. The first line of the program has no pattern, so it's action is applied to every record in the file. In this case, all we are doing is storing each line into an array location whose subscript is the number of the line in the source file (NR). Once all the lines have been read from the file, the *for* loop scans the array from end to beginning, printing out the lines in reverse order.

awk Programming: *for* Loops and Arrays (con't.)

```
patience% cat cats
lions
tigers
leopards
tigers
leopards
leopards
tigers
leopards
```

```
patience% awk '{ a[$1]++ } \
                END { for (c in a) print c, a[c] }' cats
tigers 3
leopards 4
lions 1
```

awk Programming: *for* Loops and Arrays (con't.)

This example shown here is a simplified version of a program that will count the occurrences of each of the words in a file. In order to simplify the program to show only its underlying algorithm, our data file has just one word per record.

Our first array example used an array in which every element was created and filled in order. It is not necessary to fill every element, nor is it necessary to create or reference elements in any particular order. One of the most remarkable things about *awk* is that it can use strings as array subscripts. As the main body of our program we have a single action.

The contents of the first field are used as a subscript into the array *a*. As the program starts to run, the word *lions* is read from the data file. Array element *a[lions]* is created. (Remember that variables are automatically allocated and initialized to zero the first time they are referenced.) Once the variable has been created, the "++" operator increments it, so *a[lions]* is set to one.

The next word read is *tigers*, and *a[tigers]* is set to one. The same thing happens with *leopards*. When the second occurrence of *tigers* is found, the *tigers* element already exists in the array, so *awk* just increments the value stored there. This process continues until all the words are read from the file.

In the action for the END pattern, we see a new syntax for the for loop. The *in* operator indicates that the argument following it is the name of the array. Now *c* is set to each subscript in the array in turn. Thus, the print statement shows the subscript (*c*) and then the value associated with that location in the array (*a[c]*).

Note: In order to support strings as subscripts, *awk* uses a special storage technique which often causes the internal order for the elements to be quite different from what you might expect.

Module 5
Environment Files

Objectives: Upon completion of this module, the student should be able to:

- State and identify the role of the shell in SunOS.
- Use and maintain the shell setup files: *.cshrc* and *.login.*
- View and modify the execution search path variable: *path.*
- Create customized command-line prompts with the variable *prompt.*
- Identify and describe the contents of additional environment files: *.sunview, rootmenu, .exrc,* and *.mailrc.*

Evaluation:

Perform Lab 4 to 90% proficiency.

Reference information:

- Appendix A, *Special Shell Characters.*
- *Setting Up Your SunOS Environment: Beginner's Guide* (p/n 800-1704-10).
- *Doing More with SunOS: Beginner's Guide* (p/n 800-1710-10), Chapter 4.
- *Mail and Messages: Beginner's Guide* (p/n 800-1709-10), Appendix A.
- *SunView 1 Beginner's Guide* (p/n 800-1706-10), Chapter 10.
- *The UNIX System: A Sun Technical Report,* Sun Microsystems Inc. (p/n 800-1419-02).

The Role of the C Shell 

- The shell's role:

  To *receive* commands from the keyboard (or from files) and translate them into system calls to the kernel.

- A Software Utility

  A *shell* is a software utility which interprets commands for execution by the kernel.

The Role of the C Shell

When you log into your system, a program called a *shell* is automatically invoked. The shell is the most visible interface between a user and the operating system kernel. The shell's primary role is to receive commands from the keyboard (or from files) and translate them into system calls to the kernel. The shell is both the personality of the operating system and a powerful tool for gluing together utilities into quick applications. In effect, it's "wrapped around" the *kernel*; hence the name *shell*.

Which Shell?

A variety of shells are available, with interaction styles ranging from rudimentary to elaborate, from command–driven to menu–style. The focus of this module is on the shell called the *C Shell* because its syntax and semantics are familiar due to their resemblance to C, and because it is the most commonly used login shell for SunOS.

A Shell is Just Another Utility Program

A *shell* is a software utility which interprets commands for execution by the kernel. In terms of its permissions and resources, it's no different from any of the hundreds of other utility programs available under SunOS. You can write and use your own shell if you wish; this has been done at many sites.

The Login Shell

Which shell you get when you login is determined by the system administrator in the */etc/passwd* file. */etc/passwd* contains a unique entry for each user which includes the name of that user's login shell. Most commonly, this is the C Shell.

The Role of the C Shell 

- Sub–Shells
- C Shell Starting and Ending Activity
- One Shell, Two Roles

The Role of the C Shell 

Sub-Shells

SunOS is a multitasking operating system. That is, it can run many processes at the same time. Any user can run many processes at the same time, and any of these can be shells. The "parent" process for any given user is typically that user's login shell, and any shells spawned as offspring thereof are termed sub-shells.

It is common for sub-shells to be spawned. For example, each time a new shelltool is opened, a sub-shell process is started. A sub-shell is also started by the command *csh* typed or implied (e.g. *sunview*) at the command line.

C Shell Starting and Ending Activity

Whether it's a login shell or a sub-shell, the first thing a C Shell process does upon invocation is to look in the home directory for a file called *.cshrc* and execute the commands it finds there before looking for further commands from the terminal. The main difference between the behavior of the C Shell when used as a login shell and when used as a sub-shell is that a login C Shell also executes a file called *.login* (if it exists) upon user login, and a file called *.logout* (if it exists) when the user logs out. These files, too, normally reside in the user's home directory. The command *exit* or <CTRL-D> is normally used to terminate a sub-shell and logout to terminate a login shell.

One Shell, Two Roles

In addition to processing user commands, the C Shell provides a programming language interface with loops, variables, and I/O. Programs written for the shells are called *scripts* because they are simply sequences of user commands stored in a file. Scripts are generally longer and more likely to incorporate variables and control structures than are command sequences typed directly by a user at a terminal. However, every programming language feature of the C Shell is, in fact, available at the command line.

Starting up the Shell

```
patience login: student1
password:
You don't have to think too hard
when you talk to teachers.
-- J.D. Salinger
Tue Jun 21 11:06:12 PDT 1988
/home/sunray/student1

patience:/home/sunray/student1 1% ps -x
  PID  TT  STAT   TIME  COMMAND
15388  p1  S      0:00  -bin/csh (csh)
15396  p1  R      0:01  ps -x

patience:/home/sunray/student1 2% csh
Perseverance furthers.
Tue Jun 21 11:07:12 PDT 1988
/home/sunray/student1

patience:/home/sunray/student1 1% ps -x
   PID  TTSTATTIME COMMAND
15388 p1  S      0:00  -csh (csh)
15397 p1  I      0:00  -sh (csh)
15401 p1  R      0:01  ps -x

patience:/home/sunray/student1 2% logout
Not login shell.

patience:/home/sunray/student1 3% exit
patience:/home/sunray/student1 4%  patience%:/home/sunray/student1 3%
```

Starting up the Shell

Unlike many other systems, the SunOS shell is not a command line interpreter built into the operating system. It is a separate program which reads commands from a keyboard or from a file and interprets those commands for the kernel. Hence, different command languages or styles of interaction can be implemented by substituting shells.

"Student1" logs into the machine named "*patience.*" Student1 is currently in his/her home directory and a C Shell process is started to interpret his/her commands. Before it starts reading commands from student1's keyboard, it reads them from two startup files, named *.cshrc* and *.login*, in that order. Note that if a C Shell is invoked by any process other than the login process, it would not be a "login shell" and would ignore the *.login* file.

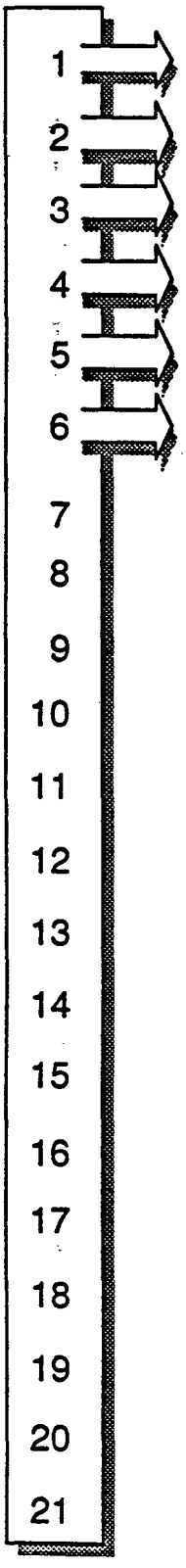
The *.login* file is useful for the automatic execution of setup commands at login time only. In this example, *.cshrc* commands produce the */home/games/fortune* output (the J. D. Salinger quip), the date, and the name of the present working directory, which are displayed before the first prompt. Finally, a prompt appears and the C Shell waits to read student1's keystrokes at the terminal.

Student1's first command is *ps –x.* The C Shell looks for an executable program called *ps.* When the program is found, a new process is created and the flag *–x* is passed to *ps.* This displays the shell process (#15388 in the example.)

Student1 types *csh.* The C Shell looks for an executable program called *csh.* When the program is found, a new process is created. We look at the processes again and discover there are now two shells running. The process–id is higher because a number of other processes were created in the interim, including those started by the C Shell executing the commands in *.cshrc.*

An attempt to logout of the second shell fails. Student1 leaves the second shell by typing *exit.*

A C Shell Setup File: *.cshrc*



```
1  # also search directories containing local commands
2  set path = (. ~ ~/bin /home/local /home/ucb /home/bin /bin )
3  alias cpi 'cp –i'
4  alias mvi 'mv –i'
5  alias rmi 'rm –i'
6  alias cd 'cd \!*; echo $cwd; set prompt="'hostname':'pwd' \! % "'
7
8      # the remaining setup is for interactive shells only
9      if ($?USER == 0 || $?prompt == 0) exit
10     set history =20
11     set ignoreeof
12     set prompt="'hostname':'pwd' \!% "
13     alias type more
14     alias dir  ls
15     alias help man
16     alias ff   'find . –name \!:1 –print'
17     alias psw  'ps –aux | sort +0b | more'
18     alias psg  'ps –aux | grep \!:1 | grep –v grep'
19     /home/games/fortune
20     date
21     pwd
```

A C Shell Setup File: *.cshrc*

When the shell parses a command it first looks for the command in a user–alterable list of *aliases* for a substitute of what was input. If the shell does not find an alias, it will next look in its list of *built–ins*. Built–ins are commands executed by the shell itself, including *cd, echo, if, switch* and so on. This means that you should not give your executable programs names such as *cd, echo, if, switch!* And, if the command is still not found, it looks along a "path" of directories for an executable file by that name.

The first command after the comment assigns to *path* a series of directories to be searched for executables. The C Shell assigns values to variables with the syntax:

set VARIABLE = name

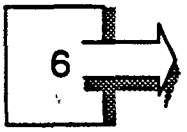
set makes an assignment for a local variable in the current shell as opposed to *setenv* which makes assignments for environmental variables which are utilized by programs, possibly outside of the current shell. Note that the syntax for *set* is different than the syntax for the environment variables *(setenv VARIABLE name)* which uses no '='.

A "path" is an array, which, in the C shell, is a list of words separated by spaces and enclosed in parentheses. This means that the first line is actually setting path[1] to '.' ("dot" is the current directory), path[2] to '~' ("tilde" is the user's home directory), and path[3] to ~/bin. The search for your executable continues in order through the entire array of directories in path. If you create an executable called *cat* and keep it in your personal *bin* directory, it will be executed when you type *cat* instead of the standard utility.

The C Shell has a lookup table, called a hash table, which is a table of contents for the executables contained in the directories specified in your path. This avoids the delay of actually looking through the path directories for executables. To update the hash table after creating or modifying an executable, or adding a new directory to your path, you must run the command *rehash* or the program will not be found and executed by the C Shell.

The next four lines create *aliases* (new names) for some SunOS commands. Someone accustomed to the command *dir* can *alias dir* to *ls*. In this way, it is possible to tailor the shell environment to the individual user.

A C Shell Setup File: *.cshrc*



```
6 alias cd 'cd \!*; echo $cwd; set prompt="'hostname':'pwd' \! % "'
```

patience:/home/sunray/student1 8% **echo $cwd**
/home/sunray/student1

patience:/home/sunray/student1 9% **set prompt="Yes, Boss\\! "**

Yes, Boss! **set prompt="Yes, Boss\\! \! >"**

Yes, Boss! 11> **hostname**
patience

Yes, Boss! 12> **pwd**
/home/sunray/student1

Yes, Boss! 13> **set prompt="'hostname':'pwd' \! > "**

patience:/home/sunray/student1 14>

A C Shell Setup File: *.cshrc* (con't.)

The fourth *alias* does more than just a simple substitution. The command *cd* (change directory) is aliased to a long line of commands which alter the prompt every time the user changes directories. The first command in this line uses history substitution. On the command line, you can use '!' to refer to previous commands. The '\' character defers history substitution from the time of aliasing to the time of execution. '\!*' substitutes all the arguments following the *cd* command whenever the user types it. It is replaced by all the arguments given on the command line. (Other modifiers of '!' such as '^' for the first argument and '$' for the last argument can take the place of the '*'). The '$' before cwd indicates that cwd is a variable. The variable *$cwd* (current working directory) is echoed after changing directories. The *set prompt* command inserts into the prompt the hostname and the pathname of the new directory.

One of the actions of the *alias* is to alter the prompt itself. You can alter the prompt at any time with the *set* command. The '\!' in the command: *set prompt="Yes, boss\!"* refers to the history number of the current command. In order to get a literal exclamation point, it must be quoted to escape its special meaning. Backslash is an escape character. So, in order to print the exclamation point, the command should be: *set prompt="Yes, boss\\!"*. The second *set prompt* command illustrates how to print both the exclamation point and the current command number.

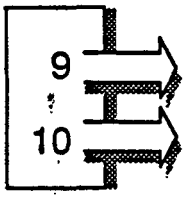
In the *alias*, the idea is not to simply replace the prompt with a fixed string but with the current host and directory. Typing *hostname* at the terminal displays the name assigned to your machine.

Typing *pwd* at the terminal gives the current working directory. The backquotes substitute the results of the backquoted command on the command line for the command itself. Thus,

'hostname'

is replaced by the name of the current host (patience), the colon and blank are preserved unchanged in the string, and 'pwd' is replaced by the full name of the current working directory. The '\!' with no modifier is replaced with the command history number and the '>' and blank are unchanged.

A C Shell Setup File: *.cshrc* (con't.)



```
 9 if ($?USER == 0 || $?prompt == 0) exit
10 set history =20
```

```
patience:/home/sunray/student1 14> history
2 csh
.
.
.
6 sleep 5 ; echo Yes, Boss!
7 cat .cshrc
8 echo $cwd $path[3]
9 set prompt="Yes, Boss!"
10 set prompt="Yes, Boss! \!>"
11 hostname
12 pwd
13 set prompt="'hostname':'pwd' \!>"
14 history

patience:/home/sunray/student1 15> set history=3

patience:/home/sunray/student1 16> history
14 history
15 set history=3
16 history
```

A C Shell Setup File: *.cshrc* (con't.)

The next command line exits the script when the contents of the parentheses evaluate to true. The '$' indicates that the following string is a shell variable, '$?*var*' evaluates to true if that variable has been set. Thus, as in C, the expression evaluates true if either the variable *USER* or the variable *prompt* have *not* been set. This is the case when the shell is not interactive. That is, it's not acting as a command line interpreter for a user but instead is executing a script. A shell that reads its commands from a script file instead of from a keyboard is not interactive and has no prompt. This test enables the script to determine whether the shell is interactive.

The complete list of preset shell variables and their meanings is given in the *csh* man page. The variable *user* is simply copied from the environment as are *home* (the home directory), and *term* (the terminal type). Changes to these shell variables are copied back to the environment. Other variables, such as *prompt* and *cwd* are set on initialization to some default value unless altered as in the following lines.

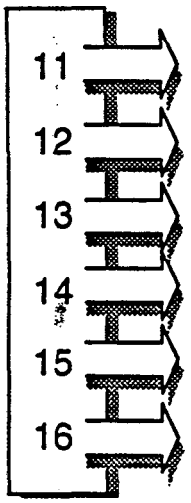
Review of History

The variable *history* contains the number of command lines to be saved for history substitution and *savehist* contains the number of such lines of history to be saved to disk when you log out. *history* typed alone on the command line displays the recent commands given to the shell (it's a built-in).

Next, *history* is set to a smaller number, three. Typing *history* now displays the shorter list of commands. "*!!* " re-executes the last command.

"*!14*" re-executes the fourteenth command. History command references can also be invoked by word designators (instead of numbers). The history command will then execute the most recent command which matches the word designator. For example, *!se* will re-execute the *set history = 3* command.

A C Shell Setup File: *.cshrc* (con't.)



```
11 set ignoreeof
12 set prompt="'hostname':'pwd' \!% "
13 alias type more
14 alias dir  ls
15 alias help man
16 alias ff   'find . -name \!:1 -print'
```

```
patience:/home/sunray/student1 17> ff .login
./.login
patience:/home/sunray/student1 18>
```

A C Shell Setup File: *.cshrc* (con't.)

Ignoreeof protects the shell scripts from recognizing *<CTRL–D>* as *logout* when it is typed at the terminal. This is a particularly handy option if you are logged in over a modem where a <CTRL–D> generated by line noise or typed inadvertently could cause you to exit the shell and drop the phone line.

The next line sets up the prompt the same way as our *cd* alias. Without this line, the prompt would default to *hostname*% until after the first *cd*. The next three aliases are for people more comfortable with non–UNIX commands. The ff alias remedies the notoriously difficult syntax of the *find* utility.

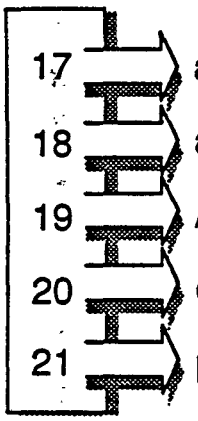
Find requires as its first argument the directory which is to serve as the base of the search tree. As usual, '.' (dot) indicates that this should be the current directory. The flag *–name* signals that the next argument is the name of the sought file. In this case, it is supplied by the first argument given on the current line. *Find* displays the names of the files it finds only if given a *–print* flag. To use this alias, you would type:

ff file_name

to execute the command:

find . –name file_name –print

A C Shell Setup File: *.cshrc* (con't.)



```
17  alias psw  'ps -aux | sort +0b | more'
18  alias psg  'ps -aux | grep \!:1 | grep -v grep'
19  /home/games/fortune
20  date
21  pwd
```

```
patience:/home/sunray/student1 18> ps -aux
USER       PID  %CPU  %MEM    SZ   RSS  TT   STAT    TIME COMMAN
student1  3033  55.2   7.9   304   264  p2   R       0:00 ps -aux
```

```
patience:/home/sunray/student1 19> ps -aux | sort +0b | more
USER       PID  %CPU  %MEM    SZ   RSS  TT   STAT    TIME COMMAN
student1 17173  29.7   4.1   336   296  p1   R       0:00 ps -aux
student1 17174   2.7   1.2   104    72  p1   S       0:00 sort +0b
student1 17175   0.0   1.1    80    64  p1   S       0:00 more
.
.
.
---More---
```

```
patience:/home/sunray/student1 20> fortune
fortune: Command not found.
```

```
patience:/home/sunray/student1 21> /home/games/fortune
The computing field is always in need of new cliches.
        -- Alan Perlis
```

A C Shell Setup File: *.cshrc* (con't.)

The command lines shown here exemplify the use of pipes within scripts. A *pipe* is invoked by the character '|' and causes the output from the program on the left to be fed as input to the program on the right.

The first command in the series of pipes in the first line is *ps*. The *ps* command's *aux* flags indicate that you want to see a list of *all* the processes ('a'), including those not associated with any particular terminal ('x'), and want user *names* ('u') as well as ID numbers.

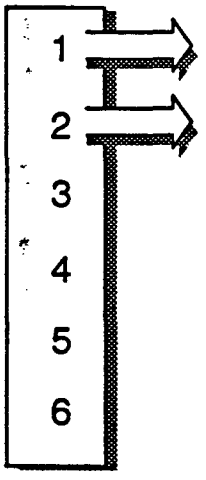
The second command in the pipeline is the utility *sort*. *Sort* copies its input to its output with the order of the data lines rearranged. Each line of input is considered a single record and blank characters are considered field separators. There are many different options to control how sorting is done. Here, the flag *+0b* is used to indicate that the sort should start ('+') with the first field ('0') and ignore leading blanks ('b'). The effect is to sort the *ps* output by user name. Finally, the results are *piped* to *more*.

The second *alias psg,* takes the entire list of processes and performs two *grep* operations. The first *grep* matches the first command line argument; if we type: *psg jane* , the effect would be to filter out all but jane's processes. The second *grep* is fed the results of the first. Its *–v* flag inverts the search (for the literal string *"grep"*) eliminating lines containing the word *grep* itself on them. We're not interested in the *grep* processes themselves.

Command number 20 fails, but it successfully executes (command number 21) when its full pathname is specified. Why? The directory */home/games* is not in the search path. It could also fail if the C Shell's hash table is out of date or if you lack permission to execute the command.

A C Shell Setup File: *.login*

patience:/home/sunray/student1 3% **cat .login**



```
1  # for vgrind
2  setenv TROFF ptroff
3  if ("'tty'" != "/dev/console") exit
4  echo -n "SunView? (^C to interrupt) "
5  sleep 5
6  sunview
```

patience:/home/sunray/student1 4% **printenv**
HOME=/home/sunray/student1
SHELL=/bin/csh
PATH=.:/home/sunray/student1:/home/sunray/student1/bin:/home/local:/home/ucb:/home/bin:/bin
TERM=sun
USER=student1
LOGNAME=student1
PWD=/home
TROFF=ptroff

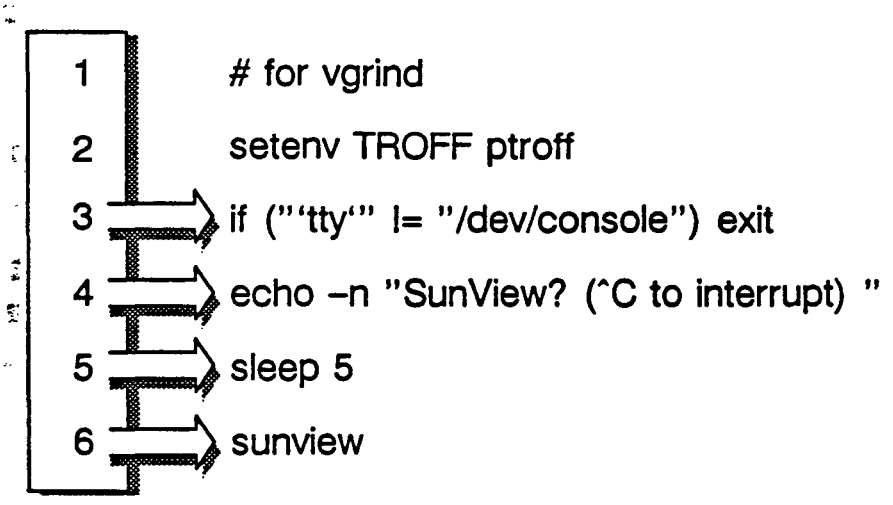
A C Shell Setup File: *.login*

The C Shell .login File

In this example the first line is a comment. Any lines which begin with '#' are ignored by the shell. The first command sets an environment variable TROFF to contain the string ptroff. The environment is an array of strings which is copied into the data space of every new process.

The command *setenv* creates a new variable in the environment or changes the value of an existing one. Typical of the way in which such variables are used, the variable TROFF is read by the program *vgrind* for name of the program that produces phototypeset output. The environment variable mechanism is a logical way to pass that sort of information. Because the *setenv* command is in *.login*, this variable will always be created automatically at login. You can display your current environment with the *printenv* command.

A C Shell Setup File: *.login* (con't.)



patience:/home/sunray/student1 5% **tty**
/dev/tty1

patience:/home/sunray/student1 6% **sleep 5;echo Yes, Boss\!**
Yes, Boss!

A C Shell Setup File: *.login* (con't.)

The next line has a test and looks much like a line of C:

if(string != string) exit

The C Shell has most of the operators available in C as well as most of C's control structures *(if, while, switch)*.

The command *tty* prints the pathname of the default I/O device for the shell. The backquotes around *tty (')* are command substitution. It replaces the command given in the script with the results of that command. In this case, the command tty is replaced by the output of the command tty. Then the two strings are compared as per the relational operator. As in C, such relational expressions have the value 1 if true and 0 if false.

The C Shell command *echo* prints to the terminal whatever arguments follow it on the command line. The *–n* flag suppresses a new–line in the echoed output. The *sleep* command waits the specified number of seconds and then continues processing. In the .login file, *sleep* gives you a little time to abort the processing of the file, which you would do by typing <CTRL–C>.

For the command line example, after waiting five seconds the system echoes the message on the right; "Yes, Boss!". The semicolon separates distinct commands sharing a single command line.

The last line of this *.login* script invokes *SunView. SunView* is a collection of programs that provide a graphical user interface based on the desktop metaphor. Since it requires a bitmapped screen, it can run only on the console. ASCII terminals connected by serial lines are not bitmapped. Thus, *.login* makes sure the user is at the console before running *SunView.*

The Startup File for SunView: *.sunview*

- The *~/.sunview* file contains the template for a customized work environment; it contains the locations of the open and closed frames.

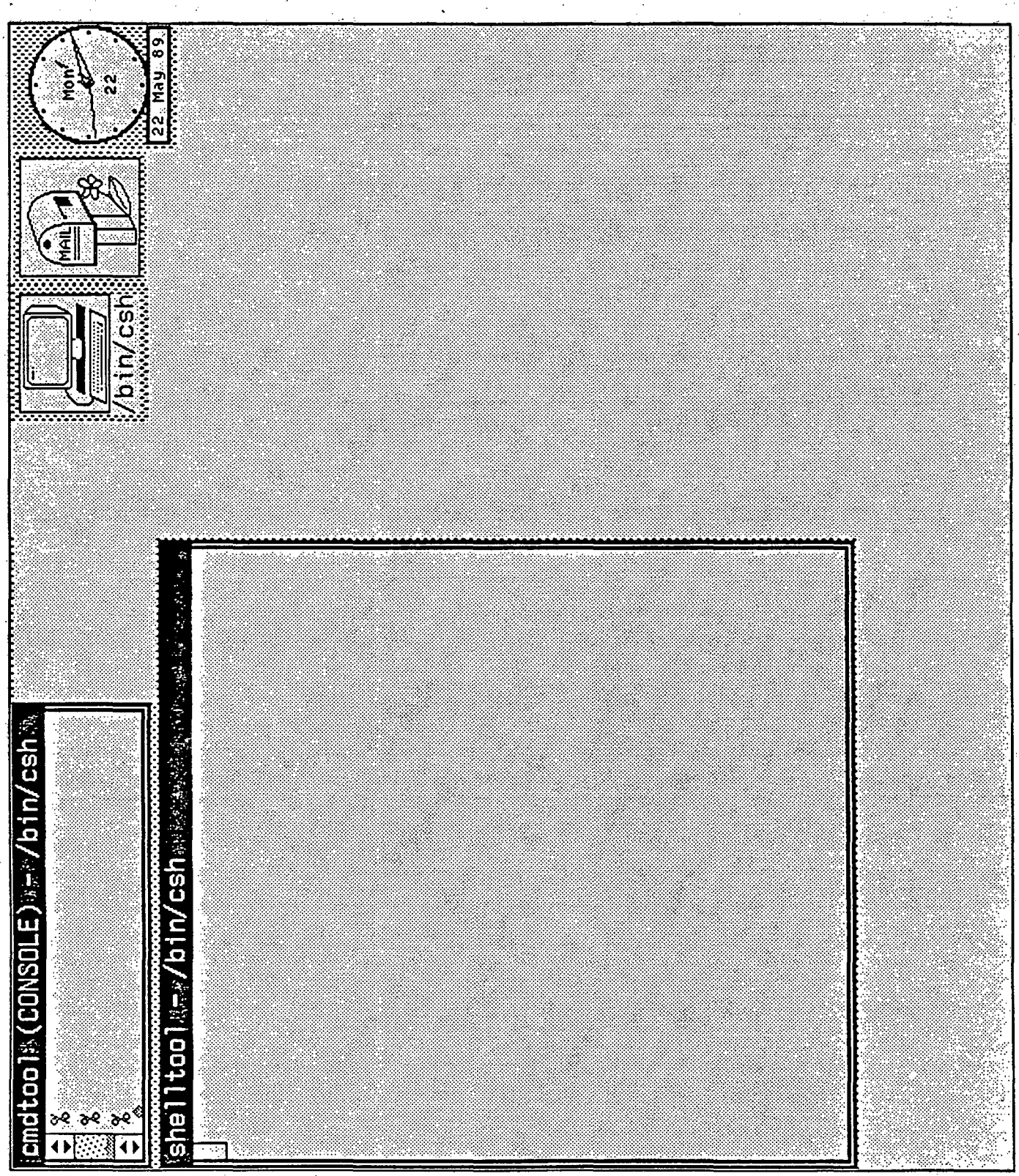
- Create this file using the 'Services⇒Save Layout' menu option or by using the *toolplaces* command.

- If this file does not exist, SunView will display the standard startup screen.

```
patience% cd
patience% cat .sunview

mailtool    -Wp 54   150  -Ws 670  720  -WP 1008   8  -Wi
cmdtool     -Wp  0     0  -Ws 435  125  -WP 1056   0  -C
clock       -Wp 244   85  -Ws 210  52   -WP 1088   8  -Wi  -f  -S  -d  dm
shelltool   -Wp 613  124  -Ws 539  776  -WP  916   8  -Wi
shelltool   -Wp  0   123  -Ws 613  777  -WP  832   8
```

- This *.sunview* file produces a screen layout similar to the one shown on the next page.

The Startup File for SunView: *.sunview*



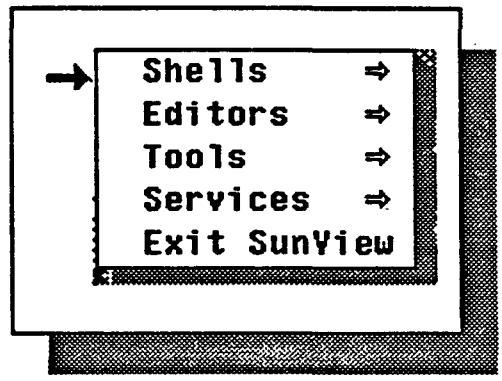
Customizing the SunView Menu: *rootmenu*

- The default root menu exists in */usr/lib/rootmenu*.
- Copy this file to your home directory (you may name it something different from *rootmenu*).
- Edit the file to suit personal tastes.
  - This file may not contain any blank lines.
  - The name of each menu item must be enclosed in double quotation marks.
  - Follow each menu item with the appropriate command.
- Specify the name of the new file to the Defaults Editor.
- Exit SunView.
- Restart SunView.

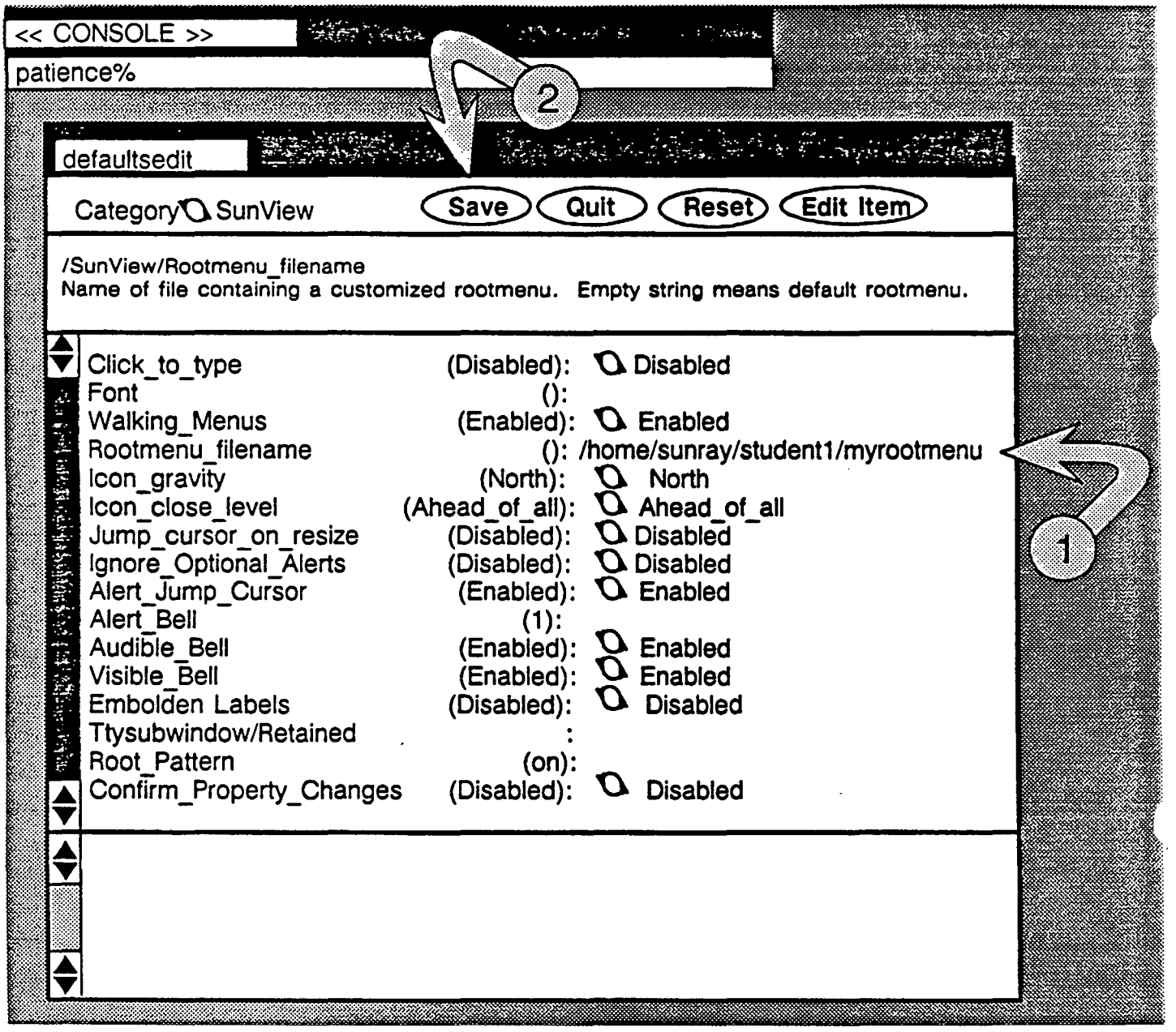
Customizing the SunView Menu: *rootmenu*

○ Part of the default root menu found in */usr/lib/rootmenu*.

```
#       @(#)rootmenu    10.13 88/03/07 SMI
#       sunview root menu
#
"Shells"                MENU
        "Command Tool"      cmdtool
        "Shell Tool"        shelltool
        "Graphics Tool"     gfxtool
        "Console"           cmdtool -C
"Shells"                END
"Editors"               MENU
        "Text Editor"       textedit
        "Defaults Editor"   defaultsedit
        "Icon Editor"       iconedit
        "Font Editor"       fontedit
"Editors"               END
"Tools"                 MENU
        "Mail Tool"         mailtool
        "Dbx (debug) Tool"  dbxtool
        "Performance Meter"     MENU
                "Percent CPU Used"          perfmeter -v cpu
                "Ethernet Packets"          perfmeter -v pkts
                "Swapped Jobs"              perfmeter -v swap
                "Disk Transfers"            perfmeter -v disk
        "Performance Meter"     END
        "Clock"                 MENU
                        </usr/include/images/clock.icon>       clock
                        </usr/include/images/clock.rom.icon>   clock -r
        "Clock"                 END
"Tools"                 END
. . . .
```

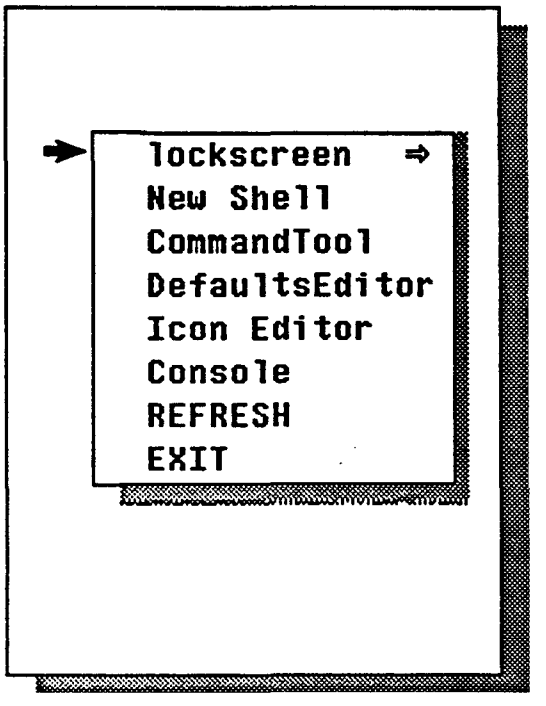


Customizing the SunView Menu: *rootmenu*



Customizing the SunView Menu: *rootmenu*

```
patience% cd
patience% cat myrootmenu
#
#     My root menu: /home/sunray/student1/myrootmenu
#     July 1, 1989
#
"lockscreen"        lockscreen
"New Shell"         shelltool
"CommandTool"       cmdtool
"DefaultsEditor"    defaultsedit
"Icon Editor"       iconedit
"Console"           cmdtool -C
"REFRESH"           REFRESH
"EXIT"              EXIT
```



Startup File for *vi*: *.exrc*

- The *~/.exrc* file contains commands to initialize the *vi* editing environment. A sample file exists in */usr/lib/Exrc*.

- Once in a *vi* editing session, the *:set* command by itself will list the options currently in effect. To obtain a list of available options, use the *:set all* command.

- Any option may be set or reset during an editing session with the *:set* command. The format is:

  :set [no]option[=value]

- For options that require a value, note that there should be no spaces between the equal sign and the *option* or *value*.

- To disable an *option* during the edit session, prefix the *option* with "*no*".

- The *.exrc* file may not contain comment characters.

Startup File for *vi*: *.exrc*

```
patience% cd
patience% cat .exrc
set autoindent
set noignorecase
set nomesg
set nonumber
set report=0
set tabstop=8
set wrapmargin=5
patience% vi testfile
```

```
~
~
~
~
:set all
autoindent          open                              tabstop=8
autoprint           nooptimize                        taglength=0
noautowrite         paragraphs=IPLPPPQPP LIpplpipbp   tags=tags /usr/lib/tags
nobeautify          prompt                            tagstack
directory=/tmp      noreadonly                        term=sun
noedcompatible      redraw                            noterse
noerrorbells        remap                             timeout
hardtabs=8          report=0                          ttytype=sun
noignorecase        scroll=24                         warn
nolisp              sections=NHSHH HUnhsh             window=48
nolist              shell=/bin/csh                    wrapscan
magic               shiftwidth=8                      wrapmargin=5
mesg                noshowmatch                       nowriteany
nomodeline          noslowopen
nonumber            nosourceany
[Hit return to continue]
```

Customizing a Mail Environment: *.mailrc*

- The *~/.mailrc* file contains commands to initialize the *mail* and Mail Tool environments. A sample file exists in */usr/lib/Mailrc*.

- The mail environment may be initialized in two ways: by editing the *.mailrc* file in your home directory, or by using the Defaults Editor.

- If editing the *.mailrc* file, commands are of the format:

  set [no]option[=value]

- For options that require a value, note that there should be no spaces between the equal sign and the *option* or *value*.

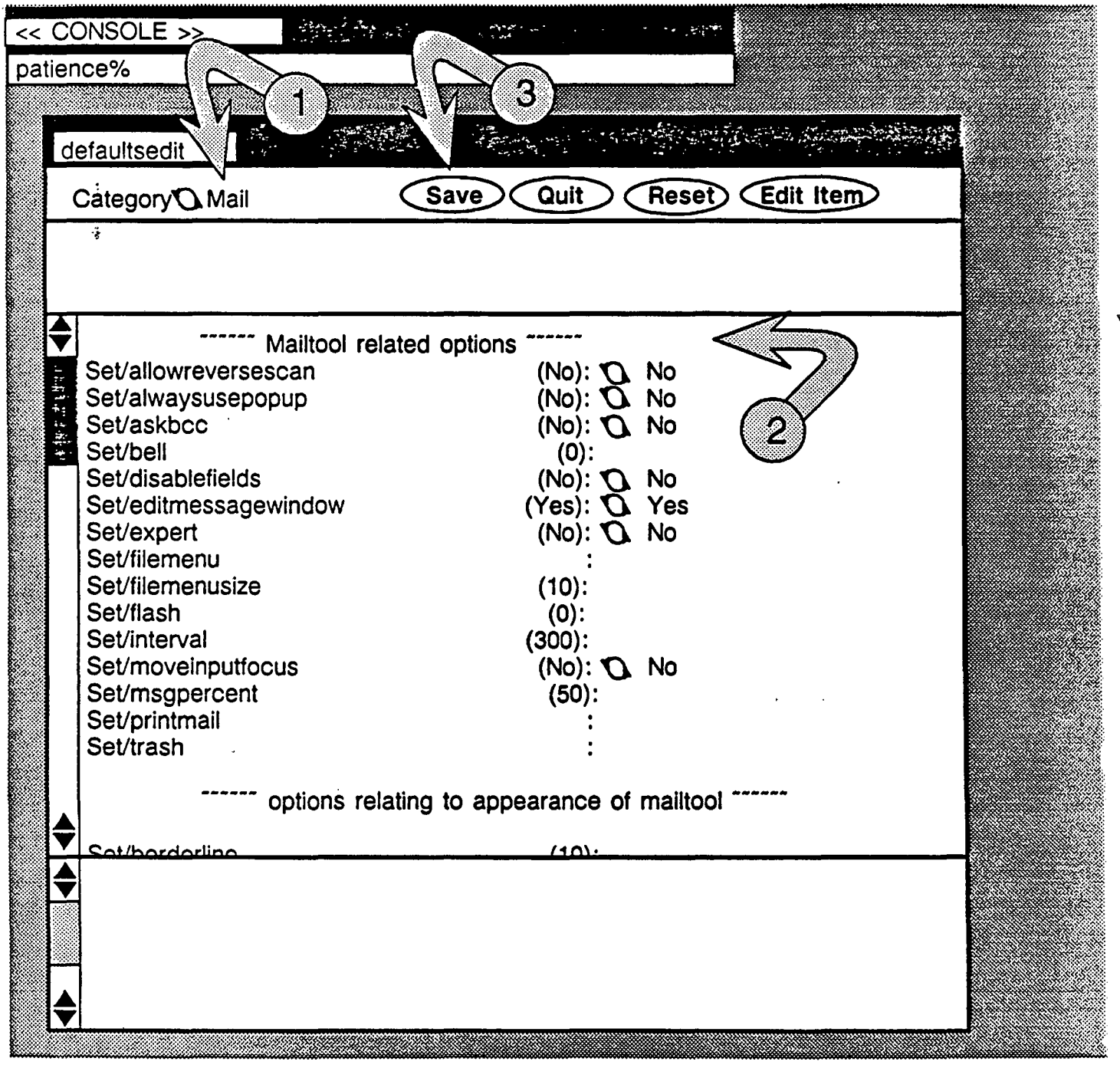
- To disable an *option*, either remove the line or prefix the *option* with "*no*".

- The Defaults Editor is easier to use because it is an interactive program and provides brief explanations for each item.

Customizing a Mail Environment: *.mailrc*

```
patience% cat /usr/lib/Mailrc

# @(#)Mailrc 1.3 88/02/08 SMI;

set alwaysignore
set askcc
set asksub
set autoprint
set cmd="lpr -p &"
set crt=15
set DEAD=~/dead.letter
set EDITOR=/usr/ucb/ex
set hold
set indentprefix="> "
set keepsave
set metoo
set PAGER="cat -s | more -22 -c"
set prompt="{Mail}& "
set record=~/.record
set SHELL=/bin/csh
set VISUAL=/usr/ucb/vi

ignore apparently-to date errors-to from id in-reply-to \
        message-id precedence received references remailed-date \
        remailed-from return-path sent-by status via
```

Customizing a Mail Environment: *.mailrc*



Customizing a Mail Environment: *.mailrc*

When using Defaults Editor to create and/or maintain your *~/.mailrc* file, first set the "Category" field to "Mail". This category is the seventh one after "SunView" (which is the first category). Next scroll through the available options and toggle those that are appropriate. The options are grouped as follows: "Mailtool related options", "options relating to the appearance of mailtool", "options affecting both Mailtool and Mail", "options that affect only the program 'Mail'". For more information on *mail* and *mailtool*, refer to *Mail and Messages: Beginner's Guide*. For information on Defaults Editor refer to *SunView 1 Beginner's Guide.*

Module 6
Advanced C Shell

Objectives: Upon completion of this module, the student should be able to:

- State and describe the purpose of a C shell program.
- Set and use local and global shell variables, with the commands: *set*, *echo*, *setenv*, *printenv*.
- Design a C shell program using the *foreach–end* construct.
- Design a C shell program using the *while–end* construct.
- Utilize *echo* and command line arguments in shell scripts.
- Construct and perform software installations using shell archives which contain *here documents*.

Evaluation:

Perform Lab 4 to 90% proficiency.

Reference information:

- Appendix A, *Special Shell Characters*.
- Appendix C, *Install*.
- *Doing More with SunOS: Beginner's Guide* (p/n 800–1710–10), Chapter 4 and Appendices B and D.
- *The UNIX System: A Sun Technical Report*, Sun Microsystems Inc. (p/n 800–1419–02).

What is Shell Programming?

- A shell program is a text file that contains a sequence of commands.

- Benefits:
  - Simplifies repetitive tasks
  - Simplifies execution of syntactically sensitive commands
  - Reusable

- Features:
  - Interactive capabilities
  - Constructs for:
    - Making decisions
    - Looping
    - Handling program arguments
    - Performing arithmetic operations
    - Evaluating expressions
  - Interrupt and error handling
  - Status reporting
  - May be called from programs and vice versa

What is Shell Programming?

We have seen in the previous module that C shell scripts may be used to accomplish a variety of tasks, such as setting variables (local and environmental), defining aliases, and using the history utility. Shell scripts also provide programming constructs for more complex tasks.

The Shell Supports SunOS's Building-Block Approach

The underlying philosophy of software development in SunOS is to write and use a multiplicity of small, clearly-defined program modules. Each can be considered a reusable tool which performs a single function and can be combined with other tools in shell scripts. Most SunOS utilities are written to be invoked by scripts as well as interactively by users. By combining utilities within scripts, modest amounts of programming can yield powerful applications.

Executing a C Shell Script

patience% **source scriptfilename**

OR

patience% **csh scriptfilename**

OR

patience% **chmod +x scriptfilename**

patience% **scriptfilename**

Executing a C Shell Script

Running a Shell Script

The commands contained in the *.cshrc* and *.login* files are the same as commands you might type at the terminal. Files which contain such commands are called scripts; hence both *.cshrc* and *.login* are scripts. Use *source* to execute the commands in a script for the current shell. The two setup files, *.login* and *.cshrc*, are two such files.

The remaining two methods of executing a shell script do so in a sub–shell. It is important to note that the default sub–shell invoked for the execution of scripts is in fact not the C Shell; it's a more rudimentary shell called the Bourne shell, for its author, Steven Bourne, formerly of Bell Labs. The differences between the C Shell and the Bourne shell are sufficient that scripts written for one rarely execute properly with the other.

When a shell script is executed as a command, the first line in the script determines which shell will be used to interpret it. A first line of,

```
#! /bin/sh
```

will cause the script to be executed by the Bourne shell. If the first line is,

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

then the shell will be executed by the C shell.

If the first line does not explicitly specify the shell, then another rule is applied to determine which shell should be used to interpret the script. The alternate rule is that if the first character in the script is a pound sign (#), the script will be executed by the C shell; otherwise, the Bourne shell will be used.

Shell Variables

- Creating and deleting regular (local) variables

  set flower = rose

  unset flower

- Creating and deleting environmental (global) variables

  setenv flower rose

  unsetenv flower

- Commonly used predefined shell variables

  argv

  status

- Setting variables to a string that is longer than one word

  set flower = "Josephs Coat"

  set flower = 'Josephs Coat'

  set flower = Josephs\ Coat

  set flower = (Josephs Coat)

Shell Variables

Shell Variables

The C Shell maintains two kinds of variables: regular variables which are local to that individual shell, and environment variables which are global to all the user's processes. Shell variables, whether local or global, can have only one data type: *character string.* Regular variables are set using the command *set*. Environment variables are set using the command *setenv*. Notice that one uses '=' and the other does not:

set flower = rose
setenv flower rose

To display the current environment variables, use the command *printenv*; for local variables the command *set* alone will do it. To delete an environment variable, use the command *unsetenv*. To delete a local variable, use the command *unset*.

The shell variable, *argv*, is used by the shell to keep track of (list) the arguments that were entered on the command line. The *status* shell variable contains the exit status returned by the most recent command. This is useful for error detection.

Accessing Shell Variables

- Performing variable substitution:

  echo The $flower is pretty.

  echo The ${flower}s are lovely this year.

- Obtaining the number of words in the string:

  patience% **echo $#flower**
  2

- Determining whether the variable has been set:

  patience% **echo $?flower**
  1
  patience% **echo $?haha**
  0

Accessing Shell Variables

A '$' prefixed to either kind of variable expands the variable. For example, *$flower* would be replaced with *rose*. Braces around a variable name isolate it from literal text. For example:

```
patience% echo The ${flower}s are lovely this year.
```

would yield:

```
The roses are lovely this year.
```

As shown earlier, a variable may be set to a longer string in several ways.

```
set flower = "Josephs Coat"
set flower = 'Josephs Coat'
set flower = Josephs\ Coat
set flower = (Josephs Coat)
```

However, the command *echo $flower[1]* produces *Josephs Coat* for the first three cases and produces *Josephs* for the last case. This is because parentheses are used in the last example. Remember that in the C shell, parentheses define an array.

The modifier '#' (yes, the same character as in comments, but with another meaning in this context) yields the number of words in the string:

```
patience% echo $#flower
2
```

The modifier '?' tells whether the variable has been set:

```
patience% echo $?flower
1
patience% echo $?haha
0
```

Accessing Shell Variables (con't.)

- Variable suffixes for parsing pathnames:

```
patience% set my_flower = ~/flowers/roses/climbing.beautiful
patience% echo $my_flower:e
beautiful
```

Accessing Shell Variables (con't.)

There are also a series of variable suffixes which provide an easy way to separate pathnames into different pieces:

h	provides the leading pathname
t	removes the leading pathname, providing the filename only
r	removes the filename extension after the "."
e	provides the filename extension only

These can be used on a variable like *my_flower* as follows:

```
patience% set my_flower = ~/flowers/roses/climbing.beautiful
patience% echo $my_flower:h
/home/sunray/student1/flowers/roses
patience% echo $my_flower:t
climbing.beautiful
patience% echo $my_flower:r
/home/sunray/student1/flowers/roses/climbing
patience%
```

There are other such modifiers for which your man page on the C Shell (*csh(1)*) is the best guide.

Accessing Command Line Arguments

Argument	Meaning
$#argv	Specifies the number of arguments
$argv[ ]*	Refers to all arguments supplied on command line
$argv	Refers to all arguments supplied on command line
*$**	Refers to all arguments supplied on command line
$0	Command or program name* *Note: $argv[0] is illegal; must use $0
$argv[n]	Argument number *n*, where *n* represents any argumen
$n	Argument number *n*, where *n* represents any argumen
$argv[n–n]	Refers to a range of arguments

Note on quoting mechanisms:

Double Quotes:	turn off the special meaning of all C shell metacharacters except single quote ('), back quote (`), backslash (\), and bang (!).
Single Quotes:	turn off the special meaning of all C shell metacharacters except backslash (\) and bang (!).
Backslash:	turns off the special meaning of any C shell metacharacter that immediately follows the backslash.

Accessing Command Line Arguments

Remember that the shell maintains a list of the command line arguments entered when invoking a shell script in the predefined shell variable, *argv*. The format for executing shell scripts with arguments is:

scriptfilename argument1 argument2 argument3. . .

A few examples:

```
patience% cd
patience% cat echo_arg
#!/bin/csh
#This shell script echoes all arguments entered on the command line.
#Usage:  echo_arg arg1 arg2 arg3. . .
echo $*
patience% echo_arg 1 2 3 4
1 2 3 4
patience% ls j*
No match.
patience% cat scopy
#!/bin/csh
#This simple copy program will accept one directory name and up to five file-
#names.  It will create the directory and copy the specified files to the new
#directory.
#Usage: scopy dirname filename filename filename filename filename
#!/bin/csh
mkdir $1
echo "Created directory $1"
cp $2 $3 $4 $5 $6 $1
echo "Copied files $2, $3, $4, $5, $6, to directory $1"
patience% scopy junk magazines memofile merger
Created directory junk
Copied files magazines, memofile, merger, , , to directory junk
patience% ls junk
magazines       memofile        merger
```

Shell Control Structures
(if–then–else–endif)

- *if* (condition) command

- *if* (condition) *then*:

```
if ($#argv < 2) then
   echo "usage: scopy dir_name file_name(s)"
   exit 1
endif
```

- *if–then–else–endif*:

```
if ($flower == "rose") then
   echo I like ${flower}s
else if ($flower == "snapdragon") then
   echo ${flower}s are okay\!
else
   echo I hate ${flower}s
endif
```

Shell Control Structures
(if–then–else–endif)

This statement is true if the variable *flower* is equal to "rose." Most C relational expressions are supported, including '==', '||', '&&', '<', '<=', '>' and '>='. As in C, such expressions have the value 1 when true and 0 when false. You may also test for the existing readability and writeability of files or test for the success or failure of a command's execution.

Example of an *if–then–else–endif:*

```
if ($flower == "rose") then
        echo I like ${flower}s
else if ($flower == "snapdragon") then
        echo ${flower}s are okay\!
else
        echo I hate ${flower}s
endif
```

Unlike in C, you do not have the freedom to place the *if* and *then* just anywhere. An *if,* followed by *the expression* and *then* must be on the same line. An *endif* must be on its own line.

Shell Control Structures
(foreach, repeat)

- *foreach* loop:

```
foreach flower (rose gladiolus sweetpea snapdragon)
    echo $flower
end
```

- *repeat*:

```
repeat 4 echo I like roses
```

Shell Control Structures
(foreach, repeat)

C Shell Control Structures

There are many types of control structures in the C Shell, most with syntax similar to C. Exceptions are the *foreach* and *repeat* loops.

foreach loops through the words in a list, assigns each in turn to a variable, and performs the commands within the loop.

```
patience% foreach flower (rose gladiolus sweetpea snapdragon)
? echo $flower
? end
rose
gladiolus
sweetpea
snapdragon
patience%
```

The *repeat* command is a very simple loop:

```
patience% repeat 4 echo I like roses
I like roses
I like roses
I like roses
I like roses
patience%
```

Shell Control Structures
(while loop*)*

- *while* loop:

```
while ( "$flower" != "rose")
   echo Wrong... I do not like ${flower}s. Try again:
   set flower=$<
end
```

Shell Control Structures
(while loop*)*

The *while* loop has a test at the top of the loop and executes commands within the loop body.

For example,

```
echo "What is my favorite flower?"
set flower=$<
while ("$flower" != "rose")
echo Wrong... I do not like ${flower}s. Try again:
set flower=$<
end
echo Right. I do like ${flower}s
```

is a simple guessing game. It loops until you answer "rose."

The expression '$<', used in the example above, is reading a user's input from the keyboard. Flower is set to the string input at the terminal.

Relational Expressions and Tests

- Relational Expressions:

```
$flower != "rose"
```

- Built–in Command Tests { }:

```
if ( { grep  –s  rose /home/sunray/student1/flowers } )  then
echo "I like roses too"
endif
```

- Built–in File Tests:

```
if (–flag filename) echo tested
```

Relational Expressions and Tests

Relational Statements

The C Shell's programming language also supports relational expressions. For example, the test:

$flower != "rose"

is true if the string input is not equal to "rose." Most C relational expressions are supported, including ==, ||, &&, <, <=, > and >=. As in C, such expressions are equal to the value 1 when true and 0 when false.

Built–in Tests

There are a number of very useful special tests built into the shell. Particularly handy is the simple ability to test for the success or failure of a long command by surrounding it with curly braces: "{}".

if ({ grep –s rose /home/sunray/student1/flowers }) echo "I like roses too"

If the *grep* command succeeds, the *echo* will be performed. Note that spaces are required between the curly braces the the command (i.e. between "{" and "g" and between "s" and "}").

File Tests

The C Shell also contains built–in tests on files.

The format is:

*if (–*flag filename*) echo tested*

The flags are:

flag	Description
–r	readable
–w	writable
–x	executable
–e	exists
–o	owned by the executing user
–z	zero size
–d	a directory
–f	a file

Case Study: C Shell Programming

```
#adduser skeleton
#create environment variables:
#       login_name
#       group_name
#       directory
#       user_id
#convert group_name to group_id
#add an entry to /etc/passwd
#make home directory and home/bin directories
#move to home directory
#copy template .cshrc, .profile, .login files
#set permissions
#change group membership
#change ownerships
```

Case Study: C Shell Programming

Shown here is a set of comments describing the steps performed by the *adduser* script. Normally, a system administrator would have to enter these commands by hand (the script is not a part of UNIX). *Adduser* is a procedure typically performed by a system administrator to create a new login account, thereby granting access by a new user to the system. A step–by–step analysis of this script comprises the remainder of this module.

Case Study: C Shell Programming

```
patience% setenv login_name student2

patience% setenv group_name guest

patience% setenv directory student2

patience% setenv user_id 110

patience% echo $login_name $group_name $directory $user_id
student2 guest student2 110

patience% grep "$group_name" /etc/group
guest:*:40:

patience% grep "$group_name" /etc/group | awk -F: '{print $3}'
40
```

Case Study: C Shell Programming

Complex scripts are usually developed from simplified skeletons and are tested a line at a time. A useful technique to use when developing scripts on a Sun workstation is to use two windows. One window can be used for editing the script, the other for testing commands before they are added to the script.

Another technique is to defer the development of the interactive portion of the script. This can be done by using environment variables instead of eliciting them from the user. The *setenv* command initializes the variables and the *echo* command verifies the results. This technique saves the developer from having to type the answers for each of the user inputs every time a test run is made.

The *grep* command returns the line from */etc/group* which has the contents of the variable *group_name*. We pipe that line to *awk* to filter out all but the group number. The *–F:* flag to *awk* declares the field separator to be ':' and *print $3* (print the third field only) is the action to be taken. The effect of this command is to replace the group *name* by the group ID number.

Case Study: C Shell Programming

```
patience% set gid='grep "$group_name" /etc/group | awk -F: '{print $3}' '

patience% echo $gid
40

patience% set parent="/tmp/home"

patience% set home_directory=${parent}/${directory}

patience% echo ${login_name}::${user_id}:${gid}::${home_directory}:/bin/c:
student2::110:40::/tmp/home/student2:/bin/csh

patience% echo ${login_name}::${user_id}:${gid}::${home_directory}:\
/bin/csh >> /tmp/passwd
```

Case Study: C Shell Programming

The full command can be typed at the terminal before inclusion in the script. Since the */etc/passwd* file wants the group *ID number* not the group *name,* we store the result of this substitution in a variable, *gid.* Later, when we construct the new entry in */etc/passwd,* this information will be available simply by referencing this variable. Echoing *$gid* verifies that the variable has been correctly set. This script assumes that the home directories of any new users are to be rooted at a single location, stored in the variable *parent.* This makes it easy to change that location per the needs of any particular system.

The second *echo* command demonstrates the concatenation of variables. The braces ("{}") around a variable allow it to be recognized without any spaces around it.

Finally, we are ready to construct the new entry for */etc/passwd.* First we attempt this interactively. The entry must consist of:

1. Login name
2. A pair of colons (which will later surround the user's encrypted password)
3. The new user's ID number followed by another colon
4. The group ID number
5. A pair of colons (to surround the user's name)
6. The new user's home directory followed by another colon
7. The shell to be invoked for the new user upon login.

The dummy file */tmp/passwd* is created and one line is put into it – the entry for the new user. In actual practice, this line would be appended to the *system's /etc/passwd* file, but for purposes of this module, we don't want to risk the integrity of this critical system file. Note that the trailing backslash ('\') is a technique for continuing a command on a new line.

Case Study: C Shell Programming

```
patience% head -19 adduser
#!/bin/csh -f
# Csh Program: adduser
# create environment variables:
#     login_name
#     group_name
#     directory
#     user_id
setenv login_name student2
setenv group_name guest
setenv directory student2
setenv user_id 110
set gid=`grep "$group_name" /etc/group | awk -F: '{print $3}'`
set parent="/tmp/home"
set home_directory=${parent}/${directory}
echo Adding $login_name
echo "member:" $group_name $gid
echo "as" $user_id in $home_directory
if (! -e /tmp/passwd) touch /tmp/passwd
echo ${login_name}::${user_id}:${gid}::${home_directory}:/bin/csh >>
/tmp/passwd
```

Case Study: C Shell Programming

Assembling these commands into the file *adduser* provides us with a starting script.

The script sets the environment variables, converts the group name to a group number, attaches the new user's home directory to */tmp/home* and creates the appropriate entry in */tmp/passwd*. We are confident that this all works because we have tried each command individually.

Case Study: C Shell Programming

```
patience% tail -9 adduser
mkdir $home_directory ${home_directory}/bin
cd $home_directory
cp /.profile .
cp /.cshrc .
cp /.login .
chmod 755 . bin .cshrc .login .profile
chgrp $group_name . bin .cshrc .login .profile
# only root can run the following line
#/etc/chown $login_name . bin .cshrc .login .profile


patience% set login_name=$<; echo $login_name
student2
student2
patience% cat script_frag1
#! /bin/csh -f
echo -n "Please enter the new user's login name: "
set login_name=$<
if ({ grep "^${login_name}:" /etc/passwd }) then
        echo $login_name already in /etc/passwd
        exit
endif
```

Case Study: C Shell Programming

The work of the *adduser* script is not yet done. It must not only create a new entry in the *passwd* file, but must also create for the new user a home directory, a directory in which to put new executables, and some rudimentary setup files including *.cshrc* and *.login*. The *adduser* script must set the appropriate ownership and permissions for these new directories and files.

However, to be truly useful, *adduser* must be interactive. That is, it must elicit certain information from its user rather than reading it from environment variables. This can be accomplished with a form of redirection. The construct '$<' reads from the standard input, which is, by default, the keyboard. Here, the variable *login_name* receives a string typed at the keyboard. The command *echo $login_name* verifies the result.

Next is a fragment of a script we can use to enhance the *adduser* script. First, it prompts the user with an *echo* command. Second, it sets the variable to the string input at the keyboard. Third, it runs a *grep* command and tests the outcome. The purpose is to make sure that the new user name is not already taken. The braces ('{}') indicate that we are looking for success or failure of the enclosed command. The string sought by *grep* is enclosed in double quotes (" ") and includes *regular expression metacharacters* which mean: at the beginning of a line (^) find the *$login_name* followed immediately by a ':'. The braces around the variable *login_name* make it recognizable even though there is no white space around it. If *grep* finds such an occurrence, the commands following the *then* are executed and the user is told that the name is already in use and the script terminates. The *then* must be on the same line as the *if*.

Case Study: C Shell Programming

```
patience% chmod +x script_frag1
patience% script_frag1
Please enter the login name for the new user: student1
student1::401:40:Good Student:/home/sunray/student1:/bin/csh
student1 already in /etc/passwd


patience% cat script_frag2

#! /bin/csh –f
set again = yes
while ($again == "yes")
echo –n "Please enter the group name for the new user: "
set group_name=$<
  if ({ grep –s "^${group_name}:" /etc/group }) then
    set again = no
    set gid='grep –s "$group_name" /etc/group | \
              awk –F: '{print $3}' '
    endif
end
echo $gid

patience% csh script_frag2
Please enter the group name for the new user: some
Please enter the group name for the new user: other
40
```

Case Study: C Shell Programming

Although there are other ways to run scripts, the most common is to make them executable and to invoke them as ordinary commands. This is accomplished with a *chmod* command. This is done, and the script is run.

A second script fragment provides error handling more graceful than simply exiting. It sets the variable *again* to "yes" and tests this variable in a *while* loop. The *end* in the second–to–last line is the target of the *while* which stops looping when *again* is modified. The user is prompted to enter a group name with the *echo* command and the response is captured in the variable *group_name.* As before, *grep*'s outcome is tested to determine whether the group name exists (in the file */etc/group).* This time, however, the error condition arises if the name is *not* on file. If *grep* finds the string *$group_name* at the beginning of a line followed by ':', the variable *again* is set to "no" and the group number is checked as it was in our non–interactive version of the script. If *grep* fails, the user is prompted again.

Next, we run the interactive portion of the script on a group name that is not on file ("some") and then on one that *is* ("other"). When it finds "other", it *echoes* the group ID it found.

Case Study: C Shell Programming

```
#! /bin/csh -f
# Csh Program: adduser
# create environment variables:
#     login_name
#     group_name
#     directory
#     user_id
setenv login_name student2
setenv group_name guest
setenv directory student2
setenv user_id 110
set gid=`grep "$group_name" /etc/group | awk -F: '{print $3}'`
set parent="/tmp/home"
set home_directory=${parent}/${directory}
echo Adding $login_name
echo " member:" $group_name $gid
echo " as" $user_id in $home_directory
if (! -e /tmp/passwd) touch /tmp/passwd
echo ${login_name}::${user_id}:${gid}::${home_directory}:/bin/csh \
    >> /tmp/passwd
mkdir $home_directory ${home_directory}/bin
cd $home_directory
cp /.profile .
cp /.cshrc .
cp /.login .
chmod 755 . bin .cshrc .login .profile
chgrp $group_name . bin .cshrc .login .profile
# only root can run the following line, IF we were using /etc/passwd
#/etc/chown $login_name . bin .cshrc .login .profile
```

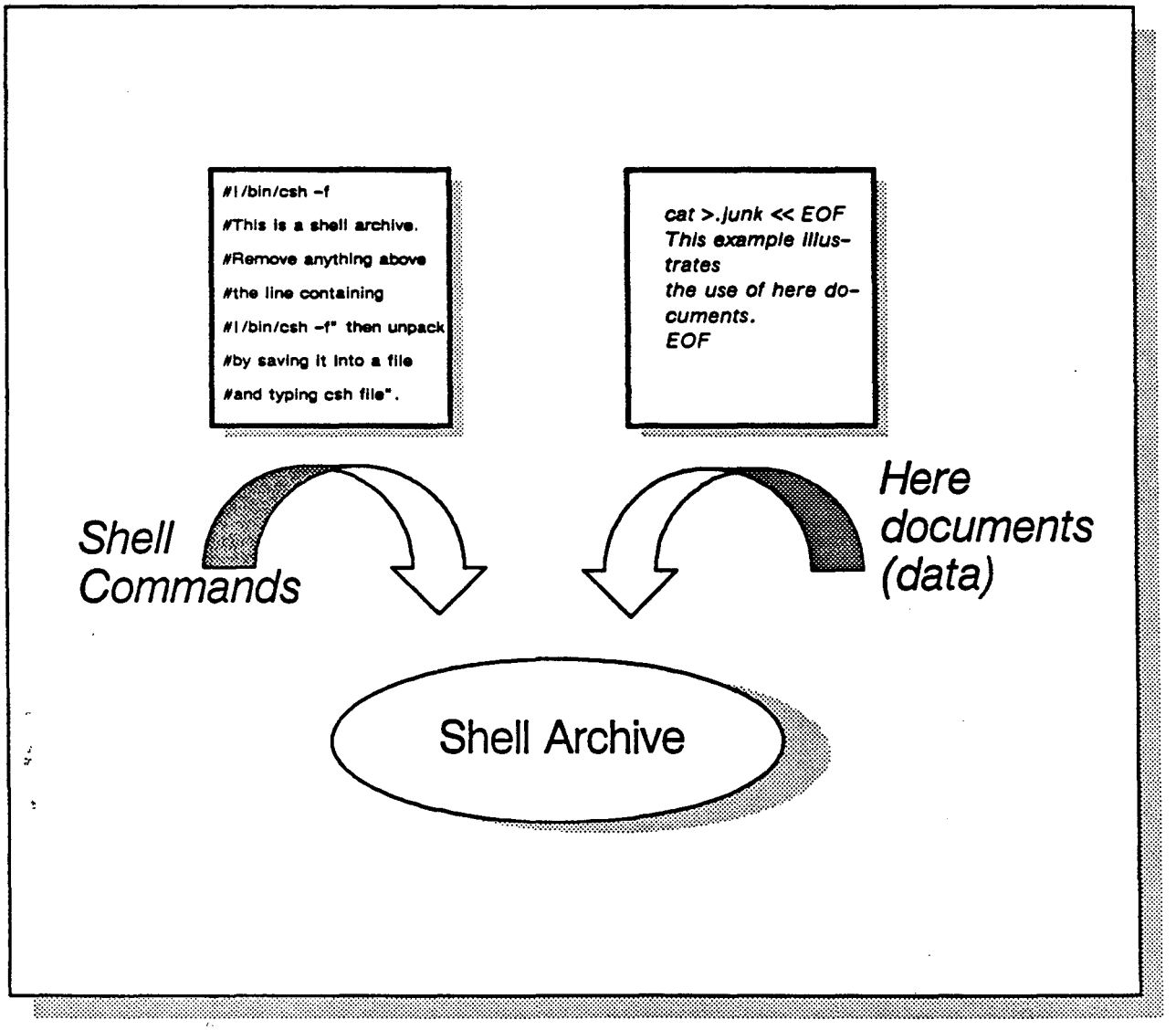
Case Study: C Shell Programming

Combining the incomplete *adduser* script with these script fragments – to make a complete and useful system administration utility – is left as an exercise for the student.

Shell Archives

Definition of a *Shell archive*:

- A *shell script* that installs files contained within the script upon execution.



Shell Archives

Bulletin boards, electronic mail, and freeware have popularized shell archives. Shell archives are shell scripts that can be delivered by electronic mail systems and automate the installation of software. They are composed of a combination of shell commands and embedded data. This embedded data is called a *here document.* A *here document* is used to pass data to commands that are normally executed interactively.

Here Documents

The following illustrates a *here document.* This would appear as part of a shell script.

```
.
.
cat > .junk << abc
This example illustrates
the use of here documents.
abc
.
.
```

The construct << *abc* translates: "take the input starting on the next line from this script up to a line containing only *abc*". Consequently, the two lines following the *cat* are placed in the file *.junk.*

When creating a shell archive and sending it via electronic mail, avoid sending lines containing only a period or lines starting with a tilde (~), as these characters have special meaning to the SunOS mailer. This insures the shell archive will reach its destination undamaged.

A Shell Archive

```
patience% cat gsort.shar
#! /bin/csh -f
#This is a shell archive. Remove anything above
#the line containing #! /bin/csh -f then unpack
#by saving it into a file and typing csh file.
#If this archive is complete, you will see the
#following message at the end:
#
#End of shell archive."
#
#Contents: gsort gsort.awk gsort.sed gsort.test
#
#Edit the following line to install gsort
#in another directory:
set DIR=/usr/local/bin
echo Installing gsort in directory \"$DIR\"
echo shar: Extracting \"gsort\"
cat > $DIR/gsort << END_OF_gsort
#! /bin/csh -f
awk -f $DIR/gsort.awk \$* | sort | \
        sed -f $DIR/gsort.sed
END_OF_gsort
chmod 755 $DIR/gsort
```

A Shell Archive

This example illustrates the use of a shell archive to install the Group Sort utility named *gsort.* This utility sorts files in which the individual items to be sorted span several lines, and are separated by a period on a line by itself. For example, *gsort* is useful in sorting a list of names and addresses separated by periods.

This shell archive installs four files: the C shell script *gsort*, the *awk* script *gsort.awk* and the *sed* script *gsort.sed* and *gsort.test.*

A shell variable *DIR* is set to the name of the directory in which *gsort* is to be installed. The quotation marks in *echo* statements are preceded by a backslash (\) to prevent the C shell from interpreting them. Otherwise, they would not appear in the output produced by the shell archive.

This *cat* statement builds the *gsort* script. The output is directed to *$DIR/gsort.* This becomes */usr/local/bin/gsort* after expansion of *$DIR.* The input to *cat* is taken from the shell archive itself through I/O redirection. The sequence *<< END_OF_gsort* on the *cat* command line instructs the C shell to copy subsequent lines of the script itself, up to but not including the line 21, into the *stdin* of the *cat* command. This is called a *here document.* Notice that the *$DIR* is expanded with the *here document.*

The contents of */usr/local/bin/gsort* become:

```
#! /bin/csh -f
awk -f /usr/local/bin/gsort.awk \$* | sort | \
        sed -f /usr/local/bin/gsort.sed
```

A Shell Archive

gsort.shar continued . . .

```
23      echo shar: Extracting \"gsort.awk\"
        cat > $DIR/gsort.awk << END_OF_gsort.awk
25      /^\.\$/  {
                print rec "."
                rec = ""
                next
        }
30      {
                rec = rec\$0":NL:"
        }
        END {
        if (rec != "")
35              print rec "."
        }
        END_OF_gsort.awk
38      chmod 644 $DIR/gsort.awk
```

A Shell Archive

The file */usr/local/bin/gsort.awk* is created next. *Lines 25* through *36* are copied into the file *gsort.awk*.

The order of the *awk* statements is important. The /^\.$/ pattern matches a period on a line by itself. The associated action prints any combined lines that have been accumulated in the variable *rec* then appends a period. The *rec* variable is cleared. The *next* statement restarts processing with the next input record.

Otherwise, the statement without any pattern matches all input lines. This action appends the current record *$0* to the *rec* variable followed by **:NL:** . The *END* actions prints the last accumulated record.

The *gsort.awk* script joins all lines surrounded by lines containing only periods into one line. The string **:NL:** is placed between the original lines. For example, the file:

Name 1
Address 1
.
Name 2
Address 2
.

becomes:

Name 1:NL:*Address 1*:NL: .
Name 2:NL:*Address 2*:NL: .

This output can then be sorted properly using the SunOS utility *sort*.

A Shell Archive

○ *gsort.shar continued . . .*

```
39      echo shar: Extracting \"gsort.sed\"
40      cat > $DIR/gsort.sed << END_OF_gsort.sed
        s/:NL:/\
        /g
        END_OF_gsort.sed
        chmod 644 $DIR/gsort.sed
45      echo shar: Testing gsort. The following
        echo: names and address should appear
        echo: in alphabetical order:

        sed "s/^X//" << END_OF_gsort.test | \
50              $DIR/gsort
        XJoy, WN
        XSun Valley Way
        XSunnyvale, CA
        X.
55      XAho, AV
        XGarden State Street
        XMurray Hill, NJ
        X.
        XKarels, MJ
60      XComputer Center Court
        XBerkeley, CA
        X.
        END_OF_gsort.test
        echo shar: End of shell archive.
65      exit 0
```

A Shell Archive

The shell archive next creates the file */usr/local/bin/gsort.sed*. The *sed* script *gsort.sed* globally replaces each **:NL:** with a new-line character.

Therefore the *sed* command:

```
s/:NL:/\
/g
```

converts every occurrence of the string **:NL:** into a new-line.

The next part of the script provides data to test *gsort*. Three names and addresses separated by periods are piped into *gsort*. Notice the test input contains an *X* in the first column. Prefixing the test data with *X* (or any other character that will not be misinterpreted by the mail utility) prevents the shell archive from being truncated when mailed. This is necessary because the mail utility will interpret a "." (dot) on a line by itself as the end of data. *sed* removes the *X* from the test data.

A Shell Archive

gsort.shar continued . . .

```
patience% gsort.shar
Installing gsort in directory "/usr/local/bin"
shar: Extracting "gsort"
shar: Extracting "gsort.awk"
shar: Extracting "gsort.sed"
shar: Testing gsort. The following
names and addresses should appear
in alphabetical order:
Aho, AV
Garden State Street
Murray Hill, NJ
.
Joy, WN
Sun Valley Way
Sunnyvale, CA
.
Karels, MJ
Computer Center Court
Berkeley, CA
.
shar: End of shell archive.
```

A Shell Archive

The shell archive is executed with the command *csh gsort.shar*. The output displays the results of the installation.

The message:

shar: End of shell archive.

indicates the script completed execution.

Finally, an *ls –l* lists the properly installed files.

```
patience% ls -l /usr/local/bin/gsort*
-rwxr-xr-x  1 admin   110  Jul  5 12:12 gsort
-rw-r--r--  1 admin   134  Jul  5 12:12 gsort.awk
-rw-r--r--  1 admin    19  Jul  5 12:12 gsort.sed
```

Module 7
Processes and Memory Management

Objectives: Upon completion of this module, the student should be able to:

- Identify and describe the components of a SunOS process.
- Describe the layout of the components of a process in virtual memory.
- Use the *ps* command and the *-a, -u, -x* options to display process information.
- State the difference between physical and virtual memory and describe the memory management techniques of paging and swapping.
- Use the process-spawning system calls *fork() and execve()*.
- Use the process-control call *exit()* and the system call *wait()*.

Evaluation:

Perform Lab 5 to 90% proficiency.

Reference information:

- *The UNIX System: A Sun Technical Report,* Sun Microsystems Inc. (p/n 800-1419-02).

Introduction to SunOS Processes

SunOS Operating System Features:

- Multi–tasking
- Multi–user

SunOS Processes:

- Managed by the *kernel*

  The *kernel is* a set of routines and data structures.

- SunOS kernel system calls

  Allows programmers to gain access to system resources.

Introduction to SunOS Processes

SunOS is a multitasking, multiuser operating system. This means that a number of operations (tasks) can be running simultaneously and that a number of people (users) can be using the computer hardware at the same time. Most computers, of course, can execute only one instruction at a time, so multitasking is usually implemented by interleaving the execution of instructions from the different tasks in some organized manner. The multiuser aspect is just an extension of multitasking, which allows the tasks to be controlled by different computer users at the same time. The operating system hides the details of running the tasks from the user, giving the impression that each person has the machine's undivided attention.

Processes

Tasks running under SunOS are called processes. Processes are managed by the operating system kernel, which keeps track of the machine resources being used (CPU, memory, peripheral devices). The kernel itself is not a process but a set of routines and data structures that are accessible to all computer users through programming library functions called system calls.

Using the SunOS kernel system calls is the only way to gain access to the machine resources. The kernel prevents processes from having any effect on each other, except in specified ways. This means that a process can run amok without causing damage to other processes.

Kernel Components

Process	Function
swapper	Controls swapping of entire processes.
init , getty	The ability to login to each terminal.
scheduler	The interleaving of all running processes.
pagedaemon	Controls paging.

Kernel Components

Process 0

When the computer is turned on (booted up) *process 0* is created which then starts the *init* process. The *init* process then spawns a number of other processes, one of which is called *getty.* This is the process which is responsible for putting the *login* prompts up on each terminal. Then *process 0* becomes the *swapper.*

The *init* process spawns a *getty (get tty)* process which is owned by *process 0*. The *init* then waits for the getty process to terminate. The *getty* displays the login prompt. When you have entered your user name, then *getty* spawns the login process which is responsible for verifying the user's name and password. If they are valid, *login* overlays itself with a login shell which is owned by that user. When the user logs out this process terminates, the *init* process is reawakened, and a new *getty* is spawned at that terminal port.

The swapper

When there are too many processes running on the system and the kernel cannot find enough free pages of physical memory, then the *swapper* is awakened. The *swapper* uses various criteria to find a process that can be swapped to disk in its entirety.

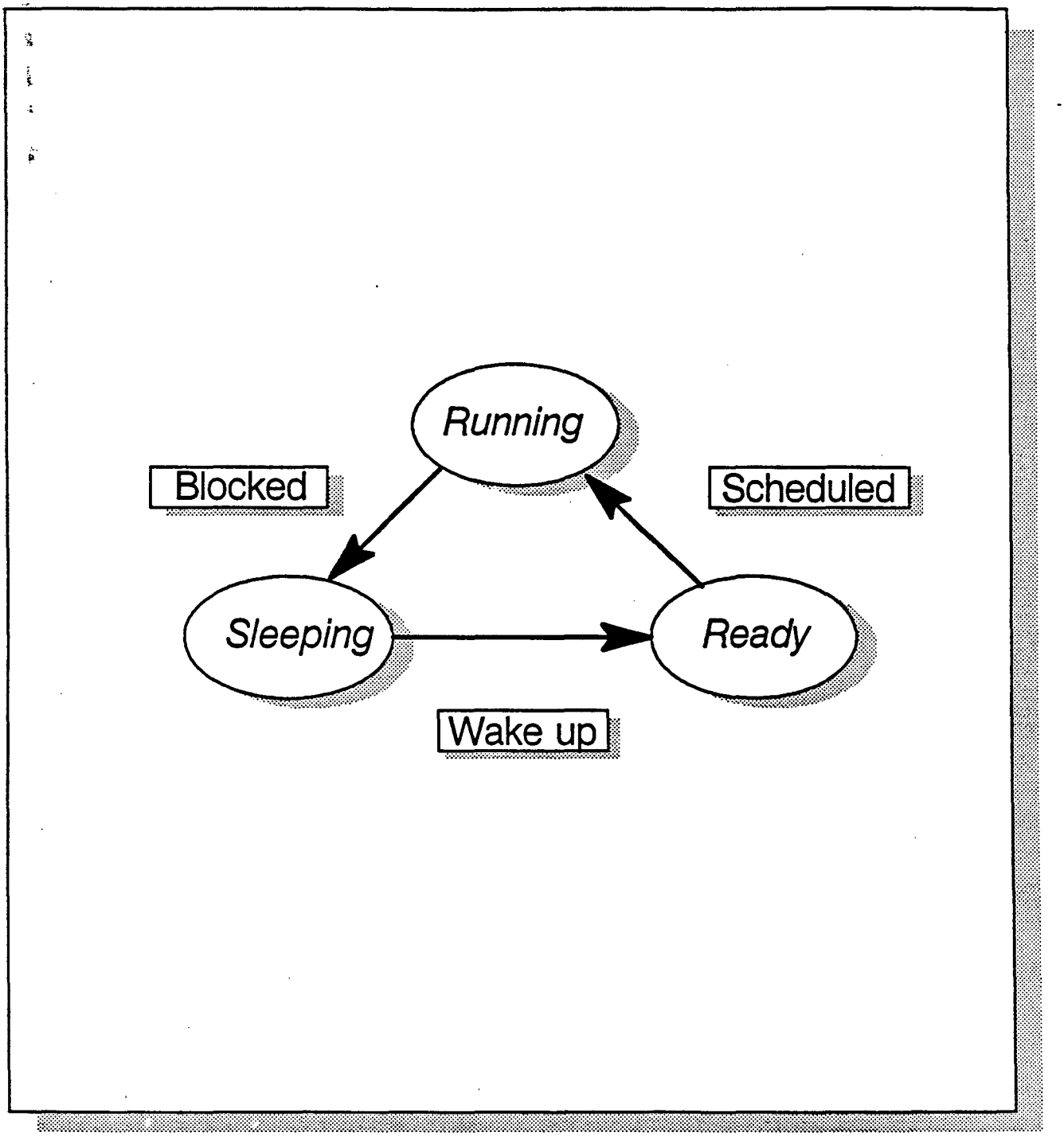
The scheduler

The *scheduler* allocates the CPU to all processes active on the system.

The pagedaemon

A *daemon* is a process that is always present on a system but is usually sleeping. Daemons run without controlling terminals. The *pagedaemon* is awakened when a process requires a page that is not currently in memory.

The Process Scheduler



The Process Scheduler

The kernel contains a process table which is used by the *scheduler* to track the state of every process running on the system. The *scheduler* is the closest thing to a real-time process in the SunOS system. It is almost entirely interrupt driven, and falls between being a real SunOS process and being the only program running on the machine, i.e., it is in charge of every other process.

Process States

Whenever a SunOS process becomes *blocked* (waiting for a character from the terminal keyboard, for instance) the *scheduler* is notified. It causes the process to be moved from the running state to the sleeping state and replaces it with one which is in the ready state. This is called a *context switch* because the computer starts to execute a completely different task.

The *scheduler* is also awakened periodically by a clock timer. If the currently running process has been hogging the computer, it is forced out and another process takes its place. This allows every process a chance to run. If a process has been blocked for a long time and the memory it occupies is needed by executable processes, it may be swapped out. This is described in more detail in the Memory section.

Parent and Child Processes

Process Types:

- Parent

  Process 0 is the ultimate parent of all processes.

- Child

  Spawning creates new child processes.

Process Ownership and Groups:

- Users *cannot* control processes which they do not own.
- Processes can be grouped into subsets by *owner*, by *group*, or by their immediate parent.
- System calls can change the ownership of the calling process.

Parent and Child Processes

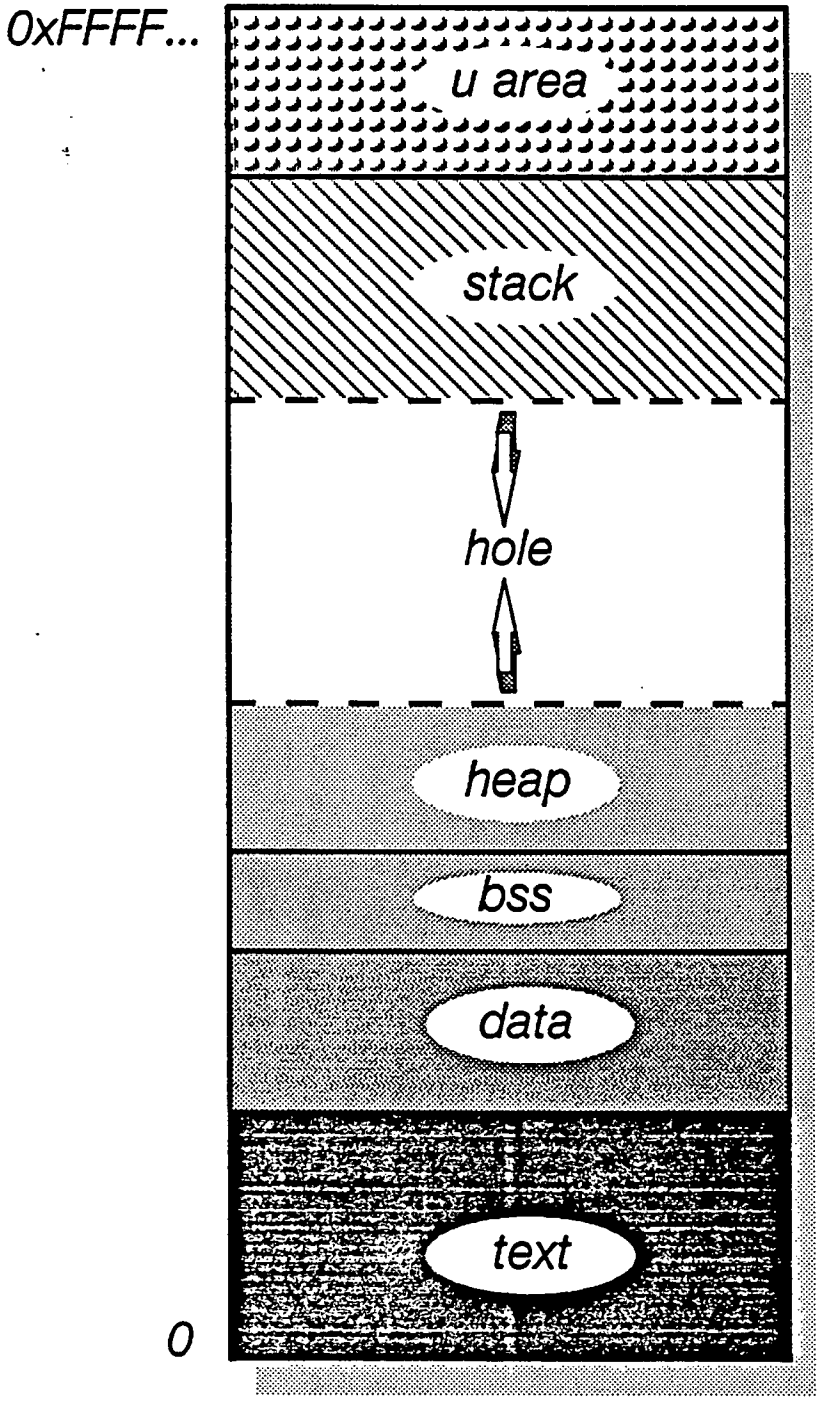
Process 0 remains on the system until the computer is taken down (or goes down of its own accord). It is the ultimate parent of every other process. Each child process has a parent, and the mechanism of creating a new child is called *spawning.* Any process, child or otherwise, can spawn sub–processes. Each process has an identifying number called the *process id.* This *pid* is assigned sequentially, starting at 0 and distinguishes one process from another.

Process Ownership and Groups

Processes have owners just like files and can be grouped into subsets by owner, by the group memberships of the owner, and by their immediate parent. Users cannot control processes which they do not own, and it is difficult for a child to have an unplanned effect on a parent.

Process Virtual Address Space

What a process looks like in memory:



- Process Header
  File Descriptors
  Env. Variables
- Function Calls
- Free space
- Dynamic Allocation
- Uninitialized
- Initialized
- Program Instructions

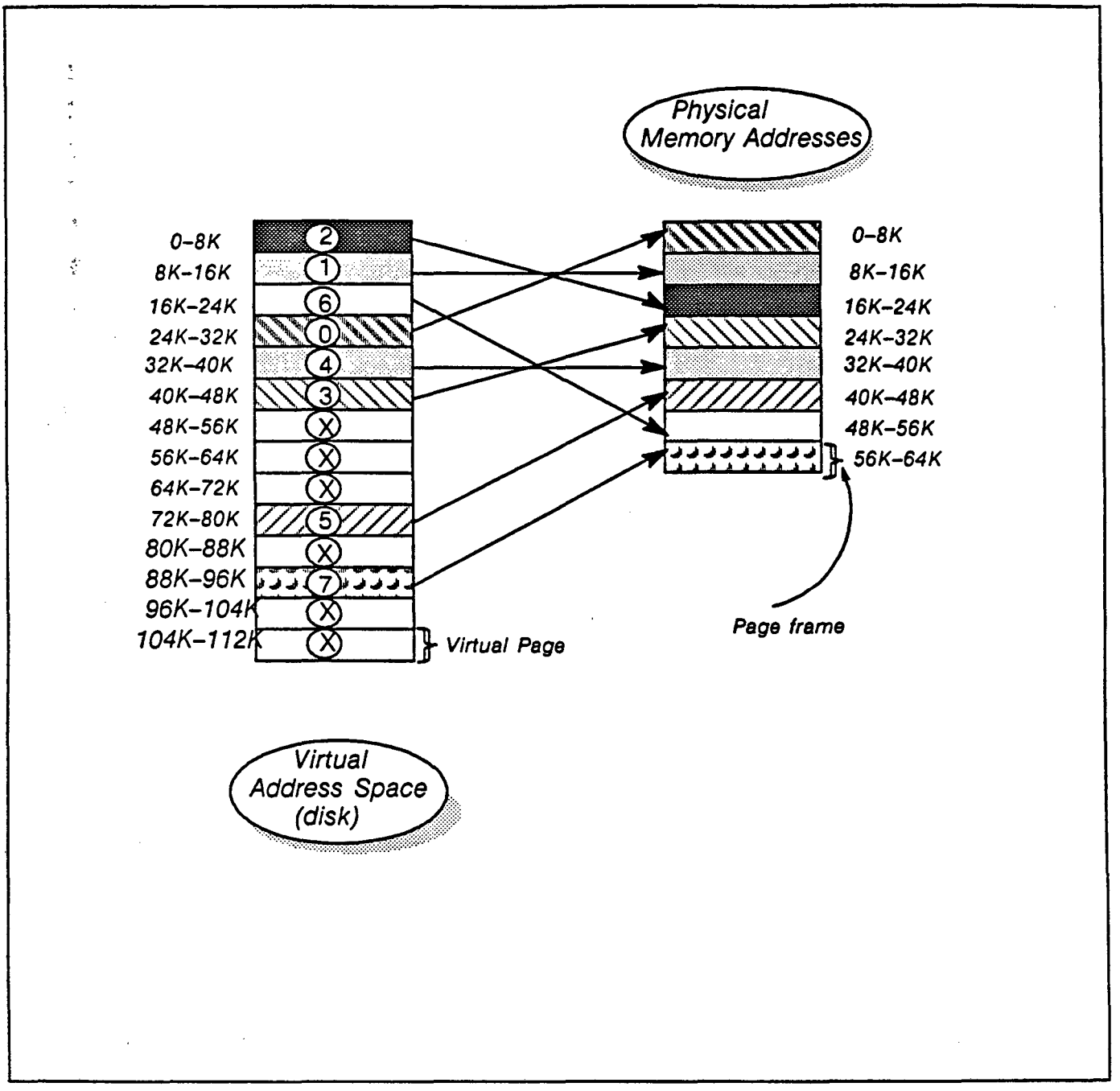
Process Virtual Address Space

Process virtual address space is divided into seven sections:

text	The program instructions.
data	Initialized program variables.
bss	Uninitialized program variables.
heap	Dynamically allocated structures.
free space	Area available for either dynamic allocation or stack frames.
stack	Contains: calling arguments, return address and variables local to the called routine.
u area	Composed of: 1. Process header which contains: user–id, group–id, priority, parent process–id and other system bookkeeping information. 2. File descriptors for each open file. 3. Environment variables.

The *text* and *data* areas are at the lowest virtual memory address for the process. The *heap* area grows upward as the process requests more memory. The *stack* area is located at a high virtual memory address and grows downward.

Physical vs. Virtual Memory



Physical vs. Virtual Memory

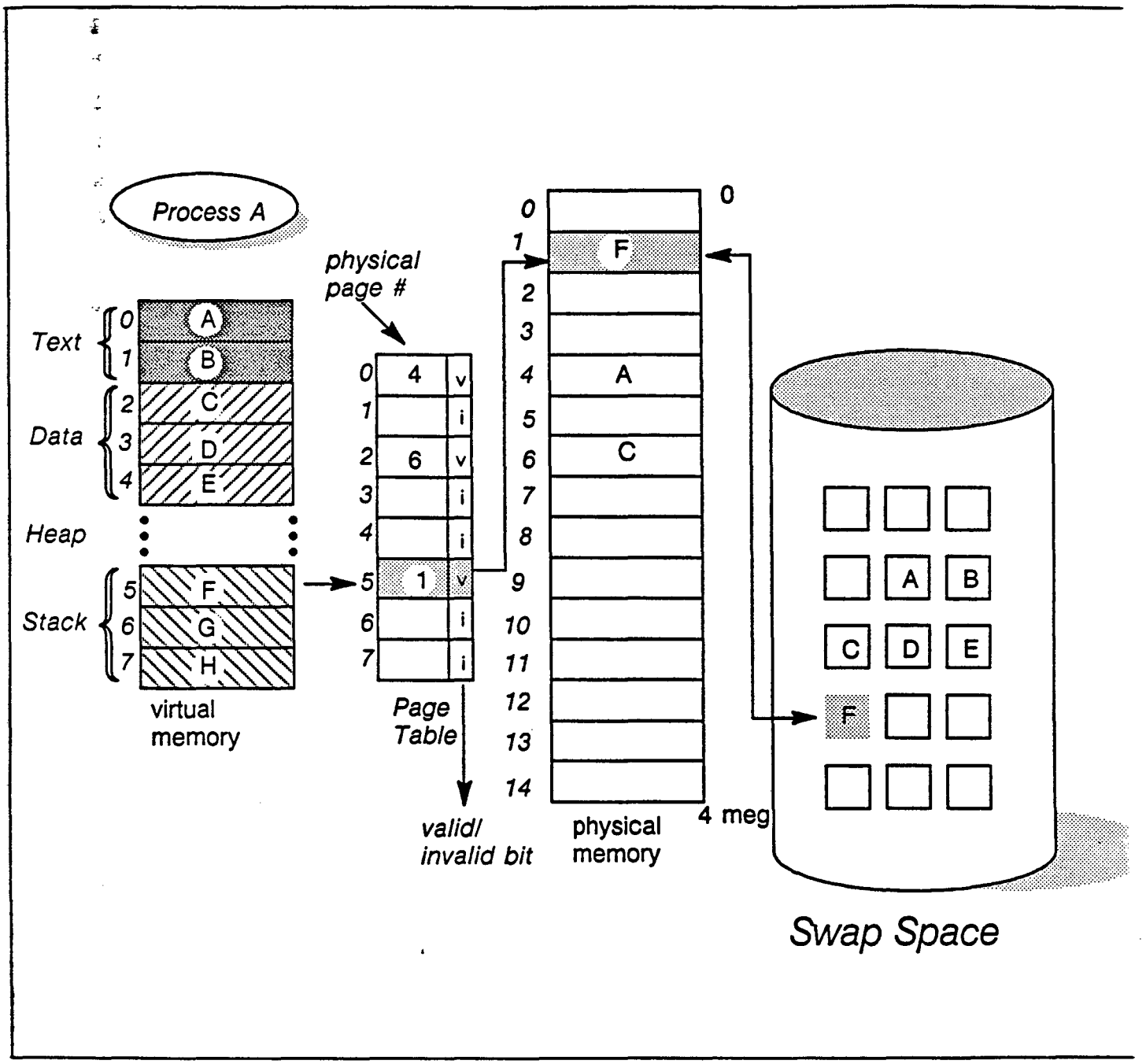
Memory

All computer systems have memory devices which store instruction sequences (programs) and data for the instructions to act upon. The most useful type of memory is under direct control of the processor and is called physical or core memory. (Core is an old type of memory which was made of little magnetic donuts ("cores") with wires passing through them. These days, physical memory is made of integrated circuits containing millions of transistors.) The next most useful type of memory is a magnetic disk or tape, often called secondary storage because it is used only when there is not enough room in core. Access to information stored on secondary memory devices is usually slower.

Physical and Virtual Memory

Because every computer has a limited amount of physical memory, techniques have been developed which use secondary devices to store not only whole executable programs and data files, but also parts of processes that are currently being run. SunOS manages both the physical memory and the secondary devices to make it appear that there is a much larger amount of physical memory available on the system. The two main techniques employed are called *paging* and *swapping*.

Memory Paging



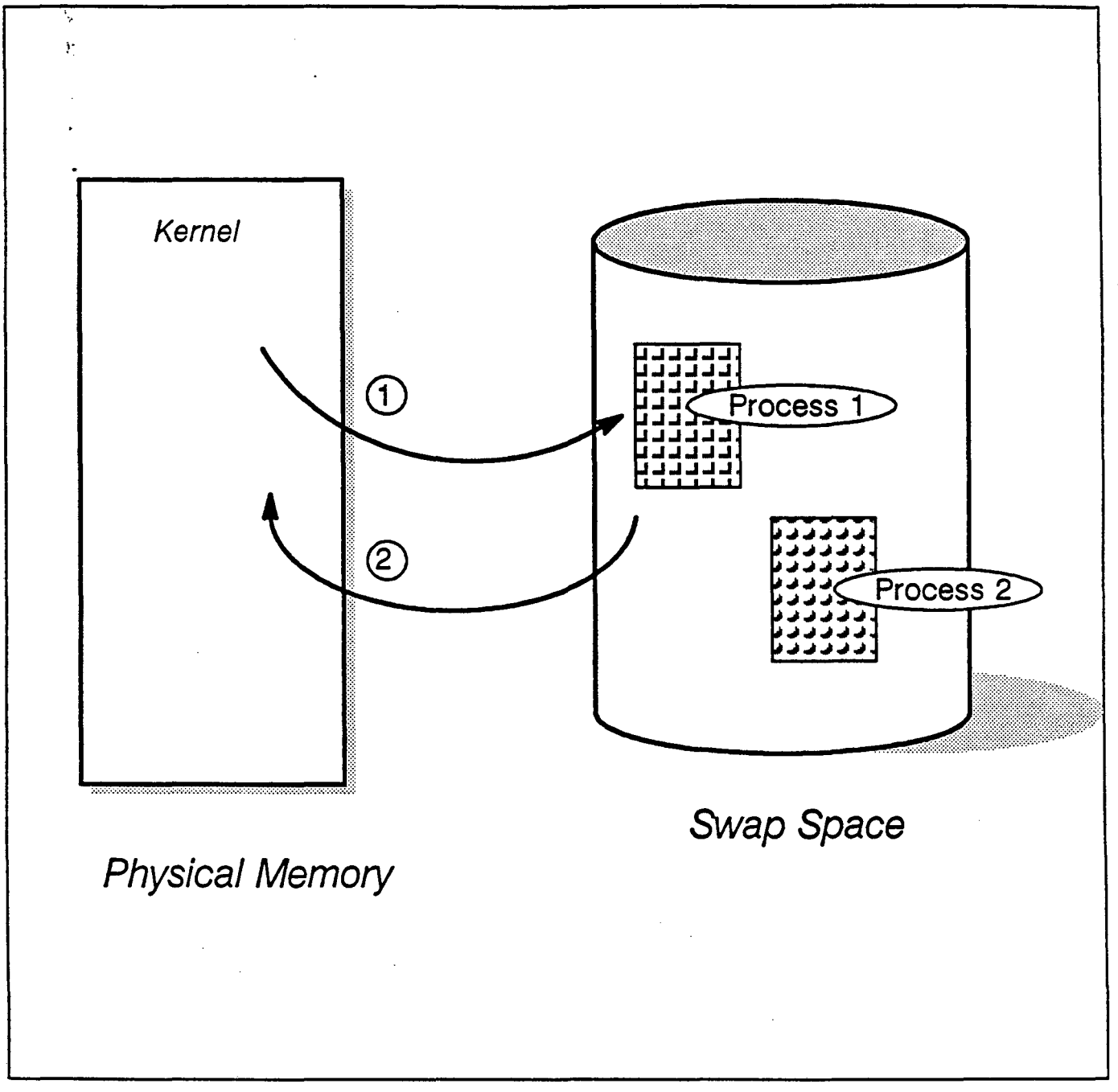
Memory Paging

A program on the Sun computer can address 2**32 bytes of memory, but the system probably contains less than 2**24 bytes of physical memory. Because only one byte of memory is in use at any particular time, not all the memory used by a program has to be present in core at the same time. The contents of memory which is not in immediate use may be copied to a secondary storage device until needed. This is called *paging.*

On SunOS, physical memory is divided into *page frames* of some convenient size (8 Kbytes on a Sun–3 and Sun–4, 4Kbytes on the Sun–386i, and 2 Kbytes on a Sun–2). Processes request and release pages of virtual memory as needed using system calls. When a process addresses one of its *pages* (to read or write its contents) the system checks to see if the *page* is actually resident in physical memory. If it isn't, the process pauses while the contents of the page are *paged in* (retrieved from secondary storage). This is called a "page fault".

The *paging* process loads the requested virtual page into an available physical memory page and causes the system hardware to map the virtual address used by the process to the actual physical address of the page.

Swapping



Swapping

When there are many processes running on the system, and all the available physical memory is in use, the paging process may not be able to find a page of memory to page out. In this case the *swapping process* is notified. The *swapper* uses various criteria to find a process which can be sent in its entirety to secondary storage. This process is *swapped out* and doesn't get a chance to run for a few seconds while its old physical memory is put to other uses.

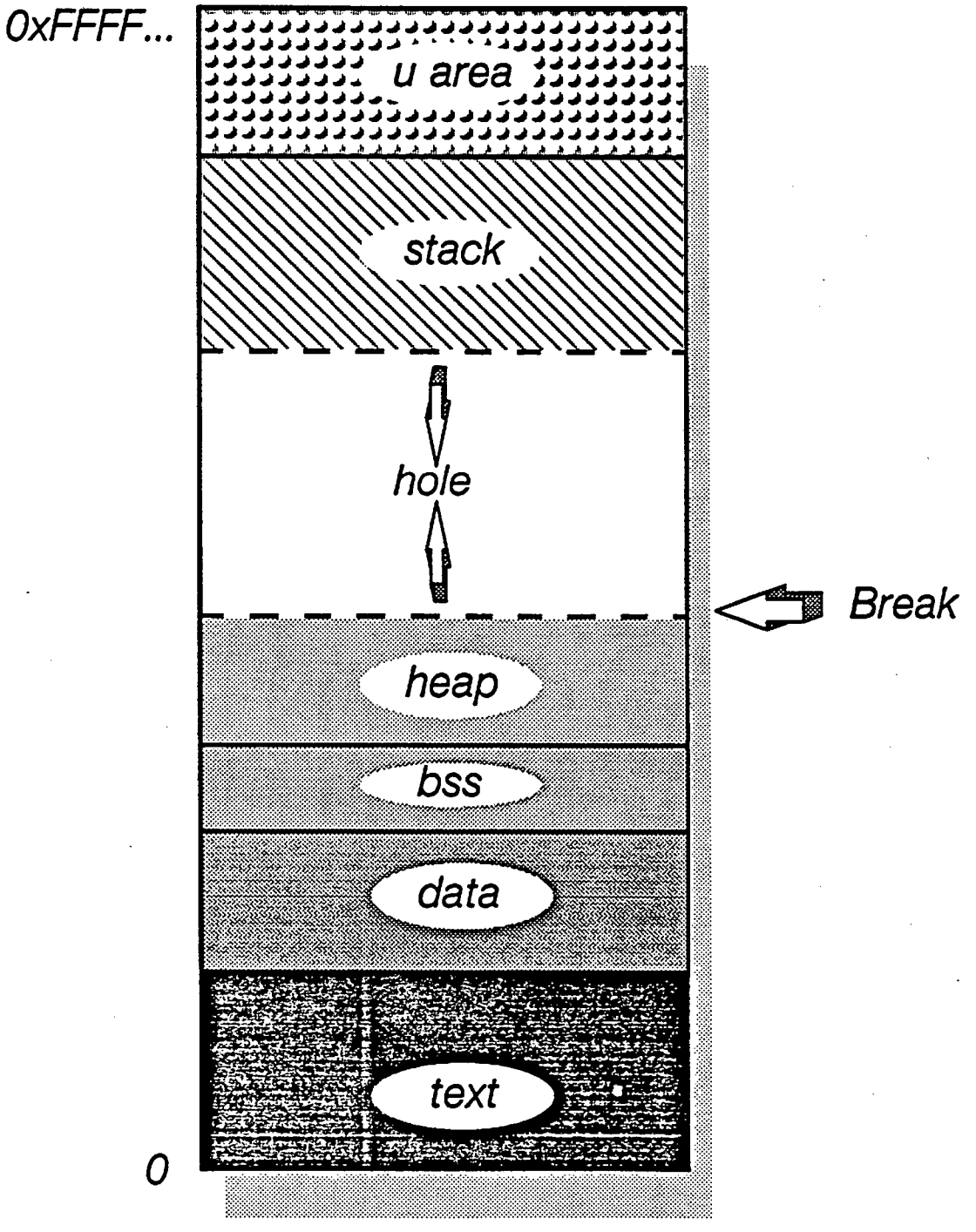
The *scheduler process* can cause the swapped process to be *swapped in* and run at a later time. *Swapping* is faster than the original *exec* from the disk because it is just a copy of memory contents that is being moved, usually in a large block.

Both *paging* and *swapping* use up secondary storage (disk), which is not infinite either. There is an absolute limit to the amount of virtual memory a process can get. This limit is defined by the amount of swap space allocated on the secondary storage devices. The kernel can also set user level limits on the amount of memory which can be allocated to any given process.

Since it is used by every process, the memory occupied by the kernel itself is not paged in and out for efficiency reasons.

Memory Management Library Calls

- *malloc()*
- *free()*



Memory Management Library Calls

The boundary between the *heap* and *free space* is called the *break*; it is the boundary between valid and potentially valid addresses.

The SunOS library calls which manage a process's memory are:

malloc() Allocates a block of memory of a requested size (and in doing so moves the *break.)*

free() Releases memory allocated by *malloc().*

Dynamic Memory Allocation

Program : *getmem.c*

```
1   /* getmem.c -- Allocates memory and sleeps.
    * Gets new-line-terminated message from stdin
    * and executes do-nothing behavior:
    * 1st message word = amount of memory to grab.
5   * 2nd message word = time to sleep before exit.
    * The rest is ignored.
    */
    #include <stdio.h>
    #include <strings.h>
10  #include <malloc.h>

    #define MAX_BUF 100  /* input buffer size */

    main (argc, argv)
15  {
      char inbuf[MAX_BUF];
      unsigned bytes;
      int seconds = 0;
      char *cp;
20    int count;
```

Dynamic Memory Allocation

Program Objective

This program uses C library routines to get a line of text from the standard input, parse it, and convert it into two numbers. The numbers are then used as arguments to functions which allocate memory to the process and cause it to suspend operation for a specified amount of time. *getmem.c* demonstrates the memory allocation functions.

Lines 1–20

This is the preamble to the program. It contains C preprocessor directives, and the variable declarations.

Dynamic Memory Allocation

Program continued: *getmem.c*

```
21      if (gets (inbuf) && *inbuf)
        {
           bytes = atoi (inbuf);
           if ( cp = index (inbuf, ' ') )
25            seconds = atoi (cp+1);
           if ( (cp = malloc (bytes)) == NULL )
           {
              fprintf(stderr, "getmem: malloc() failed");
              exit (1);
30         }
```

Dynamic Memory Allocation

Line 21

The call to *gets()* fills *inbuf* with characters from standard input. *gets()* does not return until a <Return> is received on the port (the new-line character is not put into the string). Normally *stdin* is connected to the user's terminal keyboard, so this call gets a line of characters typed on the terminal. If *gets()* returns 0 the input data reached the End Of File (EOF). If the first character of *inbuf* is '\0' the user just typed a <Return> and *gets()* returned an empty line.

Lines 23–25

These lines parse and convert the input line into two integers. *atoi()* converts the first set of numeric characters it finds into a decimal integer. *index()* returns a pointer to a character (in this case, a space) in the given string.

Line 26–29

The call to *malloc()* allocates the requested number of bytes of memory to the process. *malloc()* returns a pointer to the first byte of this memory. The pointer is guaranteed to be aligned on a usable boundary for any data type, and the memory block is in one contiguous piece. The library call *malloc()* returns a NULL pointer if not enough memory is available. Then the program prints an error message and exits.

Dynamic Memory Allocation

Program continued: *getmem.c*

```
31          else
            {
               /* do something with the memory */
               for (count = 0; count != bytes; ++count)
35                cp [count] = (char) count;
               free (cp);
               sleep  (seconds);
            }
         }
40       exit (0);
      }
```

Dynamic Memory Allocation

Lines 31–35

Accessing the memory forces the pagedaemon process to put the allocated pages into "core" physical memory.

Line 36

The call to *free()* sends the memory block allocated by *malloc()* back to the process's free memory list. It doesn't go back to the system until the process exits. As a result, it can be recovered if another *malloc()* is done. This also means that the process exists in its largest size on the swapping device until it exits. This may prevent other processes from being able to start up or run.

Line 37

sleep() suspends a process for a specified number of seconds (give or take 1). If the process sleeps long enough it may be swapped out to make room for executable processes. This, of course, also fills up the swap device space.

Line 40

The process terminates, sending an exit status of 0 to the parent.

Dynamic Memory Allocation

Compiling and executing *getmem.c*:

patience% **cc getmem.c –o getmem**

patience% **getmem**
1000 1

patience% **time getmem**
1000000 10
2.1u 0.2s 0:16 14% 0+256k 0+1io 3pf+0w

patience% **echo 1000000 10 | (time getmem)**
2.2u 0.1s 0:12 19% 8+256k 0+0io 19pf+0w

patience%

Dynamic Memory Allocation

To run the program *getmem.c* first compile it, then enter the program name *getmem*, followed by a carriage return at the command prompt. (The C compiler and *cc* command will be discussed in more detail in the Compiling and Linking module.) Two numbers, **1000** and **1** are now input followed by a carriage return. The process allocates memory and sleeps. Then it exits and the shell prompt is displayed.

The C Shell *time* command can be used to run any command line and print the amount of time used during its execution. The fields of the *time* output line are:

2.1 u	Time used in user function calls
0.2s	Time used in system calls
0:16	Total elapsed time.
14%	Processing time as percentage of elapsed.
0+256k	Average amount of shared + unshared memory used (in Kbytes)
0+1io	Number of input + output block operations.
3pf+0w	Number of page faults + process swaps.

The *echo* command and a shell pipe can be used to eliminate the amount of time it takes to type in the input line needed by *getmem*. The *stdin* of the *getmem* process is connected to the *stdout* of the echo process. Thus *echo* writes its arguments into *getmem* as if they were typed on the terminal directly. The parentheses around *(time getmem)* force the command to be executed as one unit. This version of *time* is a C Shell built–in. A later module contains more details on *time*.

Monitoring Processes *ps*

```
patience% getmem
1000000 100
^Z
patience% bg
[1]    getmem &
patience% ps -u
USER  PID %CPU %MEM   SZ  RSS TT STAT TIME COMMANC
schip  227   7.7    3.8  312   272 p1 R      0:00   ps u
schip  225   5.4    6.3 1040  464 p1 R      0:01   getmem
patience% ps -u
USER  PID  %CPU %MEM   SZ  RSS TT STAT TIME COMMANC
schip  225   10.1   13.5 1040 1016 p1 S     0:02  getmem
schip  231    0.0   3.8    312 272  p1 R     0:00  ps u
patience% ps -u
USER  PID  %CPU %MEM   SZ   RSS TT STAT TIME COMMAN
schip  233   23.1    3.8  312  272  p1 R     0:00 ps u
schip  225    6.6   13.5 1040 1016  p1 I     0:02 getmem
patience%
[1]    Done       getmem
patience% ps -u
USER   PID %CPU %MEM   SZ  RSS TT STAT TIME COMMANI
schip  250   23.1    3.8 312  272  p1 R     0:00  ps u
```

Monitoring Processes *ps*

The program *getmem.c* uses C library routines to get a line of text from the standard input port, parse it, and convert it into two numbers. The numbers are then used as arguments to functions which allocate memory to the process and cause it to suspend operation for a specified amount of time.

Running the *ps* command displays information about processes. The *-a, -u* and *-x* options are most commonly used. The *-a* option allows you to include processes that are not owned by you (that do not have your user ID). The *-u* option displays user-oriented output. This includes fields: *USER, %CPU, SIZE*, and *RSS* as described below. Lastly, the *-x* option allows you to include processes without controlling terminals.

USER	The user's name.
PID	Process id number.
%CPU	Average cpu usage over life of process.
%MEM	Average virtual memory usage.
SZ	Actual size of virtual memory image (Kbytes).
RSS	Resident size of process (amount in core).
TT	Terminal this process is associated with.
STAT	Process state, up to four letters: *R* runnable *T* stopped *P* page wait *D* wait on disk *S* sleeping (< 20 seconds) *IW* idle (> 20 seconds) and process is swapped out *Z* zombie, a dying process
TIME	CPU time used
COMMAND	Command line

Process Creation and Control

System and Library calls which create and control new processes:

Call	Function
fork()	Creates the process.
exit()	Terminates a process.
wait()	Waits for a forked process to finish.
execve()	Loads a program for the process to run.

Process Creation and Control

Process Creation

The *fork()* system call makes an identical copy of the process which called it. *fork()* returns an integer value which is either:

-1	An error occurred and *fork()* failed.
0	This is returned to the child copy of the original process.
anything else	This is returned to the parent copy of the original process, and the value is the process-id(pid) of the child.

The only differences between the parent and child processes at this point are the *fork()* return value, the process id of each process, and the settings of some resource usage variables (which are reset in the child). The child does have a copy of the parent's file descriptors which, if manipulated, can affect subsequent reads and writes by the parent.

Process Control

When a process calls *exit()*, it causes itself to be terminated. It dies and is removed from the process table. *exit()* has an argument which is passed back to the parent process as the child's exit status through the *wait()* system call in the parent. The child process isn't actually removed from the process table until the parent reaps (receives) this exit status. The parent must call *wait()* for each of its children, or they will become *zombies*, processes which aren't doing anything but still exist in the system. This type of process is a drag on system resources. If the parent process exits before the child, the child is inherited by the *init* process, which reaps (and ignores) its exit status.

Process Creation and Control

System calls which create and control new processes:

Call	Function
fork()	Creates the process.
exit()	Library call to terminate a process.
wait()	Waits for a forked process to finish.
execve()	Loads a program for the process to run.

Process Creation and Control

The *execve()* system call, and its library relatives *execv()*, *execl()*, etc, cause the calling process's memory contents to be replaced by a new program, usually loaded from disk storage. This is the way a new program starts executing. Since the original program has been replaced, the *exec* calls do not return on success (there is nowhere to return to). *execv()* is useful to use when the number of arguments is unknown in advance and *execl()* is useful when a known file with known arguments is being called. The next example will illustrate the use of *execl()*.

The most common cause of failure of an *exec* call is because it can't find the new program on the disk. If this occurs, *exec* will return an error status to the original program. The *exit()* call releases all the system resources which are still being used by the process. This includes closing files and freeing memory, but it is still good programming practice to do these things explicitly.

Process Creation and Control

Program: *fork.c*

```
1   /* fork.c – demonstrate fork(), execl(), exit() and wait().
    This example spawns a child to execute an 'ls', and
    waits for the child to finish, exiting if any errors occur.
    */
5   #include <stdio.h>
    #include <sys/wait.h>
    main()
    {
        int   pid,deadpid;
10      union wait status;

        switch (pid = fork())
        {
            case -1:
15              fprintf(stderr, "fork failed\n");
                exit(1); /* error */
```

Process Creation and Control

Program Objective

This C program demonstrates the use of the process creation calls: *fork()*, *execl()*, *exit()*and *wait()*. This example spawns a child to execute an *ls*, and waits for the child to finish, exiting if any errors occur.

Lines 1 –10

This is the preamble to the program. It contains C preprocessor directives, and the variable declarations. The wait union is a 32–bit word comprised of 16 unused bits, 8 bits containing the exit status (w_status), 1 bit indicating if a core image was produced (1 if true), and 7 bits containing the term signal number. It is the w_status area (an integer) that indicates the cause of termination and other information about the terminated process.

Line 12

Call *fork()* to start the child process and save the pid which is returned by *fork()*.

Lines 15 –16

It is possible for *fork()* to fail. In the event that it does fail, it will print out a message that it failed and quit. (Note: By convention any nonzero argument which is passed to *exit()* indicates an error. *exit(0)* indicates no error.)

Process Creation and Control

Program: *fork.c*

```
17              case  0: /* in child */
                     printf("child: pid = %d\n",getpid());
                     execl("/bin/ls","ls",(char *)0);
20                   fprintf(stderr, "exec failed\n");
                     exit(2); /* error */
22              default: /* in parent */
                   printf("parent: pid = %d\n",getpid());
                   printf("parent: my child is %d\n",pid);
25                 deadpid=wait(&status);
                   printf("parent: child %d just died\n", deadpi
                   exit(0); /* success! */
            }
    }
```

Process Creation and Control

Line 18

We are in the *child* process, and the *child* wants to print out its pid. The system call *getpid()* returns the pid of the process so that we can print it out.

Line 19

We want the child to execute the *ls* command via the *execl()* system call because we know which file we want to execute and what the arguments are. The parameters to *execl()* are NULL terminated character strings. The first argument is the name of the program to execute (Note: It doesn't search your path). This is followed by any number of arguments where the first is the same as the filename, in this example *ls*. The final argument must be (char *)0.

Lines 20–21

If the program executes to line 19, the *execl()* has failed so print an error message and exit. Otherwise the child is now executing *ls*.

Lines 23–24

The parent is now printing out its pid and the child's pid.

Line 25

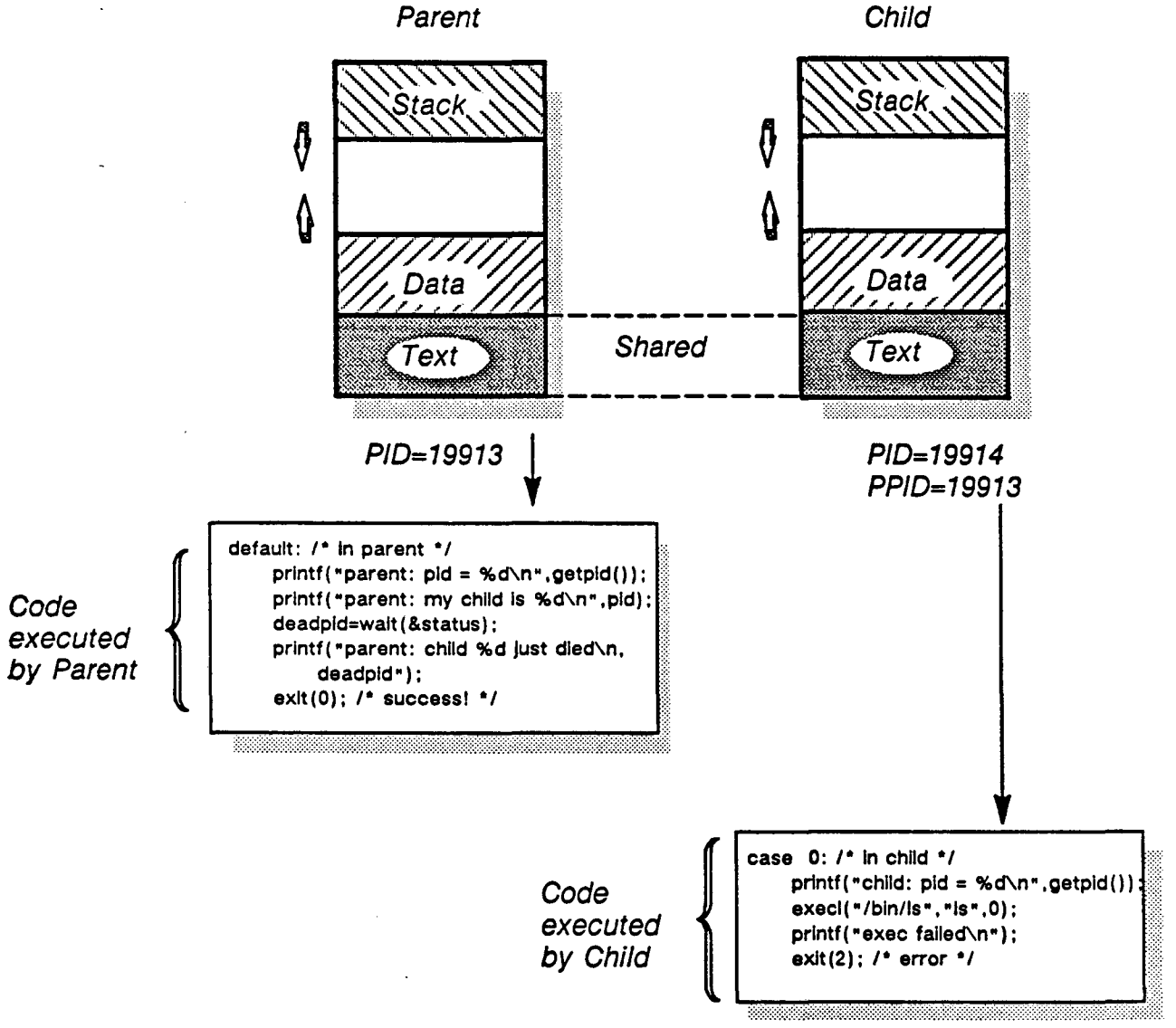
We want to wait for *ls* to finish, so we call *wait()* and the parent will sleep until the child dies and returns the pid of the dead child.

Lines 26–27

We now print out the the message that the child just died and its pid is displayed. The *exit(0)* status indicates there are no errors. The program has completed successfully!

Process Creation and Control

- *fork()* creates a child process



Process Creation and Control

```
patience% cc fork.c -o fork

patience% fork
child: pid = 19914
parent: pid = 19913
parent: my child is 19914
Asteroids   Patch       Zorkstatus        super
C           Postscript  desktop           test
Debug       Sockets     mbox              test.c
Demos       Stuff       printcap.allans
Gab         SysInt      q
News        TM          script
Othello     Worms       srogue
parent: child 19914 just died
```

Module 8
Compiling and Linking

Objectives: Upon completion of this module, the student should be able to:

- Demonstrate the usage of the C source code formatter *vgrind.*
- Utilize *indent* to beautify C source code.
- Demonstrate the usage of the C source code verifier *lint.*
- Use the C compiler *cc.*
- Use the preprocessor *cpp.*
- Use the link editor *ld.*
- Use the commands *error* and *ctags.*
- Utilize the archive librarian utility *ar* and add a table of contents with *ranlib.*
- Demonstrate the use of the object utilities: *nm* and *size.*
- Compile and link Pascal *(pc)* or FORTRAN *(f77)* programs.

Evaluation:

Perform Lab 6 to 90% proficiency.

Reference information:

- Appendix D, *Source Code.*
- *The UNIX System: A Sun Technical Report,* Sun Microsystems, Inc. (p/n 800-1419-02).
- *SunPro, The Sun Programming Environment,* Sun Microsystems, Inc. (p/n FF144/20K).

C Program Beautifiers: *indent* and *vgrind*

Command Format:

indent [–options] [inputfile [outputfile]]

patience% **more scen.c**

```
#include "scen.h"
main()
{
int             i = 10, j = 20, k;
printf("main runs \n");
k = MAX(i, j);
func1();
func2();
}
```

patience% **indent scen.c**

patience% **more scen.c**

```
#include "scen.h"
main()
{
    int             i = 10, j = 20, k;
    printf("main runs \n");
    k = MAX(i, j);
    func1();
    func2();
}
```

C Program Beautifiers:
indent and *vgrind*

C Program Formatter

indent is a general C source code formatter. It *indents* C code according to the program logic. *indent* formats the program according to options entered on the command line or according to a parameter file, *.indent.pro*, if it exists.

When the *indent* command is executed without naming a output file, it puts a copy of the original file into a file with a .BAK extension (e.g. *file.c.BAK*) and replaces *file.c* with an improved version of itself. To prepare the file for processing by TROFF (similar to processing by *vgrind)* use the *-troff* flag.

C Program Beautifiers: *indent* and *vgrind*

Command Format:

vgrind [options] file.c

A few options:

–h *header*

–l*language*

–t *file.c* | *lpr –t*

C Program Beautifiers:
indent and *vgrind*

C source Code Beautifier

vgrind produces beautiful source code listings on laser printers or other *troff* output devices. It italicizes comments, bolds keywords, lists functions at the right of the page and produces a header which includes the time and date of the listing, the file listed, and the page number. In addition to C, *vgrind* understands Bourne-shell, C-shell, emacs MLisp, FORTRAN, Icon, ISP, LDL, Model, Pascal, and Ratfor.

If the name of your typesetting program is not *troff*, you must set an environment variable (*TROFF*) to let *vgrind* know what program to use. A common case is:

setenv TROFF ptroff

The **-h** option allows you to specify a header for every page of output. The default is the current file name.

The **-l** option allows you to specify what language to use among those listed above. For example, to specify C-shell, use the flag **-l***csh*. To specify Pascal, use the flag **-l***p*.

The **-t** option must be used to avoid a "typesetter busy" message.

For more information on the options, consult the *man* pages.

C Program Verifier: *lint*

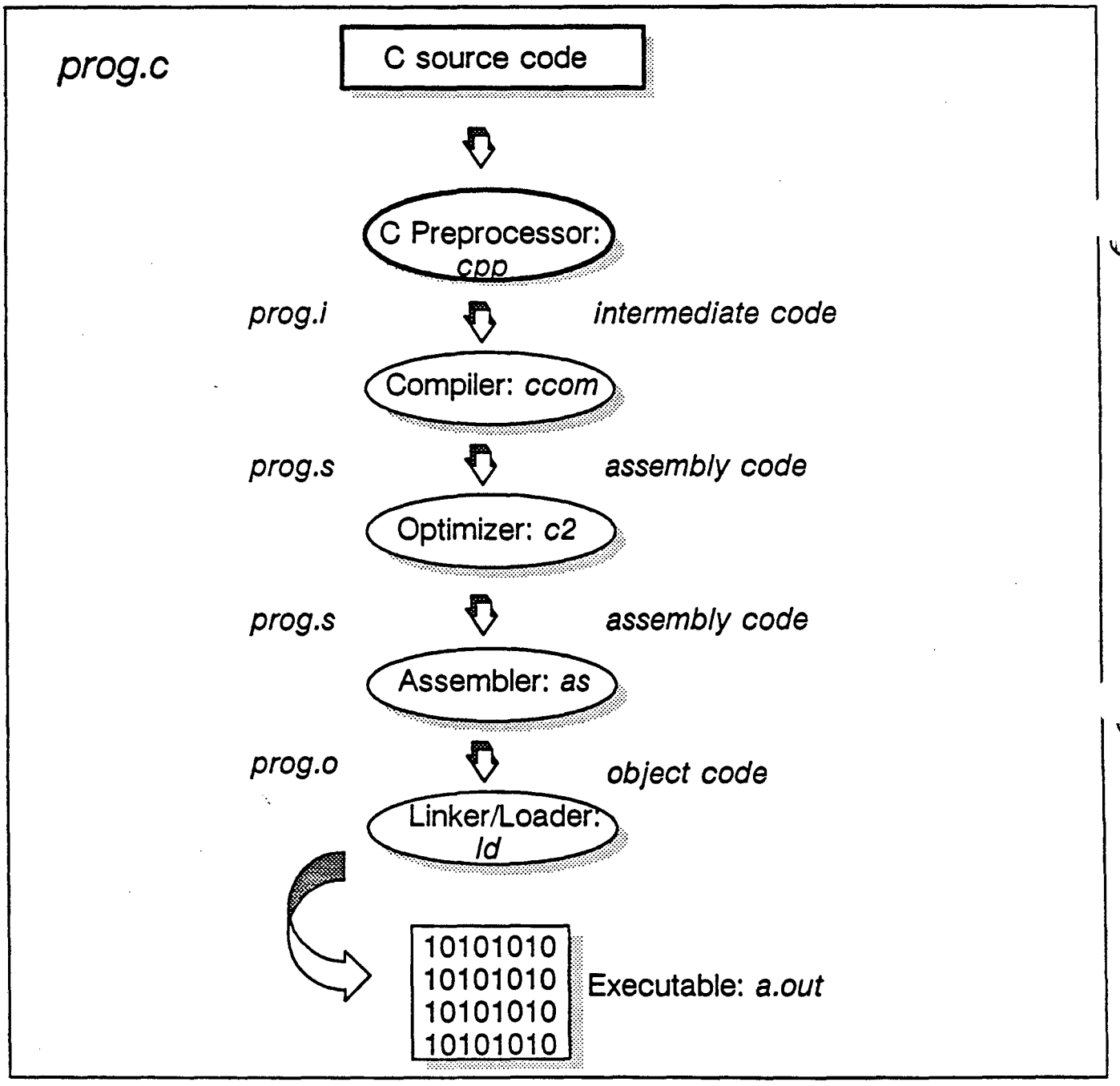
```
patience% more prog.c
    /*    lint test file */
    char a;
    main()
    {
      int x, y, z;
      x=10; y=20; z=30;

      fun(x / y);
      y = (fun(x) + --x);
    }
    fun ( m )
    float m;
   {
      return (2.0 * m);
    }
   stub(s)
   {
      if (1)
         return(27);
      else
         return(17);
   }
```

```
patience% lint prog.c
prog.c:
prog.c(9): warning: x evaluation order undefined
prog.c(6): warning: z set but not used in function main
prog.c(17): warning: argument s unused in function stub
fun, arg. 1 used inconsistently  prog.c(13)  ::  prog.c(8)
fun, arg. 1 used inconsistently  prog.c(13)  ::  prog.c(9)
a defined( prog.c(2) ), but never used
fun returns value which is sometimes ignored
stub defined( prog.c(17) ), but never used
patience%
```

C Program Verifier: *lint*

The program *lint* looks at source code, not object modules, for nonexecuted code, an incorrect number or type of arguments in a function call, return value omissions of a function (or using the return value from a function that doesn't return a value), variables that have been declared but not used, and type mismatches (values assigned to variables of a different type without an explicit cast operator), etc. Because *lint* is very thorough, be forewarned that it *may* print error messages or warnings for reasonable code; these can be ignored.

Overview of the Compile–Link Process for the C Language



Overview of the Compile–Link Process for the C Language

The job of the *compiler* is to convert the text you write into a file which can be loaded and executed by the system on which it is compiled. Such an executable file (often called an *a.out* file after the default output name of the compiler) is not just an image of how the process will look in memory. An executable file is one which is ready to be loaded by the kernel and run. This section will help you understand the processing involved in converting your program to an executable file. In addition, it will cover some of the tools available to you to better understand and control what has happened to your program during its compilation phase.

The compiler *cc* is actually a driver program that invokes several programs as if they were functions. *cc* executes five distinct programs, each of which represents a stage of the compilation process. In addition, a sixth program (*ar*) is used to create object code libraries for *ld* to use.

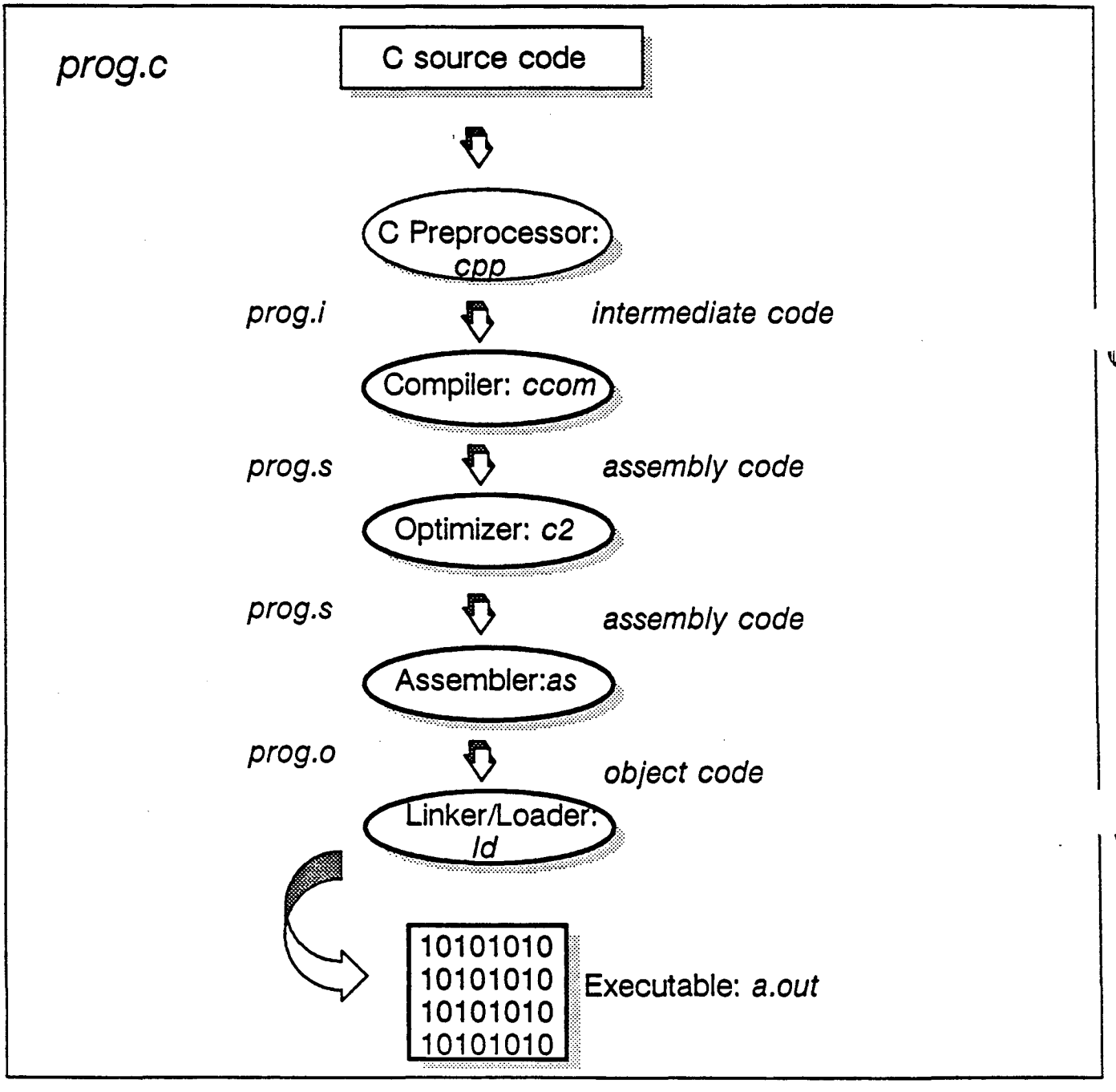
C Source Code

You create a file using an editor such as *vi*, which contains ASCII text comprising the C source code. Filenames must end in a *.c* extension.

cpp

The first program invoked is the C preprocessor (*cpp*). This is an editing pass of the source file. It substitutes all the *#defines* for their current strings, pulls in the *#include* (usually *.h*) files, and performs macro expansion. Invoking *cc* with the *–P* flag stops the processing after this pass and leaves the resultant text file (with a *.i* extension) for you to examine.

Overview of the Compile–Link Process



Overview of the Compile-Link Process

ccom

ccom takes C source code and compiles it into assembly language for the target machine.

c2

There is an optional *-O* flag to *cc* which causes the optimizer to be run. *c2* is the object code optimizer. It optimizes assembly code into assembly language source code which has a *.s* extension. This step takes extra time so you won't usually use it until you are ready to put the code into production.

as

The assembly code in the named file is now translated into object code (which have a *.o* extension) by *as*, the SunOS assembler. The output is a relocatable object module which is close to machine instructions but not executable.

ld

The linker, *ld*, looks at the symbol table in the *.o* file(s) output by *as* and attempts to resolve all the external references and brings any code from libraries which are specified. It also builds the header required for the scheduler to actually load and run a program. Library files are traditionally found in */usr/lib* and named *"libX.a."* Such a library would be referred to on the linker or compiler command line as *-lX*. Note that versions of SunOS 4.0 and higher will have dynamic linking to sharable libraries at run time (discussed later in this module). The executable file produced by *ld,* is named *a.out* by default; the *-o* option allows you to specify an alternate name.

The Compiler: *cc*

Command Format:

cc [options] sourcefiles... [-llibrary]

patience% **cc testfile1.c testfile2.c testfile3.o**

patience% **a.out**

patience% **cc -O testfile1.c testfile2.c testfile3.o**

patience% **ls /usr/lib/l***

```
/usr/lib/lang_info          /usr/lib/libmp.a
/usr/lib/ld.so              /usr/lib/libnbio.a
/usr/lib/lex                /usr/lib/libpixrect.a
. . .
/usr/lib/libm.a             /usr/lib/lpf
/usr/lib/libm.il            /usr/lib/lpfx
```

patience% **cc testfile1.c testfile2.c testfile3.o -lm**

patience% **a.out**

patience% **cc -o testing testfile1.c testfile2.c testfile3.o**

patience% **testing**

The Compiler: *cc*

Based on the file name extension, *cc* compiles, assembles and/or links the various files together as appropriate.

Filenames with a *.c* extension are assumed to be C source code. Filenames with a *.o* or no extension, are assumed to be already–compiled object modules. Filenames with a *.s* extension are assumed to be assembly language source files. All files following the *–l* option are assumed to be libraries regardless of their names. The name provided is assumed to be prefixed by *lib* and given a *.a* extension and referenced in */usr/lib*.

Compiling

testfile1.c and *testfile2.c* are compiled using *cpp* and *ccom*. As part of this process, the object modules *testfile1.o* and *testfile2.o* are created. Next, *testfile1.o, testfile2.o,* and *testfile3.o* are linked together using *ld*. Finally, some temporary files created by *cc* (such as */tmp/testfile1.i and /tmp/testfile2.i*) are deleted. Invoking *cc's* verbose option, *cc –v,* displays each step as it occurs. *cc* passes all unfamiliar and relevant (such as, *–o*) switches to the linker (*ld*).

The Executable: a.out

There are several formats for the executable file (described in more detail later in this module). In no case however, is the executable the same as the process image as it resides in memory. At the beginning of any executable file is information for the system *exec,* such as where the program is to be loaded; where the entry point for the process is; how much space to reserve for the stack; what kind of executable it is, etc.

This header is followed by a long section containing text, the machine language instructions equivalent to your program. This is followed by a data segment containing all the initialized data you requested in your program, then the size of the uninitialized data. Regardless of the name or the details of the format, the executable file created by this process is ready to be loaded and run by the system.

The Compiler: *cc*

options	Meaning
–c	Compile but do not link.
–E,–P	Run source through *cpp* only
–Dname	Same effect as a *#define name*.
–help	Display helpful information about *cc*.
–O	Optimize the object code.
–o name	Assign a filename to the executable
–S	Produce an assembly source file
–g	Generates the extended symbol table
–v	Verbose

The Compiler: *cc*

Useful cc options:

–c

Compile and assemble but do not *link*. An object module (with a *.o* extension) is created for each *.c* or *.s* file listed on the command line but these objects aren't linked together.

–E and –P

Run the source file through *cpp*(1), the C preprocessor, only. Sends the output to the standard output, or to a file named with the *–o* option. The *–E* option includes the *cpp* line numbering information.

The *–P* option also runs the source through *cpp* only, however the output is saved to a file with a *.i* suffix. The *–P* option does not include *cpp*–type line number information in the output.

–D

This option is useful for debugging purposes. For example, debug diagnostics can be surrounded with:

```
#ifdef DEBUG
printf("This is a debug diagnostic");
#endif
```

If the program is compiled with:

```
cc –DDEBUG prog.c
```

the *printf* statement will be compiled. You can also assign a value to the macro:

```
cc –DPROCESSOR=80386 prog.c
```

creates:

```
#define PROCESSOR 80386
```

inside the program.

The Compiler: *cc*

options	Meaning
–c	Compile but do not link.
–E,–P	Run source through *cpp* only
–Dname	Same effect as a *#define name*.
–help	Display helpful information about *cc*.
–O	Optimize the object code.
–o name	Assign a filename to the executable
–S	Produce an assembly source file
–g	Generates the extended symbol table
–v	Verbose

The Compiler: *cc*

-O

Controls the amount of optimization which is performed. The default (no switch) simply skips the optimizer pass so that the compile is faster. Since compilation takes longer if -O is specified, it is recommended that it is invoked after the program is fully debugged.

-o name

This option assigns a user-designated *filename* to the final executable program, in place of the default, *a.out*. If the compilation is to generate an intermediate file (*.i*, *.s*, etc.), then the *name* must have the appropriate suffix for the type of file to be produced by the compilation. Also, this *name* cannot be the same as the source file (the compiler will not overwrite the source file).

For example:

cc -o reindeer dasher.c dancer.c donner.c blitzen.c

calls the final program *reindeer* in place of *a.out.*

-S

Generates an assembly language source file and then terminates. The output file has a .s extension. This option is useful if you want to see what the compiler has done with your source code. You can also hand-optimize the assembler listing and then submit it to *as.*

-g

Generates the extended symbol table used by *dbx* and *dbxtool* to show you the names of the structures, variables, and functions you used in your source program.

The C Preprocessor: *cpp*

- Invoked as the first pass of C compilation using *cc*.
- Translates all # directives.
- Deletes program comments.

The C Preprocessor: *cpp*

cpp is not part of the compiler. It simply goes through the ASCII file, rearranging, substituting or deleting text as you specified with # directives (*#define, #elif, #else, #endif, #if, #ifdef, #ifndef, #include, #undef*). It doesn't understand C syntax, though it does know the meaning of an identifier.

For example:

```
#define    PRINTS(X)    printf("string is <%s>", X)
```

when invoked with:

```
PRINTS( buf );
```

expands to:

```
printf("string is <%s>", buf);
```

Be aware of the common error shown next.

For example:

```
#define    PRINTS(s)    printf("string is <%s>", s)
```

when invoked with:

```
PRINTS( buf );
```

expands to:

```
printf("string is <%buf>", buf);
```

Macro Expansions

Many of the routines you may think of as function calls are actually macro expansions which are done by the preprocessor. This is done because such code is simply inserted in-line with your code and avoids the overhead of calling and returning from a different function. Both *ctype.h* and *stdio.h* contain such macros.

An example of executing *cc* using the *-P* option to view the macro expansions, is provided on the next page.

The C Preprocessor: *cpp*

patience% **cc -P scen.c func1.c func2.c**

scen.h

```
#define MSG  "function %d runs\n"
#define FIRST 1
#define SECOND 2
#define MAX(A,B)          ( (A) > (B) ? (A) : (B) )
```

scen.c

```
#include <stdio.h>
#include "scen.h"

main()
{
    int i=10, j=20,k;
    printf("main runs \n");
    k=MAX(i,j);
    func1();
    func2();
}
```

func1.c

```
#include <stdio.h>
#include "scen.h"
func1()
{
    int func_no = FIRST;
    printf(MSG ,func_no);
}
```

func2.c

```
#include <stdio.h>
#include "scen.h"
func2()
{
    int func_no = SECOND;
    printf(MSG ,func_no);
}
```

The C Preprocessor: *cpp*

scen.h instructs *cpp* to substitute *"function %d runs \n"* every time it sees *MSG*. Likewise, 1 is substituted for *FIRST and 2 for SECOND*. The program *scen.c* is similarly listed and we see that its first statement is a *cpp* directive to include the file *stdio.h*. The rest of the program prints that it is running and calls *func1* followed by *func2*. Note that they each include the file *scen.h* in the second directive to the preprocessor, *cpp*. The quotes around "scen.h" mean that this is to be taken as the literal name of the file (in the current directory location.) In each program, the *cpp* is building a valid *printf* statement.

When cc is invoked with the *-P* option, this displays the *macro* expansions in a file with a name of *filename.i*. If there are multiple files listed in the *cc* command, each will have its own *.i* file (e.g. *scen.i, func1.i and func2.i*).

scen.i (without external definitions):

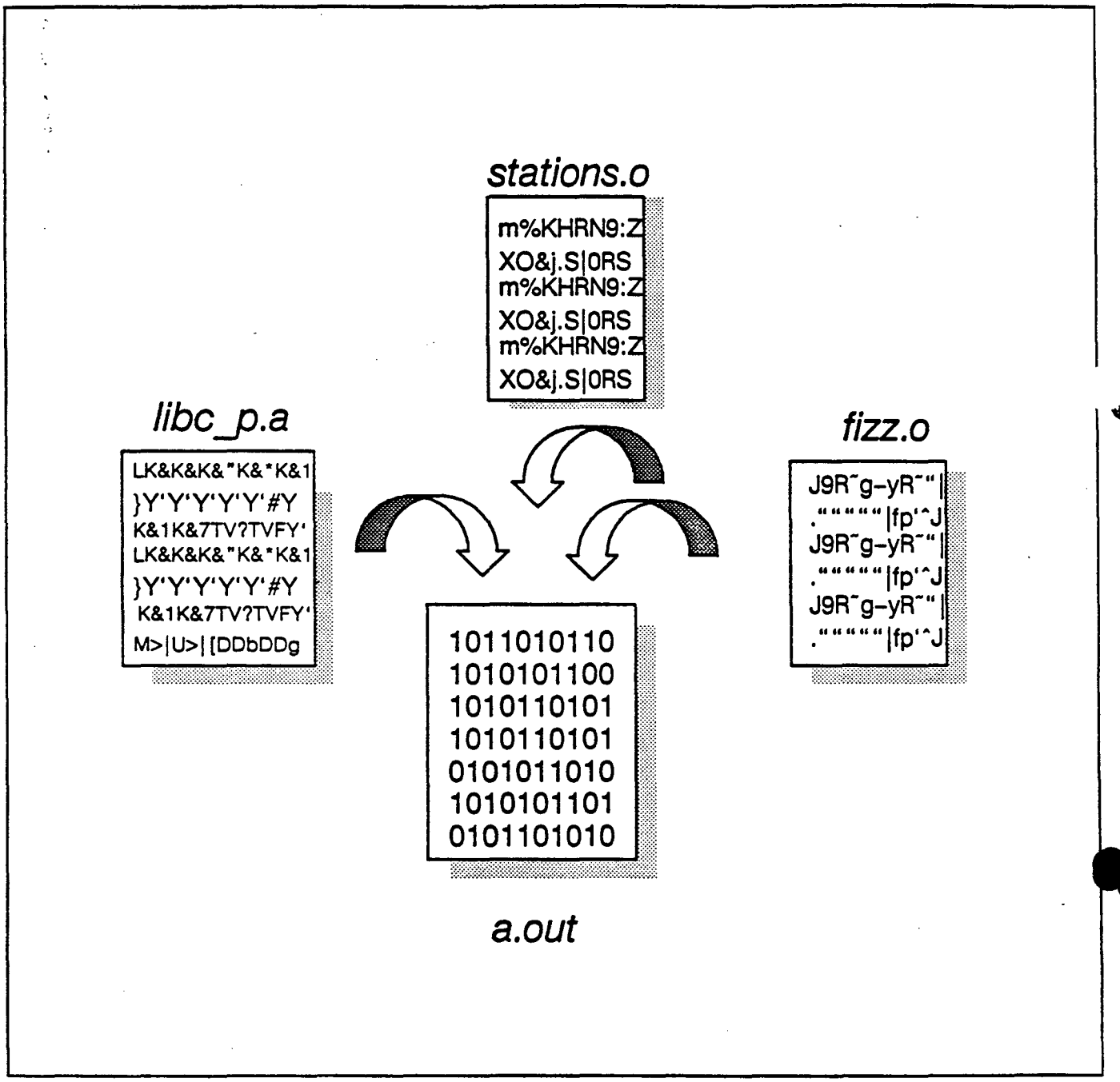
```
main()
{
   int i = 10, j = 20, k;
   printf("main runs \n");
   k = ( (i) > (j) ? (i) : (j) );
   func1();
   func2();
}
```

func1.i and *func2.i* (without external definitions):

```
func1()
{
   int func_no = 1;
   printf("function %d runs\n", func_no);
}
```

```
func2()
{
   int func_no = 2;
   printf("function %d runs\n", func_no);
}
```

The Link Editor: *ld*



The Link Editor: *ld*

The *linker* merges a collection of object modules *(.o* files) and libraries *(.a files)* together into a single file. It takes as input a list of object modules and libraries and generates as output a single file called *a.out*. A library or archive is a collection of *.o* files that have been merged together into a single file with the *ar* utility. Libraries are special because they are "scanned." Normally, if you list a *.o* file on the command line, the module will be linked into your program whether or not you use the functions in the module. A library, however, is not linked in its entirety.

ld includes in the final program only those modules in the library that include functions that are called from modules that have already been linked. If only one function in a module is required, then the entire module will be present in the final program, even if the other functions in the module aren't used. If a function has already been linked into a program, it won't be extracted from the library. So, if you call one of your functions with the same name as a function in a library, then that version of it will be used, rather than the one that's in the library.

ld Options

–lx

This option is an abbreviation for the library name *libX.a,* where *X* is a string. *ld* searches for libraries first in any directories specified with *–L* options, then in the standard directories */lib, /usr/lib*, and */usr/local/lib*. A library is searched for when its name is encountered, so the placement of a *–l* is significant. That is, since libraries are only searched once for functions that the linker cannot yet account for, the library should be last on the command line. Otherwise, some library modules may not be included when they should be.

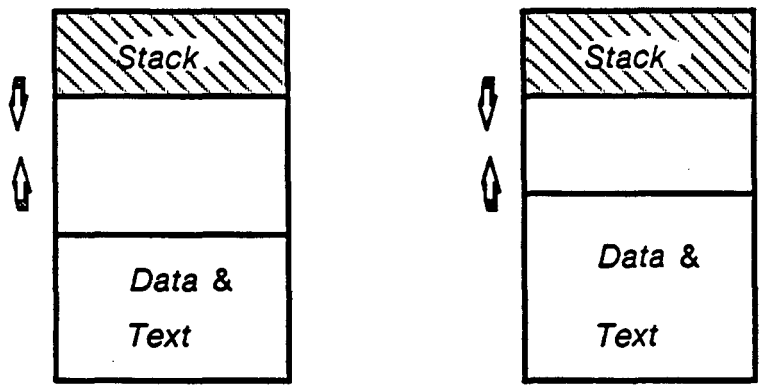
–Ldir

Add *dir* to the list of directories in which libraries are searched for. Directories specified with *–L* are searched before the standard directories */lib, /usr/lib,* and */usr/local/lib.*

Another significant option is *–B*, which specifies static or dynamic linking.

Static Linking

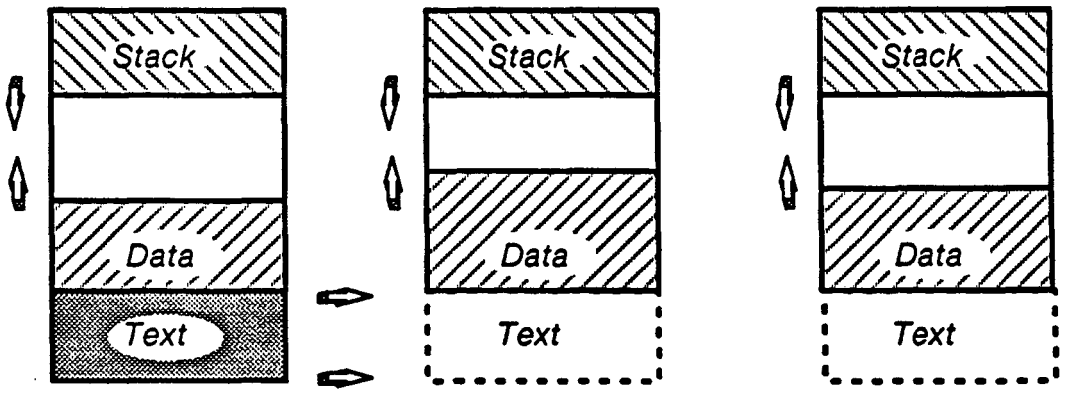
Old "impure" format programs



vi
process #1

vi
process #2

Read–only (shared text) programs



vi
process #1

vi
process #2

vi
process #3

Static Linking

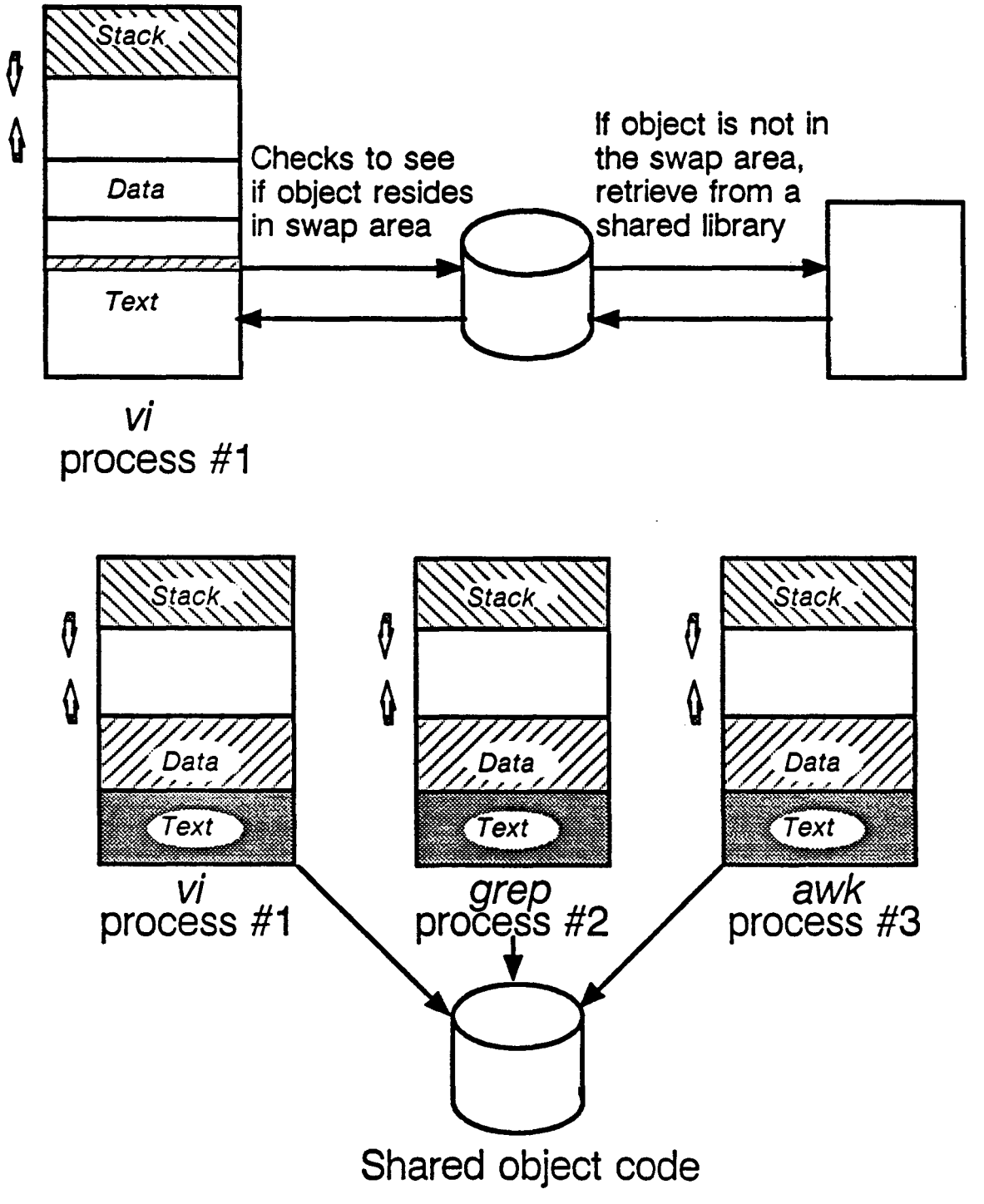
Static linking requires that all symbolic references are resolved at link time (prior to execution). All of the executable code is present in the text portion of a program.

Under the old "impure" format, a program's text and data areas are not separated on page boundaries, so each instance of the program needs its own copy of the text/data areas. However, for read-only programs, the linker places the data segment on the next page boundary. This allows the text area to be "shared" between multiple processes that run the same executable code, and only requires dedicated data and stack areas. In these examples, the miscellaneous processes happen to all be using *vi*.

These two types of executable formats have numbers, called "magic numbers", which identify them. The nonread-only, nonsharable format has a magic number of 407. This is a mostly obsolete executable format, but is sometimes useful when debugging complexes of several processes which are executing the same program. To generate a 407 executable, use the *-N* flag to *ld*.

The magic number for the read-only, sharable format is 410. The entire program should be loaded initially, which takes longer to start up and theoretically improves the paging behavior of the running program. It is rarely used, however, programmers with small programs or those for which it is expected that all of the code in the program will be exercised may find this format useful. Use the *-n* option of *ld* to create an executable of type 410.

Dynamic Linking



Dynamic Linking

Symbolic references are resolved (linked) during program execution. When a function call is made, and the object code is not resident in the text, the system will first check to see if it resides in the swap space. If it is not in the swap space, it will search the object libraries for symbolic resolution and brings the object into the swap space to make it available for future references. These are referred to as "shared libraries".

By utilizing shared libraries, different processes can share common object code. In this example there are three different processes executing three different commands which share a piece of object code.

The magic number for this type of executable format is 413. The text is read-only and sharable. Only the minimum number of pages of the executable are loaded into memory to get started. This makes startup quicker, and increases the probability of paging. If the program doesn't execute all of its text pages, however, only those pages which are referenced get loaded. This type of format is the default but may be explicitly requested with the *-z* option of *ld*.

The *ctags* Command

- A tags file contains the locations of functions and typedefs in a group of files.
- *ex* and *vi* use the tags file to locate these objects.
- This file eliminates the need for a user to remember which source file contains a specific function and/or typedef.
- Command format:

  *ctags [–aBFtw] [–***f** *tagsfile] filename. . .*

The *ctags* Command

The *ctags* command usually calls the tags file *tags*. Users may override this by using the *-f filename* option. The *-a* option tells *ctags* to append the output to an existing tags file. A *-B* says to use backward searching patterns while a *-F* says to use forward searching patterns (the default). A *-t* will create tags for typedefs and a *-w* suppresses warning diagnostics.

Creating a Tags File

```
patience% cat lt.main.c
main()
{
   funA();
   funB();
   funC(funD());
}

funA()
{
   funE();
   funE();
}

funB()
{
   funE();
   funF();
}
patience% cat lt.fun.c
funC()
{
   funC();
   funA();
}

funD()
{
   funB()
}
```

Creating a Tags File

In this example there are two files: *lt.main.c* and *lt.fun.c*. These two files contain several function calls to functions *funA, funB, funC, . . ., funF.*

Creating a Tags File (con't.)

```
funE()
{
   funA();
   funB();
   funC();
}

funF()
{
   funE();
   funD(funE());
}
```

```
patience% ctags lt.main.c lt.fun.c
patience% cat tags
Mlt.main        lt.main.c       /^main()$/
funA    lt.main.c       /^funA()$/
funB    lt.main.c       /^funB()$/
funC    lt.fun.c        /^funC()$/
funD    lt.fun.c        /^funD()$/
funE    lt.fun.c        /^funE()$/
funF    lt.fun.c        /^funF()$/
```

Creating a Tags File (con't.)

To create the tags file for these two files, just execute the *ctags* command and specify the names of these files. Each line of the tags file contains the object name, the file in which it is defined, and an address specification for the object definition.

Using a Tags File

```
patience% vi -t funD
funC()
{
   funC();
   funA();
}

funD()
{
   funB()
}

funE()
{
   funA();
   funB();
   funC();
}

funF()
{
   funE();
   funD(funE());
}
"lt.fun.c" 24 lines, 141 characters
```

Using a Tags File

vi looks in the tags file and discovers that *funD()* resides in *lt.fun.c*, opens the file, and puts the cursor on the line: funD().

Catching Compiler Errors

- The command, *error*, inserts compiler error messages into the source file.
- *error* will accept input from either standard input or a file.
- *error* must be executed with its standard input connected via a pipe to the error message source.
- *error* understands the error messages produced by: *make*, *cc*, *cpp*, *as*, *ld*, and *lint*.
- *error* touches source files only after all input has been read.
- Command format:

*error [–nsqv] [–***t** *suffixlist] [–***I** *ignorefile] [filename]*

Catching Compiler Errors

Because some language processors put their error messages on standard error and others on standard out, it is best to pipe these two together (|&) when using the *error* command.

The *–n* option tells *error* not to touch any files and all error messages are sent to the standard output. The *–q* option will inquire whether the files should be touched. *–v* invokes *vi* on each touched file. The *–t* option followed by a suffix list allows the user to specify which files are to be touched.
And, finally, the *–s* option prints out statistics regarding the error categorization.

Using *error*

```
patience% cat broken.c
main()
{
   printf ("next no. is %d \n", next_no())
   printf ("next no. is %d \n", next_no());
   printf ("next no. is %d \n", next_no());
   printf ("next no. is %d \n", next_no());
}

patience% cc broken.c
"broken.c", line 4: syntax error at or near variable name "printf"

patience% cc broken.c |& error -v
1 file contains errors "broken.c" (1)

File "broken.c" has 1 error.
        1 of these errors can be inserted into the file.
You touched file(s): "broken.c"

main()
{
   printf("next no. is %d\n",next_no())
/*###4 [cc] syntax error at or near variable name "printf"%%%*/
   printf("next no. is %d\n",next_no());
   printf("next no. is %d\n",next_no());}
~
~
~
~
~
"broken.c" 9 lines, 239 characters
```

Using *error*

Note that *error* inserts the error message into the file such that it follows the line on which the error occurs.

The Library Archiver: *ar*

Command Format:

ar [options] archive [member_file]

options	Meaning
r	Replace the named files in the archive.
v	Verbose. Produce messages indicating what is being performed by *ar*.
t	Table of contents of the archive file is displayed.
d	Delete a module from a library.
x	Extract a module from the library to a .o file of the same name.
q	Quick append to the end of library.

Command Format:

ranlib archive

The Library Archiver: *ar*

A library or archive file is a collection of files that have been merged together into a single file. With *ar*, you can create libraries, insert files into them, delete files from them, extract a file (copy it from the library to a *.o* file without removing it from the library), replace an archive file and perform other simple maintenance tasks. Archives are normally comprised of *.o* modules, but may comprised of any type of files such as source code or ASCII text. The archive header and the headers for the files are in a format which is portable across all machines. Hence, archive format is often used for distributing code over networks.

When you pass the *-l* flag to the linker, it looks in the directory */usr/lib* for a file by the name of *liblibrary.a*. To change the directory which is searched by the librarian, use the *-L* flag, i.e. *-L/usr/student* will cause the linker to look in */usr/student* for the library.

To create a library called *libstuff.a* from four object files: *one.o, two.o, three.o, and four.o.* Use this command format:

ar rv libstuff.a one.o two.o three.o four.o

This replaces files in the archive (if they are not already there) with all of the files which follow. In order to update a library, say if *one.o* is recompiled, you can update the version in the library with:

ar r libstuff.a one.o

Table of Contents

The *ranlib* command creates a table of contents, called __.SYMDEF, and places it at the beginning of the archive. By doing this, the archive may be linked more rapidly. In order to build a table of contents that the compiler recognizes in a file named *libstuff.a,* which can be randomly accessed by the linker, use the command:

ranlib libstuff.a

To view the table of contents of *libstuff.a*, use the command:

ar t libstuff.a

If you modify the contents of a library with *ar*, it is necessary to follow it with *ranlib* in order to update the table of contents as well.

The Library Archiver: *ar*

```
patience% ar rv libsample.a func1.o func2.o
a – func1.o
a – func2.o
ar: creating libsample.a

patience% ar t libsample.a
func1.o
func2.o

patience% nm libsample.a

func1.o:
00000000 T _func1
U _printf

func2.o:
00000000 T _func2
U _printf

patience% ar r libsample.a func3.o

patience% ar t libsample.a
func1.o
func2.o
func3.o
```

The Library Archiver: *ar*

Create a library using *ar*. We are working on the library sample which is in the file */libsample.a*. The flags *rv* specify that we are creating a new library. In it, we will create objects or replace objects that are already there. We wish the process to proceed in verbose mode (i.e. tell what is going on). As the objects are added, we get messages to that effect.

We get a list of the objects in the library with the *t* (table of contents) flag.

nm is an extremely useful problem-solving tool. It gives a name list of all the symbols found in a symbol table. It may be run on a program or a library and will give the value (if defined) and a letter describing the symbol (*T* stands for text segment symbol, *U* for undefined) as well as the actual name of the symbol. Note that C has prepended an underscore to each of our names.

We add a third object, *func3.o*, to our library and then immediately look at the table of contents *(ar t)* to see that it was indeed added.

Unlike *tar* format files, archive libraries can have tables of contents. *tar* can show the names of the archived objects only by scanning the entire archive.

You can run *nm* over an object file (or, in this case over a collection of archived object files) and get a more complete understanding of the symbols referenced in the file. The *T* indicates that the symbol following is a text segment symbol. The *U* before the *printf* indicates that the symbol is referenced here, but undefined. This is all right. It is defined in */libc.a*. Notice that C has prepended an underscore to the name you gave the function.

Adding another function with *ar q* appends that function to the archive file. We look again at the table of contents to verify that the file was indeed added.

Object Utility: *size*

```
patience% more func3.c
func3()
{
        func1();
        func2();
}

patience% cc -o scen1 scen.c func1.c func2.c func3.c

patience% size scen1
text    data    bss     dec     hex
16384   8192    3640    28216   6e38

patience% scen1
main runs
function 1 runs
function 2 runs
```

Object Utility: *size*

The size of the resulting executable is displayed with the *size* command. This displays the size of the text, data, and bss (uninitialized) areas of the executable in memory, and then decimal and hexadecimal totals of all three.

If we hand compile the program and include an object module which is not actually called, this additional module of executable code is included with the rest even though it can never be invoked. Hence, we get a slight increase in the size of the resulting executable. This is not a condition we would ordinarily desire and, in a larger program, it is very easy to do. To prevent this, as well as to eliminate the constant recompilation of common modules, libraries are used.

Object Utility: *nm*

- Command Format:

  nm [–options] [filename(s)]

–g	print only external symbols
–u	print only undefined symbols

- Example:

patience% **nm nextn**

```
00002298 t Fcrt1.o
00020000 d __DYNAMIC
00020098 D _edata
000200a0 B _end
00020088 D _environ
0000250c T _etext
000022a0 T _fun
000022c8 T _main
00002020 t crt0.o
00002298 T fsoft_used
000022a0 t nextn.fun.o
000022c8 t nextn.main.o
00002020 T start
00002298 T start_float
```

Object Utility: *nm*

nm

The Name List Printer, *nm* prints out a symbol table of the files passed to it on the command line. The *linker* has the task of scanning these symbol tables and resolving all external references. If a variable or function is used in a module but not defined there, the *linker* expects a definition of that object to be made somewhere else. If the *linker* cannot resolve the reference, it cannot build an executable and indicates which objects it couldn't find.

Each symbol in the file is preceded by its value (blank if undefined) an identification of what it is, whether it is local or external and where it is. If you run *nm* on a library file, you can find out just what objects it uses and creates.

Sample Compiling Session

```
patience% cc -c func1.c func2.c func3.c

patience% ar rv libsample.a func1.o func2.o func3.o
r - func1.o
r - func2.o
r - func3.o

patience% cc -o scen1 scen.c -L/home/sunray/student1\
-lsample
ld: /home/sunray/student1/libsample.a:
warning: archive has no table of contents;
add one using ranlib(1)

patience% size scen1
text    data    bss    dec    hex
16384   8192    3624   28200  6e28

patience% more scen2.c
#include <stdio.h>
main()
{
        printf("main runs \n");
        func3();
}
```

Sample Compiling Session

Rather than performing a forced link to the functions in the library, the linker pulls in only those functions which are actually called. The line following *-L* indicates where the libraries are to be sought (in addition to the default location */usr/lib*) and the line following the *-l* indicates that the previously-built library, "sample," is to be searched for unresolved references. Note that once we decide to archive our functions 1 through 3, we use only those versions in the library henceforth. A change to the source code of that function is not reflected in the library unless we compile it and add it explicitly. This can be a curse or a blessing. In many cases, the addition of a recompiled object to the appropriate library is made automatic via the makefile. In other cases, this can interfere with work others are doing and is avoided.

The non-fatal error message given by the linker is a result of not having executed *ranlib* on the archive. To avoid this problem and the problem of an out-of-date archive, run *ranlib* on archives after each modification. Note that the build of this identical program is back to its original size because the linker could find no reason to execute in *func3.*

Next, we see a slightly different version of *scen1* which should do the same thing. Function *func3* is revealed to be nothing more than a call to functions *func1* and *func2.*

Sample Compiling Session (con't.)

```
patience% cc -o scen2 scen2.c -L/home/sunray/student1\
 -lsample
warning: archive has no table of contents;
add one using ranlib(1)
Undefined:
_func1
_func2
*** Error code 1

patience% ranlib libsample.a

patience% cc -o scen2 scen2.c -L/usr/student -lsample

patience% scen2
scen2
main runs
function 1 runs
function 2 runs
```

Sample Compiling Session (con't.)

Next we build *scen2* (which actually uses *func3).*

The linker cannot find *func1* or *func2!* We know they are there but they can't be found. The problem is that the function *func3* contains references to functions that precede it physically in the archive.

To solve this problem, build a table of contents by running *ranlib* over the sample library. This makes the library contents available for random access by the linker rather than limiting it to sequential access. The solution in a case of a single library is indicated by a message from the linker.

More commonly you will be working with a large number of libraries which are grouped by common functions and created by someone other than yourself. There is no way that a table of contents can be created across different libraries and you may have to experiment with the order in which the libraries are called in order to resolve all symbol references.

Compiling FORTRAN and Pascal Code

```
patience% more fortest.f
      program fortest

      write (*,100) 'FORTRAN Speaking'
100   format (A)
      stop
      end
patience% f77 -o fortest fortest.f
patience% fortest
FORTRAN speaking


patience% more ptest.p
program ptest(output);
begin
  writeln ('Pascal speaking');
end.
patience% pc -o ptest ptest.p
patience% ptest
Pascal speaking
```

Compiling FORTRAN and Pascal Code

FORTRAN and Pascal programs are compiled by drivers similar to *cc*. Flags on the driver command line have similar effects.

In the first example, the first file is identified by the *.f* extension as a FORTRAN (Version 77) source code file which prints the words "FORTRAN speaking." If you have a FORTRAN compiler on your system you probably also have the FORTRAN preprocessor, RATFOR as well. RATFOR (Rational FORTRAN) compiles to standard FORTRAN augmented by structured control statements such as *while* and *switch*.

The program is compiled.

The program is invoked and runs.

The second example shows how, unlike standard Pascal, you may invoke *cpp* on your Pascal source program. The second file is identified by the *.p* extension as a Pascal program. It prints the words "Pascal speaking".

This program too, is compiled and run.

Compiling FORTRAN and Pascal Code

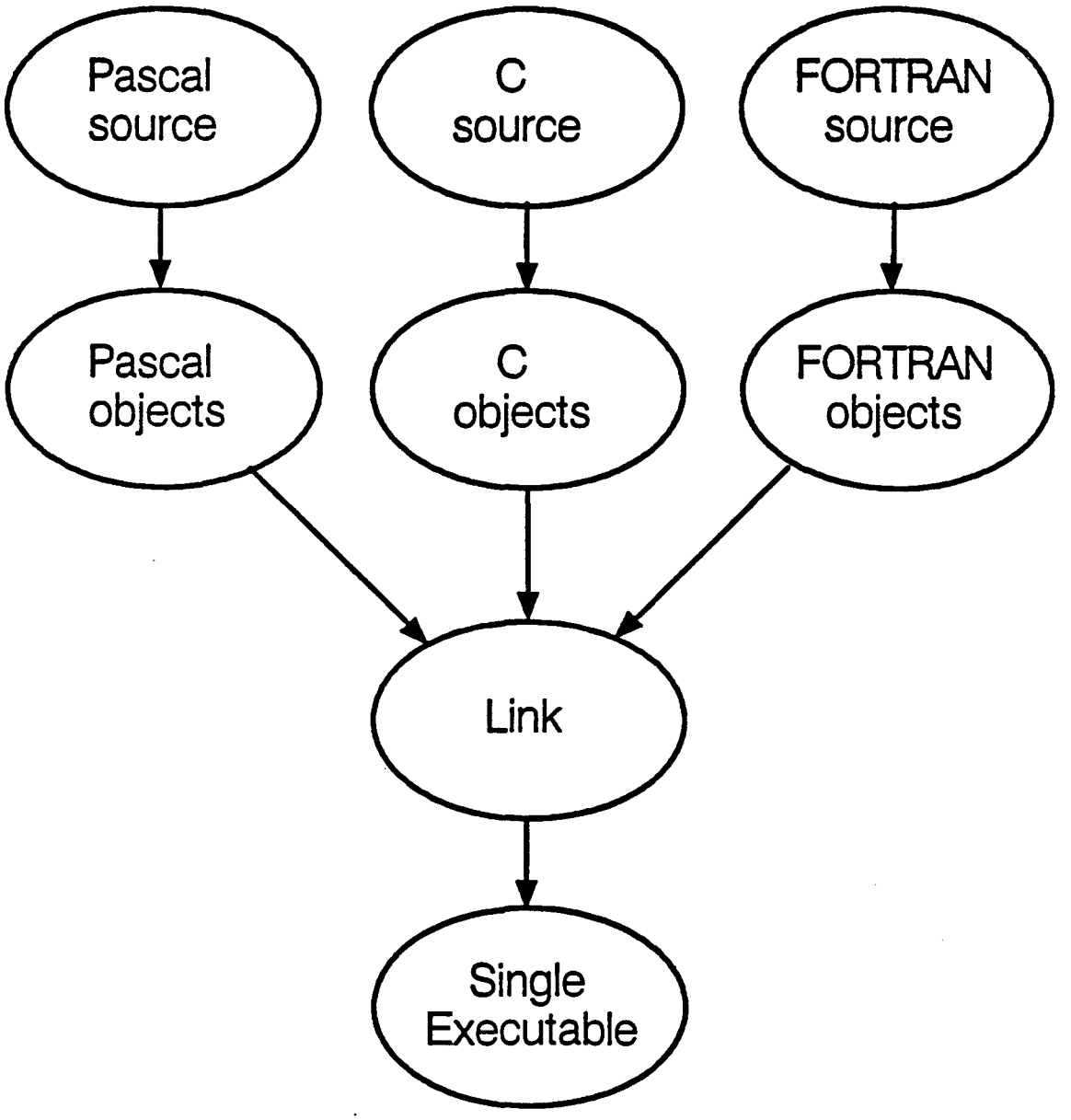
```
patience% more fstations.f
        function fstations (icount)
        integer icount
        write (*, 100) icount
100     format (//, i3, ' shiny new workstations left!')
        write (*, 200) icount
200     format (i3, ' shiny new workstations!;')
        icount = icount - 1
        write (*, 300)
300     format ('you pack one up, and ship it out!')
        write (*, 400) icount
400     format ('There''s', i3, ' shiny new workstations left!')
        return
        end

patience% more pstations.p
procedure pstations(Count:integer);
begin
  writeln(Count:3, ' shiny new workstations left!');
  writeln(Count:3, ' shiny new workstations!');
  writeln('you pack one up, and ship it out.');
  Count:=Count-1;
  writeln('There''s', Count:3, ' shiny new workstations left!');
end;
```

Compiling FORTRAN and Pascal Code

A more interesting sample FORTRAN function and Pascal procedure are shown here. Functionally they are identical and print modified lyrics to the tune of that well-known favorite "99 Bottles of Beer on the Wall".

Note that neither program fragment is sufficient to create an *a.out* file; they are functions. It is not uncommon to find well-debugged, documented and tested routines in Pascal or FORTRAN. FORTRAN in particular has been around so long in the research environment that many more libraries of mathematical, statistical, signal processing, and engineering routines may be available than are available in C.

General Concept for Merging Code



General Concept for Merging Code

Compiling FORTRAN, Pascal, and C Programs Together

```
patience% cat stations.c
main()
{
   static int wstation_count;
   wstation_count = 99;
   while (wstation_count > 1)
   {
      fstations_(&wstation_count);
      printf("\n\n");
      pstations(wstation_count);
      wstation_count--;
   }
   printf("And the backup is gone!\n");
}
```

```
patience% cc -g -c stations.c
patience% pc -g -c pstations.p
patience% f77 -g -c fstations.f
patience% cc -g -o stations stations.o pstations.o fstations.o
 -lpc -lF77 -lI77
```

Compiling FORTRAN, Pascal, and C Programs Together

We now see a main program written in C that uses these two functions. It has an entry point, sets the workstation count to 99 and calls each function once within the *while* loop. The three libraries specified are libraries required for Pascal and FORTRAN. We actually build a single executable from the three source files and the three libraries.

Thought for the day: How can you use *nm* and *grep* to determine the minimal set of libraries required for a successful link? (Hint: identify your external references and determine which include file each occupies).

Compiling FORTRAN, Pascal, and C Programs Together

```
patience% stations
   99 shiny new workstations left!
   99 shiny new workstations!;
you pack one up, and ship it out!
There's 98 shiny new workstations left!

   98 shiny new workstations left!
   98 shiny new workstations!
you pack one up, and ship it out.
There's 97 shiny new workstations left!

   97 shiny new workstations left!
   97 shiny new workstations!;
you pack one up, and ship it out!
There's 96 shiny new workstations left!
.
.
.
.
```

Compiling FORTRAN, Pascal, and C Programs Together

The program runs!

Module 9
make

Objectives: Upon completion of this module, the student should be able to:

- List and describe the components of a makefile: targets, dependencies, and rules.
- Describe how *make* processes a makefile.
- Write a simple makefile that will compile and link a C program.
- Describe the features that help make a makefile more elegant and flexible: macros, directives, special targets and implicit rules.
- Use these features to write a more flexible makefile.

Evaluation:

Perform Lab 7 to 90% proficiency.

Reference information:

- Appendix C, *Install.*
- Appendix D, *Source Code.*
- *Doing More with SunOS: Beginner's Guide* (p/n 800-1710-10), Section 6.4.
- *make User's Guide* in *Programming Utilities and Libraries* (p/n 800-1774-15).
- *SunPro, The Sun Programming Environment,* Sun Microsystems, Inc. (p/n FF144/20K).

Introduction to *make*

- Use *make* to automate complicated or repetitive tasks:
  - Update and maintain object libraries
  - Compile programs consistently
  - Run test suites
  - Install files onto a system or tape

Introduction to *make*

Make and Makefiles

make is a program that is often used to build object libraries and programs, and install systems. *make* reads highly-structured scripts named *makefiles* (or *Makefiles*) for instructions on how to perform its tasks. *make* is good for programs which are compiled from several modules and/or libraries because it simplifies the compilation process for the user; the user need only type in the command *make,* and *make* is faster since it only recompiles the pieces of code which have changed since the last compilation.

Preliminary Steps

- Identify the task you wish to perform. For example:

  Compiling and linking files (object files and library files)

- Determine the goal of the task:

  An executable (*stations*)

- Identify the components needed to accomplish the objective:

  stations.c, pstations.p, fstations.f,
  /usr/lib/libI77.a, /usr/lib/libF77.a, /usr/lib/libpc.a

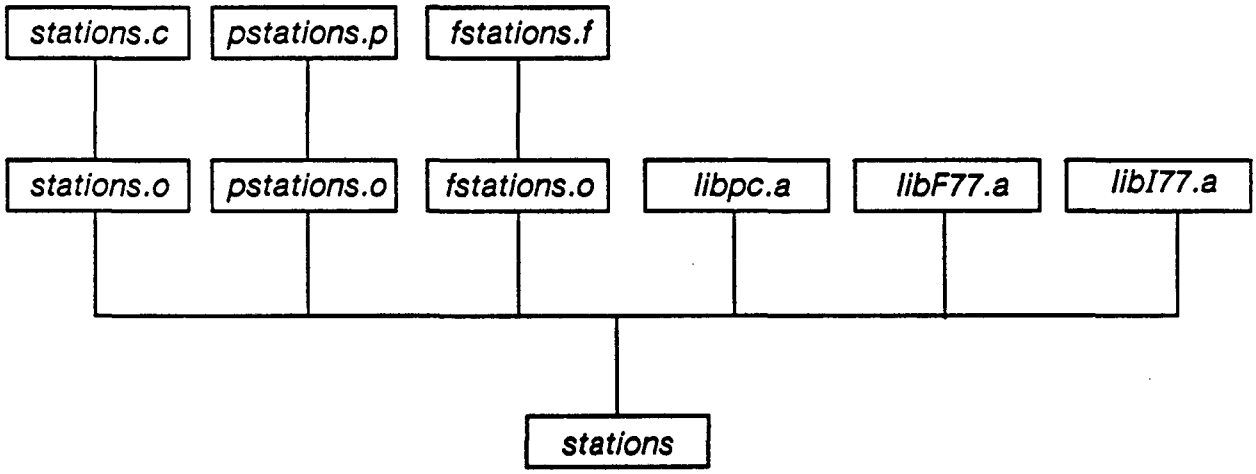
- Create a *Makefile*

Preliminary Steps

The repetitive task:

patience% **cc -c -g stations.c**
patience% **pc -c -g pstations.p**
patience% **f77 -c -g fstations.f**
patience% **cc -g -o stations stations.o pstations.o fstations.o -lpc -lF77 -lI77**

Required files to build stations:



Components of a *Makefile*

- A *Makefile* is made up of *targets* and *dependencies.*

- A *target* is a file to build (such as *pstations.o* or *stations)* or a command line to execute (such as *cc, ls, or touch*).

- A *dependency* is a file necessary to build the target (such as *pstations.p* or */usr/lib/libpc.a).*

- A target may have multiple dependencies.

- The list of commands that build the target is called a *rule. Rules* are of the form:

```
target. . .  :  [dependency]
   [command]
   . . .
```

Components of a *Makefile*

make applies the concepts of *targets* and *dependencies* to build a system. A target is made if it does not exist or is out of date. *make* determines if a target is out of date by comparing the date(s) of the target(s) to the dates of its dependencies. The target will be remade if a dependency is newer than the target. For example, since *stations* depends on *stations.c, stations* will be regenerated if *stations.c* is modified (because *stations.c* is newer than *stations*).

Format of Targets and Dependencies

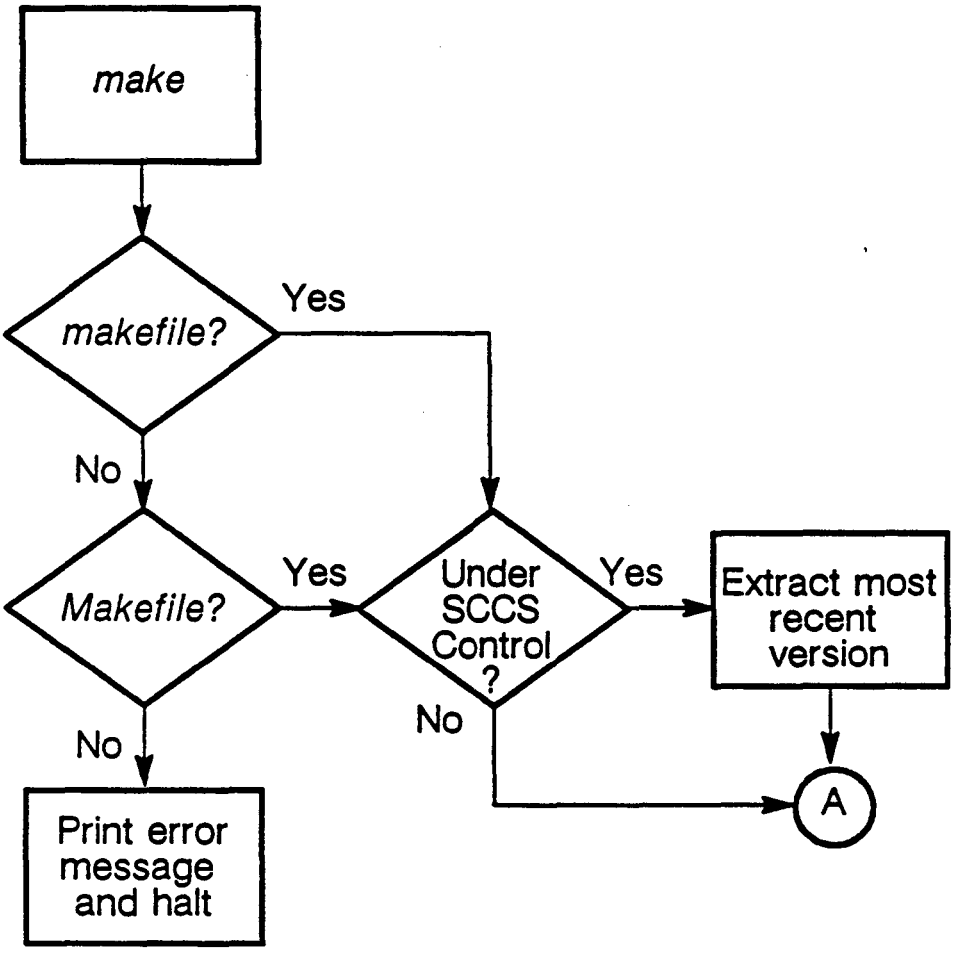
A dependency line in a makefile lists both targets and dependencies, separated by a colon. These entries define how to bring targets up to date when dependencies change.

There can be one or more targets and zero or more dependencies. There may be zero or more commands (also termed *rules*). Commands *(rules)* are omitted when they are unnecessary and when default commands built into *make* are sufficient. Default commands include predefined macros and additional rules, such as the rule for extracting SCCS files. These default commands and macros are presented later in this module. SCCS (Source Code Control System) is a file version maintenance utility that is presented in a later module.

Note that:

- A command must start with a TAB. A SPACE will not work.
- Lines that start with a # are treated as comments up until the next unescaped newline.
- A target is terminated by by the end of the file or the next non-blank line that does not begin with a TAB or a # (comment character).

An Overview of How *make* Works



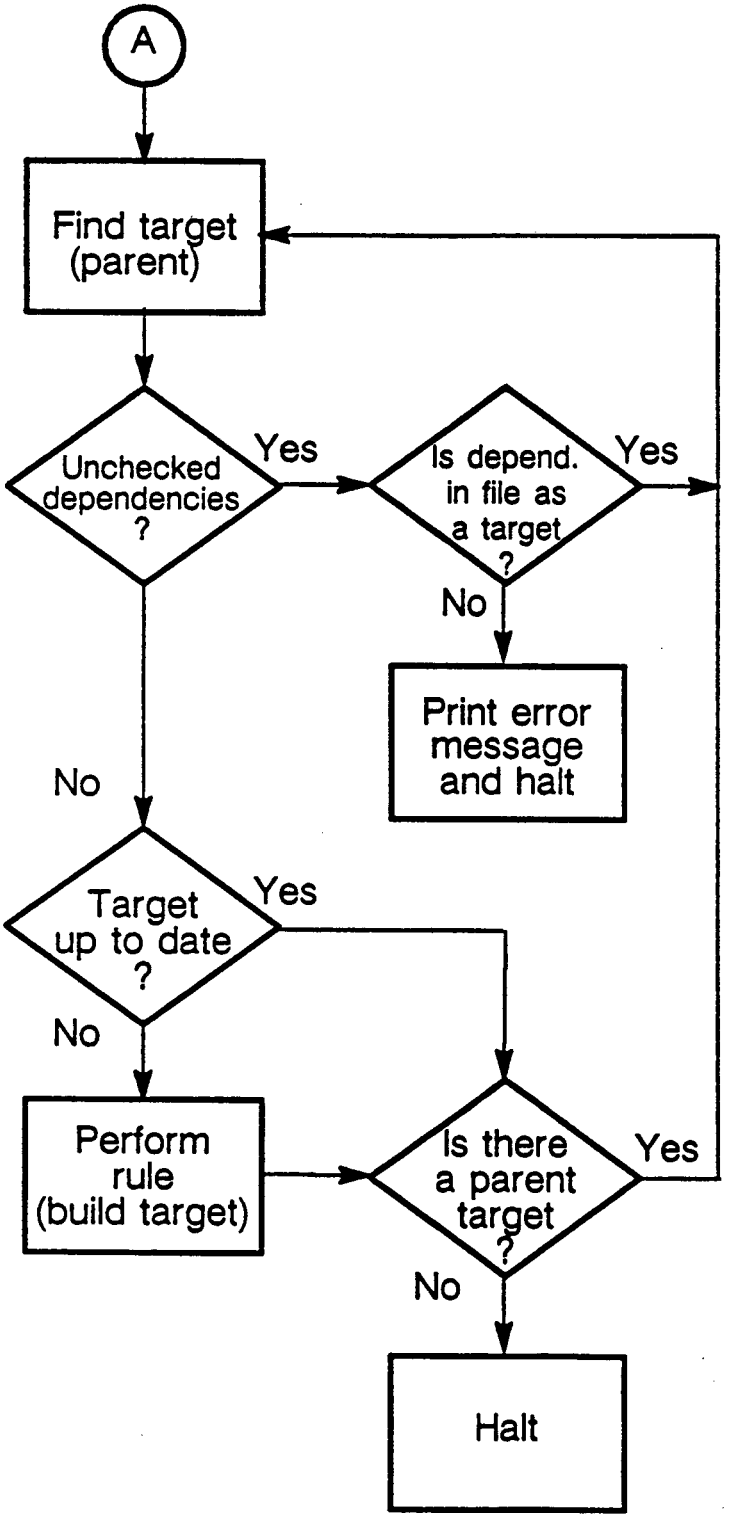
An Overview of How *make* Works

Executing *make* with no arguments tells *make* to first find a *makefile* (or *Makefile*), and begin dependency checking with the first target entry in that file. *make* displays each command it runs while building its targets. Note that it is possible to specify which target you want to build by placing the target name after the *make* command on the command line.

Trying to execute *make* with no arguments and when neither a *makefile* or a *Makefile* exists in the current directory, will cause an error condition and *make* will quit.

```
patience% make
make: Fatal error: No arguments to build
patience%
```

An Overview of How *make* Works (con't.)



An Overview of How *make* Works (con't.)

Executing *make* without any arguments will cause *make* to begin dependency checking with the first target. If the target has dependencies that have not been checked, *make* will check the dependencies first (and any dependencies they may have and so on) and then check the target against the dependencies and build it, if necessary.

A Simple *makefile*

```
patience% pwd
/home/sunray/student1/make_practice
patience% ls
Makefile        fstations.f     pstations.p
force_make      make_void       stations.c

patience% cat Makefile
batch: a b
        touch batch
b:
        touch b
a:
        touch a
c:
        echo "you don't see me"

patience% make
touch a
touch b
touch batch
patience%

patience% make
'batch' is up to date

patience% make c
echo "you don't see me"
you don't see me
patience%
```

A Simple *makefile*

*Makefile*s can contain commands as well as filenames. In this example, the makefile consists of several *touch* commands. *touch* updates the date/time modified of a file to the current date and time. If the file does not exist, *touch* creates it. *touch* is very useful with *make* since *make* recompiles/links objects that have date/times modified which are more recent than the date/time modified of the targets.

In the first example, the target *batch* depends on both *a* and *b*. *a* and *b* are dependencies. *make* checks to see that *a* and *b* are up to date. If they are, then *make* executes the *touch batch* command. If either *a* or *b* are not up to date, *make* executes the commands under the respective targets (*touch a* and *touch b*). The target *c* will not be executed unless you enter the command *make c* because *batch* does not depend on *c*.

The second example shows that the target is up to date.

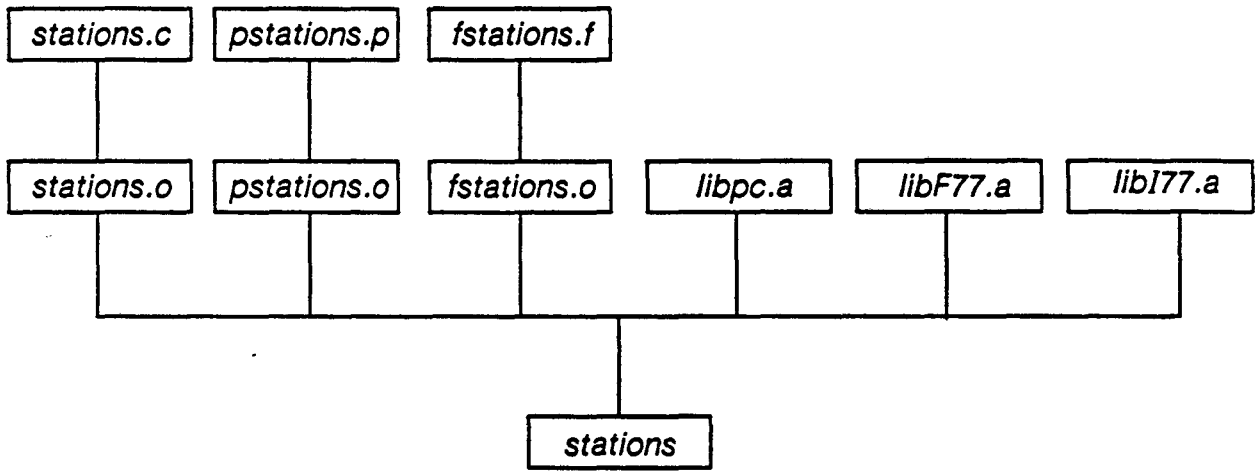
The final example shows how to specify a target on the command line.

Another Example: Building *stations* with *make*

```
patience% mv Makefile make.batch.save
patience% vi Makefile
    (create new makefile - see notes on next page)
patience% cat Makefile
stations: pstations.o  fstations.o  stations.o
    cc -g -o stations stations.o fstations.o pstations.o -lpc -lF77 -lI77

pstations.o: pstations.p
    pc -c -g pstations.p

fstations.o: fstations.f
    f77 -c -g fstations.f

stations.o: stations.c
    cc -c -g stations.c

patience% make
pc -c -g pstations.p
f77 -c -g fstations.f
fstations.f:
        fstations:
cc -c -g stations.c
cc -g -o stations stations.o fstations.o pstations.o -lpc -lF77 -lI77

patience%
```

Another example: Building *stations* with *make*

Required files to build stations:



Recall this dependency diagram from earlier in the module. Our goal is to create the executable *stations*. In order to build *stations*, we need the *.o* files and the library files. A preliminary *Makefile* may look something like:

```
stations: stations.o pstations.o fstations.o libpc.a libF77.a libI77.a
```

From this we decide that we can include instructions on how to build the *.o* files (with the *.c*, *.p*, and *.f* files), but the library files should not be on the dependency line because they already exist in */usr/lib*. Consequently, the *Makefile* would look something like:

```
stations:  stations.o pstations.o fstations.o
pstations.o:      pstations.c
fstations.o:      fstations.f
stations.o:       stations.c
```

Now we just need to add the commands to build the targets; as shown on the previous page.

More About *make* and Makefiles

- Options
- Line breaks
- Directives
- Macros
- Special targets
- Missing targets
- Implicit rules

More About *make* and Makefiles

Up to this point the module has covered the basics of creating makefiles and how *make* processes makefiles. But there is a lot more about *make* and *makefiles* that can make your tasks easier to accomplish. The topics listed on the opposite page are all features that you can use to make your *makefiles* more flexible and powerful. The remainder of this module is devoted to these topics.

make Options

- Command format:

  make [–f filename] [–dinpPs]

- Useful options:

–f filename	Specify the name of the makefile when it is not *makefile* or *Makefile*.
–d	Display reasons why a target was rebuilt.
–i	Ignore error codes returned by commands.
–n	Print commands but do not execute them.
–p	Print out the complete set of macro definitions and target descriptions.
–P	Report dependency hierarchy without rebuilding any targets.
–s	Do not print command lines before executing them (silent mode).

make Options

```
patience% cp Makefile newmakefile
patience% cat newmakefile

stations: pstations.o  fstations.o  stations.o
   cc -g -o stations stations.o fstations.o pstations.o -lpc -lF77 -lI77

pstations.o: pstations.p
   pc -c -g pstations.p

fstations.o: fstations.f
    f77 -c -g fstations.f

stations.o: stations.c
    cc -c -g stations.c

patience% make -f newmakefile -P
stations:       newmakefile pstations.o fstations.o stations.o
pstations.o:    newmakefile pstations.p
fstations.o:    newmakefile fstations.f
stations.o:     newmakefile stations.c

patience% make -f newmakefile -n
'stations' is up to date.
patience%  rm *.o stations
patience%  make -f newmakefile -n
pc -c -g pstations.p
f77 -c -g fstations.f
cc -c -g stations.c
cc -g -o stations pstations.o fstations.o stations.o -lpc -lF77 -lI77
patience%
```

Line Breaks

- Each command line in a makefile is executed in a separate shell.

```
patience% mv Makefile make.st.save
patience% vi Makefile
   (Create a new makefile)
patience% cat Makefile
test1:
        cd  /tmp;
        pwd

test2:
        cd  /tmp; pwd

test3:
        cd  /tmp; \
        pwd

patience% pwd
/home/sunray/student1/make_practice
```

Line Breaks

```
patience% make test1
cd  /tmp;
pwd
/home/sunray/student1/make_practice

patience% make test2
cd  /tmp; pwd
/tmp

patience% make test3
cd  /tmp; \
pwd
/tmp
```

As shown in the first example above (*make test1*), *pwd* indicates that we are still in student1's home directory, not */tmp* as expected. This is because each command line is executed in a newly spawned shell (Bourne is the default). A shell metacharacter is required on the command line when using built-in shell commands (*cd* in this example) to tell the shell that the command is not an executable, rather, it is a built-in command. This is why the first command line of the first rule ends with a semicolon. If the semicolon or other shell metacharacter was not present on the command line, *make* would try to execute the command directly (for better performance) but would not be able to locate an executable for *cd.*

The second example illustrates how to execute two (or more) commands in a single shell: separate the commands with semicolons. The third example shows an alternative to this. Escape the NEWLINE on a command line with a backslash (i.e. make sure the backslash is the last visible character on the line) and place the succeeding command on the next line (preceded by the usual TAB).

make Directives

@ don't echo command to screen

– continue execution even if particular command fails

```
patience% rm Makefile
patience% mv make.batch.save Makefile
patience% vi Makefile
        (Make some changes)
patience% cat Makefile
batch: c a b
        touch batch
b:
        rm adirectory
        touch b
a:
        touch a
c:
        echo "you don't see me"
```

```
patience% mkdir adirectory
patience% rm a b batch
patience% make
echo "you don't see me"
you don't see me
touch a
rm adirectory
rm: adirectory is a directory
*** Error code 1
make: Fatal error: Command failed for target 'b'
```

make Directives

make has two special directives used in *makefiles.* The @ directive tells *make* not to echo this command to the screen prior to execution (by default *make* echoes each line that it executes). The – directive tells *make* to continue even if the particular command fails.

```
patience% vi Makefile
        (Add directives)
patience% cat Makefile
batch: c a b
        touch batch
b:
        -rm adirectory
        touch b
a:
        touch a
c:
        @echo "you don't see me"


patience% rm a
patience% make
you don't see me
touch a
rm adirectory
rm: adirectory is a directory
*** Error code 1 (ignored)
touch b
touch batch

patience%
```

Macros: Another Look at Building *stations*

- Macro definition format:

  NAME=string

```
patience% rm -r Makefile a b batch adirectory
patience% mv make.st.save Makefile
patience% vi Makefile
    (Change file by including macros)
patience% cat Makefile
CFLAGS=         -c -g
OBJECTS=        pstations.o fstations.o stations.o
LIBS=           -lpc -lF77 -lI77

stations:       ${OBJECTS}
        cc -g -o stations ${OBJECTS} ${LIBS}

pstations.o:    pstations.p
        pc ${CFLAGS} pstations.p

fstations.o:    fstations.f
        f77 ${CFLAGS} fstations.f

stations.o:     stations.c
        cc ${CFLAGS} stations.c
patience% make
pc -c -g pstations.p
f77 -c -g fstations.f
fstations.f:
        fstations:
cc -c -g stations.c
cc -g -o stations pstations.o fstations.o stations.o -lpc -lF77 -lI77

patience%
```

Macros: Another Look at Building *stations*

Macro definitions

make has the ability to perform macro substitution within makefiles. The macro definitions are of the form *NAME=string* where *NAME* is the name of the macro and *string* is a string which will be substituted for *NAME*. This substitution occurs when *make* references the macro in the form *${NAME}* (parentheses or curly braces are required if the macro name is more than one character long). For example, the macro *OBJECTS* in this *makefile* is expanded to "*pstations.o fstations.o stations.o*" when *${OBJECTS}* is encountered by *make*. Similarly, the macro *CFLAGS* is expanded to "*-c -g*" when *${CFLAGS}* is encountered. Note that if a macro is undefined, *make* replaces references to it with an empty string. It is also possible to supply a definition for a macro from the command line:

```
patience% rm *.o stations
patience% make "CFLAGS= -c -O"
pc -c -O pstations.p
f77 -c -O fstations.f
fstations.f:
        fstations:
cc -c -O stations.c
cc -g -o stations pstations.o fstations.o stations.o -lpc -lF77 -lI77
```

Defining a macro from the command line overrides all other definitions for that macro.

Macros are useful to define strings in one place which are referenced from many places. This makes writing and modifying makefiles easier. Macros are also used to supply information to *make*. *make* itself has default definitions for many macros.

Predefined Macros

- Predefined macros for the C compiler:

```
CC              =>    cc

COMPILE.c       =>
        $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c

LINK.c          =>
        $(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)

CFLAGS

CPPFLAGS
```

NOTE: *make* uses the *arch(1)* command to determine the type of the current hardware architecture. The value returned by *arch* is substituted for the macro TARGET_ARCH.

Predefined Macros

make has some predefined macros for the various compilers, host architectures, and miscellaneous commands that it supports. For a complete list, refer to the *make man* page.

The Pascal macros are PC, PFLAGS, COMPILE.p, and LINK.p. The FORTRAN77 macros are FC, FFLAGS, COMPILE.f, LINK.f, COMPILE.F, and LINK.F. Both the Pascal and FORTRAN macros have expansions similar to the C macros.

Building *stations* Using Predefined Macros

```
patience% vi Makefile
        (Change file by including predefined macros)
patience% cat Makefile
CFLAGS= -g
PFLAGS= -g
FFLAGS= -g
CPPFLAGS=
LDFLAGS=
OBJECTS=  pstations.o fstations.o stations.o
LIBS=   -lpc -lF77 -lI77

stations:       ${OBJECTS}
        ${LINK.c} -o stations ${OBJECTS} ${LIBS}

pstations.o:    pstations.p
        ${COMPILE.p} pstations.p

fstations.o:    fstations.f
        ${COMPILE.F} fstations.f

stations.o:     stations.c
        ${COMPILE.c} stations.c
patience% rm *.o stations
patience% make
pc -g  -sun3 -c pstations.p
f77 -g  -sun3 -c fstations.f
fstations.f:
        fstations:
cc -g  -sun3 -c stations.c
cc -g   -sun3 -o stations pstations.o fstations.o stations.o -lpc -lF77 -lI77
patience%
```

Building *stations* Using Predefined Macros

Using the predefined macros make makefiles more elegant and flexible. Even though the macros LDFLAGS and CPPFLAGS are not defined, they are included for the sake of clarity. The *Makefile* would also execute successfully without these two lines because for undefined macros *make* simply substitutes empty strings.

Special-Function Targets

- *.IGNORE*
- *.SILENT*
- *.PRECIOUS*
- *.KEEP_STATE*

Special–Function Targets

Special–function targets are predefined target names that perform specific tasks. Only the most commonly used targets are discussed in this module. For a more complete list refer to the *make man* page.

The *.IGNORE* special–function target performs the same operation as the *–i* option to the *make* command; all error codes returned by commands are ignored.

The *.SILENT* special–function target performs the same operation as the *–s* option to the *make* command; none of the command lines are printed prior to their execution. Use *.IGNORE* and *.SILENT* in a makefile if you always want to ignore error codes or turn off command echoing. Otherwise, use the options to *make* for specific runs.

Use the special–function target *.PRECIOUS* to identify the files you do not want *make* to remove when interrupted. Normally, if *make* is interrupted when processing a target, the target file is removed. This is useful because any incomplete files with new modification dates will not be left behind. However, in the case of library files, it is often better to leave the target intact. A subsequent run of *make* on library files will resume where the previous one stopped.

The *.KEEP_STATE* special–function target is a very useful tool in makefiles. When this target exists in a makefile, *make* not only checks each target file against it dependencies, it compares each command line with the corresponding command line it ran the last time it built the target. If the command line has changed, *make* rebuilds the target. *make* keeps track of command line execution in the file *.make.state* in the current directory.

.KEEP_STATE
stations Revisited

```
patience% cat Makefile
CFLAGS= -g
PFLAGS= -g
FFLAGS= -g
CPPFLAGS=
LDFLAGS=
OBJECTS=  pstations.o fstations.o stations.o
LIBS=   -lpc -lF77 -lI77

stations:       ${OBJECTS}
        ${LINK.c} -o stations ${OBJECTS} ${LIBS}

pstations.o:    pstations.p
        ${COMPILE.p} pstations.p

fstations.o:    fstations.f
        ${COMPILE.F} fstations.f

stations.o:     stations.c
        ${COMPILE.c} stations.c

patience% rm *.o stations
patience% make stations.o
cc -g   -sun3 -c stations.c

patience% make "CFLAGS= -O" stations.o
'stations.o' is up to date.
```

.KEEP_STATE
stations Revisited

The previous example shows that even if we change the arguments to one of the macros (in this case *CFLAGS*), *make* considers the target file, *stations.o*, up to date. This is because *make* only performs dependency checking (i.e. *stations.o* is newer than *stations.c*) unless otherwise specified. By including the *.KEEP_STATE* target in the makefile, we can successfully rebuild targets even though they may be newer than their dependencies; IF the command line(s) to build the targets have changed. To suppress command dependency checking for a specific command line, insert a question mark as the first character after the TAB.

.KEEP_STATE
stations Revisited (con't.)

○ Utilizing *.KEEP_STATE*:

```
patience% rm *.o
patience% vi Makefile
    (Add .KEEP_STATE)
patience% cat Makefile
CFLAGS= -g
PFLAGS= -g
FFLAGS= -g
CPPFLAGS=
LDFLAGS=
OBJECTS=  pstations.o fstations.o stations.o
LIBS=   -lpc -lF77 -lI77

.KEEP_STATE:

stations:       ${OBJECTS}
        ${LINK.c} -o stations ${OBJECTS} ${LIBS}

pstations.o:    pstations.p
        ${COMPILE.p} pstations.p

fstations.o:    fstations.f
        ${COMPILE.F} fstations.f

stations.o:     stations.c
        ${COMPILE.c} stations.c
patience% make stations.o
cc -g  -sun3 -c stations.c
patience% make "CFLAGS= -O" stations.o
cc -O  -sun3 -c stations.c
patience%
```

.KEEP_STATE
stations Revisited (con't.)

○ Suppressing command dependency checking for a specific command:

patience% **cp Makefile test**
patience% **vi test**
(Add a "?" as the first character after the tab in the rule for *stations.o*)
patience% **cat test**

```
CFLAGS= -g
PFLAGS= -g
FFLAGS= -g
CPPFLAGS=
LDFLAGS=
OBJECTS=  pstations.o fstations.o stations.o
LIBS=   -lpc -lF77 -lI77

.KEEP_STATE:

stations:       ${OBJECTS}
        ${LINK.c} -o stations ${OBJECTS} ${LIBS}

pstations.o:    pstations.p
        ${COMPILE.p} pstations.p

fstations.o:    fstations.f
        ${COMPILE.F} fstations.f

stations.o:     stations.c
        ?${COMPILE.c} stations.c
```

patience% **rm *.o**
patience% **make -f test stations.o**
cc -g -sun3 -c stations.c
patience% **make -f test stations.o**
'stations.o' is up to date.

.KEEP_STATE and Hidden Dependencies

- Hidden dependencies are those files that are not explicitly listed as sources on the compilation command line.

- When *.KEEP_STATE* is in effect, the names of the hidden dependencies will be maintained in the *.make.state* file.

.KEEP_STATE and Hidden Dependencies

Hidden dependencies are usually the files listed with *#include* directives in C source files. The target in a makefile depends just as much on these files as it does on those that are explicitly listed in the makefile. By utilizing *.KEEP_STATE* in your makefile, you will insure that the dependency list for each target is always accurate and up to date and you will not have had to explicitly list or generate dependency lists for included files.

Missing Targets and Dependencies

- Missing target

```
patience% make itup
make: Fatal error: Don't know how to make target 'itup'
```

- Empty rule

```
patience% cat make_void
void:
patience% make -f make_void
patience%
```

- Forcing a target's rule to be executed

```
patience% cat force_make
# null rule
test: FORCE
   echo "always executed"
FORCE:

patience% make -f force_make
echo "always executed"
always executed
```

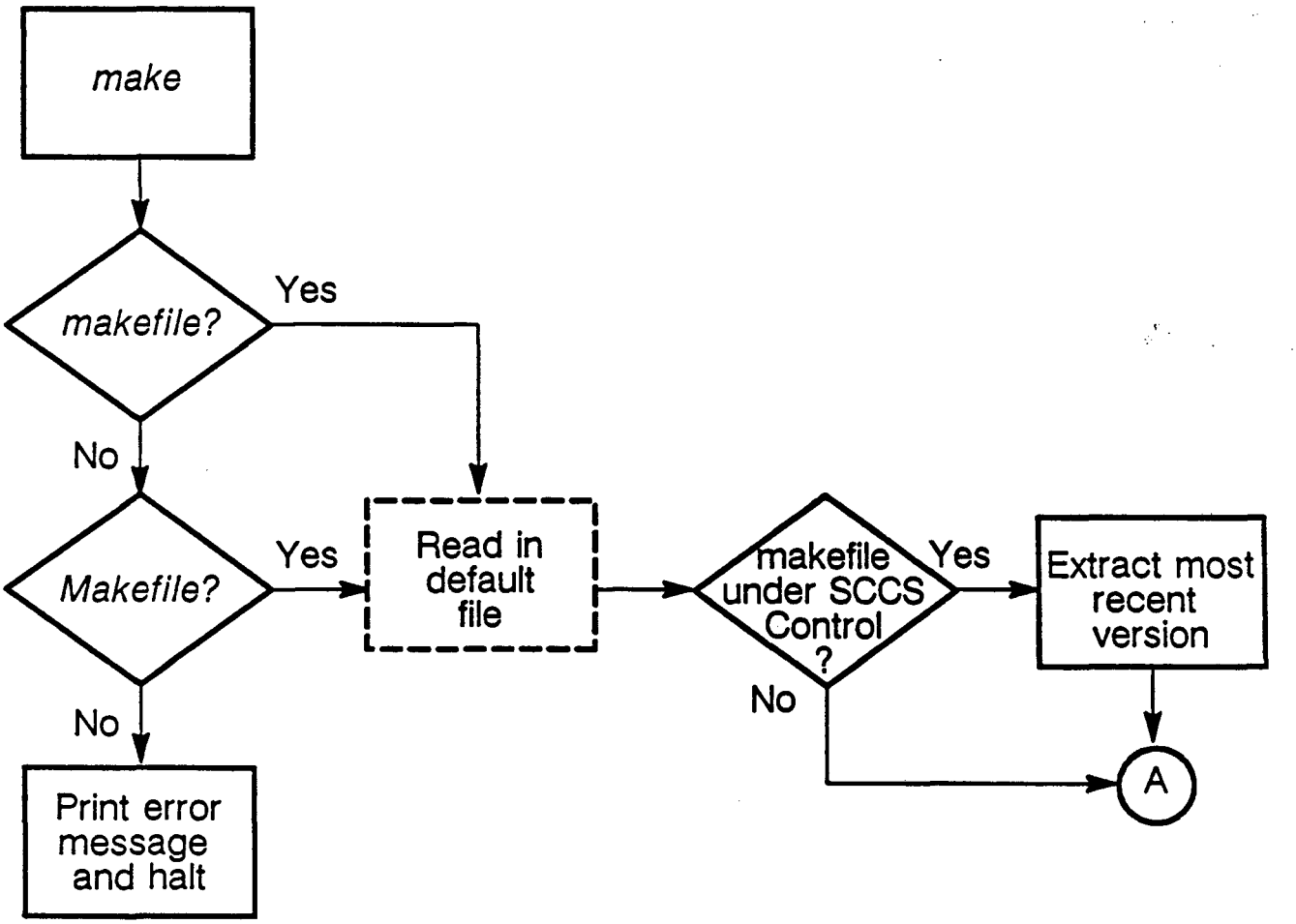
Missing Targets and Dependencies

If a target is specified on the command line (*make itup*) or in a dependency list and it is missing, has no target entry, there is no implicit rule to build it, or an SCCS file to extract it from, *make* will produce an error message.

When you want to force *make* to execute a target's rule, use the empty (null) rule as shown on the previous page. *make* finds the target entry for the dependency (*FORCE*), and executes the empty (null) rule. Since this is successful, *make* considers the dependency newer than the target (even though no specific file exists) and goes on to build the target (*test*).

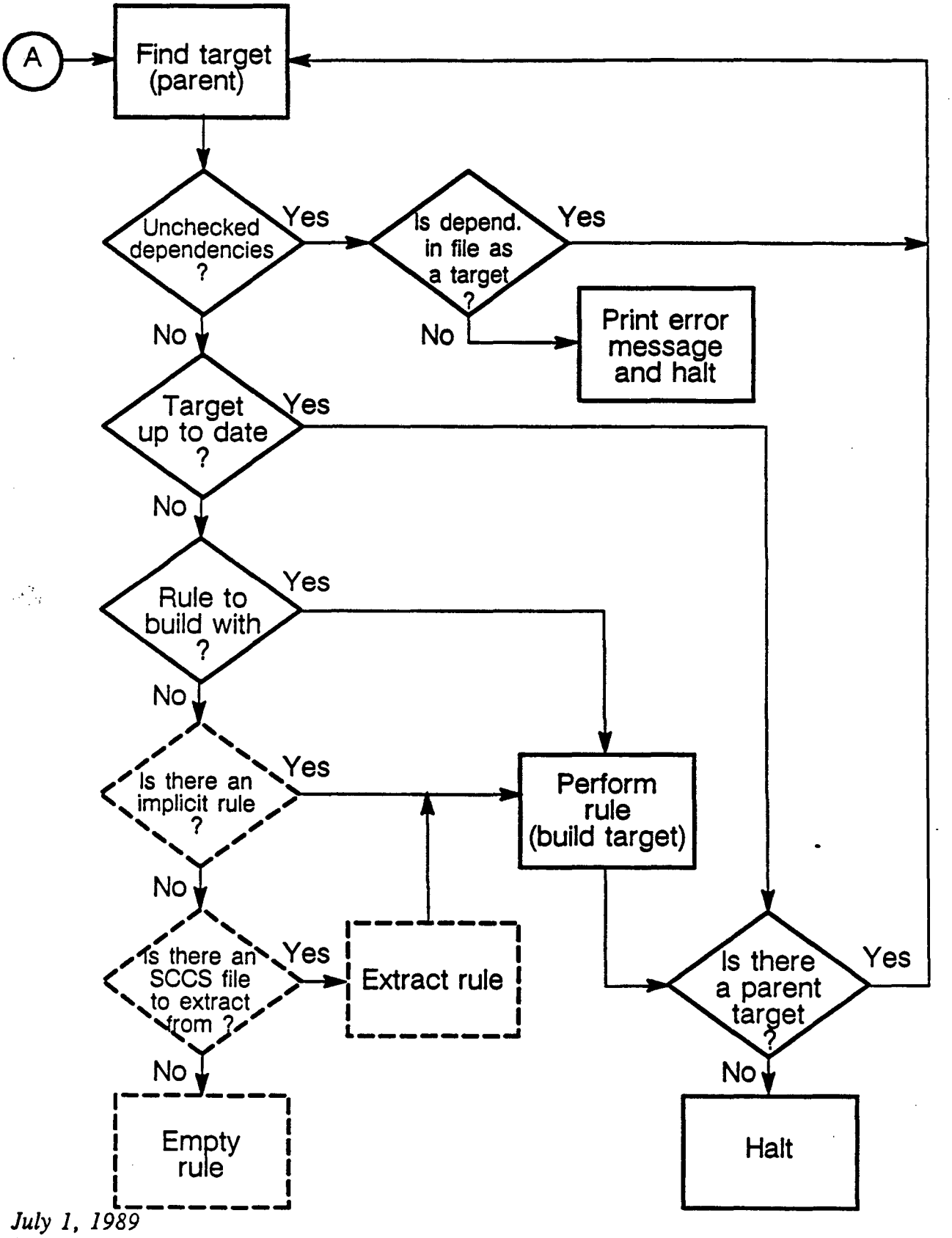
Implicit Rules

- Implicit rules exist in the file */usr/include/make/default.mk* which *make* reads in addition to the user's makefile.
- If the user omits a target entry for a specified target, *make* will consult the default file and attempt to find a rule for building it.



NOTE: The steps surrounded by dotted lines were not shown in the overview flow chart at the beginning of the module.

Implicit Rules



Implicit Rules (con't.)

- Implicit rules contain instructions for creating executables from C, Pascal, FORTRAN, modula–2, lex, yacc, and assembler language source files.

- There are two types of implicit rules:
  - Suffix rules (default rules)
  - Pattern–matching rules

Implicit Rules (con't.)

make contains many built–in rules to aid in the compilation of programs. It is not necessary to specify a command to compile programs within a makefile when default rules suffice. Other commands, such as *install,* must be explicitly included in the makefile if you want to use them.

The suffix rules are commands on how to build a file with the same basename but a different suffix. Utilizing these rules will sometimes eliminate the need for a makefile. For example, the suffix rules include the commands necessary to build a *.o* file from a *.c* file (called a "dot–c–dot–o" rule) and an executable file from a .c file (called a "dot–c" rule). Assuming there is a file in the current directory called *atest.c*, *make* will perform the following:

```
patience% make atest.o
cc   –sun3 –c  atest.c
patience% make atest
cc    –sun3 –o atest atest.c
```

Pattern–matching rules select a rule based on a target and dependency matching a certain wild–card pattern. For more information refer to the *make User's Guide.*

Module 10
Debugging

Objectives: Upon completion of this module, the student should be able to:

- Use the execution trace command: *ctrace.*
- Describe the appropriate executable file format and core file format, given a specific magic number.
- Debug C programs using *dbx.*
- Debug C programs using *dbxtool.*

Evaluation:

Perform Lab 8 to 90% proficiency.

Reference information:

- Appendix D, *Source Code.*
- *Programming Debugging Tools* (p/n 800-1775-10).
- *The UNIX System: A Sun Technical Report,* Sun Microsystems, Inc. (p/n 800-1419-02).
- *SunPro, The Sun Programming Environment,* Sun Microsystems, Inc. (p/n FF144/20K).

Tracing the Execution of a C Program with *ctrace*

- Only available with System V.
- *ctrace* traces the execution of a C program.
- *ctrace* accepts input from a file or from standard input.
- *ctrace* will print the text of each executable statement and the values of the variables referenced and/or modified.
- Loops are detected and tracing is stopped after a couple of iterations until the loop is exited or a different sequence of commands is executed within the loop. To help detect infinite loops, a message will be printed after every 1000 iterations.
- Output defaults to standard out.
- Command format:

*ctrace [–***f** *functions] [–***v** *functions] [–oxue] [filename]*

Tracing the Execution of a C Program with *ctrace*

By default, *long* and pointer variables will be printed as signed integers. The *char*, *short*, and *int* variables will also be printed as signed integers, or characters, if appropriate. *double* variables are printed as floating–point numbers in scientific notation. To change the defaults, use the *–o* (octal), *–x* (hexadecimal), *–u* (unsigned), or *–e* (floating point) flags. Use the *–f* option to specify which functions to trace or use *–v* to trace all but the specified functions.

Tracing a C Program

```
main()
{
   char *reverse();
   char *pointer;
   char inputstr[20];
   char result[20];

   printf("\nPlease enter a string (19 chars or less):  ");
   gets(inputstr);
   pointer = reverse(inputstr, result);
   printf("\nThe string is:  %s", pointer);
   printf("\n");
}


char *reverse(string, res)
char *string;
char res[20];
{
   int count;
   char *ptr;

   ptr = string;

   while (*ptr != NULL)
     ptr++;

   --ptr;

   for (count = 0; ptr - string > 0; ptr--, count++)
      res[count] = *ptr;   return(res);
}
```

Tracing a C Program

The program in this example is *reverse.c* which takes as input a string of 19 characters or less, reverses the string, and prints out the results. To compile and execute it, perform the following:

patience% **cc reverse.c -o reverse**

patience% **reverse**

Please enter a string (19 chars or less): *hello there*

The string is: ereht olle] 6

Well, the program almost works! The "h" is missing and there are some extra characters appended to the string. Using *ctrace* may help discover where the problem is in the code.

Using *ctrace*

```
 4 main()
 11    printf("\nPlease enter a string (19 chars or less): ");
Please enter a string (19 chars or less):   hello there
 12    gets(inputstr);
       /* inputstr == "hello there" */
 13    pointer = reverse(inputstr, result);
       /* inputstr == 251657460 or "hello there" */
       /* result == 251657440 */
 20 reverse(string, res)
 27    ptr = string;
       /* string == 251657460 or "hello there" */
       /* ptr == 251657460 or "hello there" */
 29    while (*ptr != NULL)
       /* *ptr == 104 or 'h' */
 30       ptr++;
          /* ptr == 251657461 or "ello there" */
 29    while (*ptr != NULL)
       /* *ptr == 101 or 'e' */
 30       ptr++;
          /* ptr == 251657462 or "llo there" */
    /* repeating */
    /* repeated 9 times */
 29    while (*ptr != NULL)
       /* *ptr == 0 */
 32    --ptr;
       /* ptr == 251657470 or "e" */
 34    for (count = 0; ptr - string > 0; ptr--, count++)
       /* count == 0 */
       /* ptr == 251657470 or "e" */
```

Using *ctrace*

Execute the following to create the traced execution:

patience% **ctrace reverse.c > rtrace.c**

patience% **cc rtrace.c -o rtrace**

patience% **rtrace**

The first command redirects the output from *ctrace* to the file *rtrace.c.* The second step compiles the output from *ctrace* and puts the executable into *rtrace.* If you wonder why this takes two steps instead of one, piping the output of *ctrace* into *cc,* it is because *cc* does not allow the use of pipes.

After entering *rtrace* the traced execution begins. The program stops at the *gets* waiting for user input. After the user enters the string "hello there" the program finishes executing (continued on the next page).

The output may look confusing at first. The numbered lines represent the executable code in the program and the information within the comment delimiters represent the value of the variables that are currently being referenced and/or modified.

The line after 12 tells us that the "hello there" was read in correctly. At line 20 the program enters function *reverse* and, after a couple of assignments, enters the while loop to find the end of the string. Note that after two iterations of the loop, tracing is discontinued. *ctrace* does note that it repeats the loop and how many times. At line 32 the variable *ptr* is decremented to point to the last character of the string and then enters a *for* loop that is responsible for putting the string, in reverse, into an array.

Using *ctrace* (con't.)

```
     /* string == 251657460 or "hello there" */
35       res[count] = *ptr;
         /* count == 0 */
         /* *ptr == 101 or 'e' */
         /* res[count] == 101 or 'e' */
34    for (count = 0; ptr - string > 0; ptr--, count++)
     /* ptr == 251657469 or "re" */
     /* count == 1 */
     /* string == 251657460 or "hello there" */
35       res[count] = *ptr;
         /* count == 1 */
         /* *ptr == 114 or 'r' */
         /* res[count] == 114 or 'r' */
   /* repeating */
   /* repeated 8 times */
34    for (count = 0; ptr - string > 0; ptr--, count++)
     /* ptr == 251657460 or "hello there" */
     /* count == 10 or '\n' */
     /* string == 251657460 or "hello there" */
37    return(res);
     /* (res) == 251657440 */
     /* pointer == 251657440 */
14    printf("\nThe string is:  %s", pointer);
     /* pointer == 251657440 */
The string is:  ereht olle] 6
15    printf("\n");

   /* return */
```

Using *ctrace* (con't.)

Line 35 puts the contents of the pointer into the array. In this first instance the first element of the array will contain the last character of the string, or "e". The second iteration of the *for* loop is shown and the second element of the array is assigned the letter "r". *ctrace* shuts off tracing for the next eight iterations. Just after this *ctrace* notes that both *ptr* and *string* point to the string "hello there". At line 37 the result is returned to *main* but the *printf* at line 14 does not print the expected string.

Take a closer look at the *for* loop. *ctrace* traced the first two iterations for the letters "e" and "r". The next eight iterations *ctrace* tells us it made would take us to the letter "e" in "hello", not the letter "h", as we might expect. Because the *for* loop also decrements *ptr* it is evident that *ptr* does point to the beginning of the string but that the "h" is not actually put into the array. This implies that the condition statement of the *for* loop needs to be changed from "ptr – string > 0" to "ptr – string >= 0". To clean up the rest of the garbage, we can use *bzero* to initialize the array before using it.

After making these two changes and recompiling, we can execute the updated code:

patience% **reverse**

Please enter a string (19 chars or less): *hello there*

The string is: ereht olleh

Miscellaneous Error Handling Techniques

Call Format:

```
fprintf (stderr, "Unknown host:  %s\n", hostname);
```

Miscellaneous Error Handling Techniques

fprintf is commonly used for error reporting. It allows redirection of error output from the shell to keep it distinct from data.

```
fprintf (stderr, "Unknown host: %s\n", hostname);
fprintf (stderr, "Could not discover the answer!\n");
fprintf (stderr, "Could not get %s's status.\n", argv[1]);
```

When capturing error output in a file, it is helpful to include with the message the name of the program, the routine which failed, and a time stamp.

It is also helpful to be able to activate and deactivate debugging messages and log file output with command line switches, environment variables, compile time toggles, or startup files. You must consider the long–term usefulness to others of internal error reporting and logging to determine how to turn on such special error reporting mechanisms. Printing information about the state of the program remains the most common starting point for all debugging efforts.

Core Dumps

Reasons for Dumping Core:

- Using null pointers
- Branching to a Data Area
- Growing Stack too Far
- Demanding Services beyond Limits
- Forced *abort()*

Tools to Analyze Core Dumps:

- *dbx*
- *dbxtool*

Core Dumps

Program crashes are usually due to exceptions such as referencing non-existent memory, trying to write to protected memory, trying to execute a non-existent statement, or otherwise demanding services above your limits. SunOS usually preserves an image of core at the time of the exception (termed a *core dump*) in a file named *core,* in the current directory.

To force a program crash you can kill a process from the controlling window with the quit character. This character can be selected with the *stty* command; <CNTRL-C> and <DEL> are common choices. It may be necessary to kill the process from another window when it doesn't respond to input from the controlling window, using the *kill* command:

kill -QUIT process_id

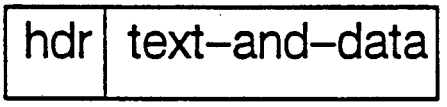
-QUIT is the mnemonic for one of the signals that can be sent to the process. Unless trapped, most signals cause the program to crash. However, there is one signal, *-KILL*, which cannot be trapped and hence always works. See *signal(), kill* and *ps* for details.

A core dump can also occur when a program detects an internal inconsistency which causes it to execute an *abort()* . A similar case is when the process receives certain signals which it neither catches or ignores. These signals include: SIGQUIT, SIGKILL, SIGTRAP, SIGABRT, SIGEMT, SIGFPE., SIGBUS, SIGSEGV, SIGSYS, SIGLOST. Examining postmortem core dumps using *dbx(tool)* is a helpful debugging technique.

Executable Formats and Core Dump Formats

407

a.out



core

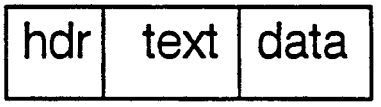
410

a.out

core

413

a.out



core

Executable Formats and Core Dump Formats

The earlier module on compiling and linking introduced the three possible formats of an executable: 407, 410, and 413. All three are comprised of a header followed by text and data areas. The text and data area for the first format, 407, is combined. The text and data areas for the remaining two formats are separate. Regardless of the type of an executable, it contains instructions on how to build an instance of a process running the program in the file.

A core file, however, represents the state of the program at a given time. The information in the core image changes as the program runs. In general, the program text does not change, but the variables comprising the data do.

Executable Headers and Core Headers

- Core headers contain the following information:
  - Magic number
  - General purpose registers
  - Executable header
  - Killing signal, if any
  - Text size, in bytes
  - Data size, in bytes
  - Stack size, in bytes
  - Command name

- Executable headers contain the following information:
  - Machine type
  - Magic number
  - Size of text segment
  - Size of initialized data
  - Size of uninitialized data
  - Size of the symbol table
  - Entry point

Executable Headers and Core Headers

The header portion of executable files and core files contain a variety of information. The general purpose registers contain information describing the status of the machine at the time of the crash. Other information, such as killing signal, text size and stack size, are also used by *dbx(tool)* to examine the state of the program when it crashed.

Debugging Using *dbx*

Features of *dbx:*

- Powerful symbolic debugger.
- *dbxtool* is *dbx* with an enhanced, graphical interface.
- Perform either *post–mortem analysis* or activate *live* debugging session.

Debugging Using *dbx*

dbx (and its SunView equivalent, *dbxtool*) are powerful symbolic debuggers which are well integrated with the SunOS programming environment. *dbxtool* is *dbx* with an enhanced, graphical interface. The remainder of this module explains the commands available under both programs.

The *dbx* debugger manages the process you are debugging as a subprocess to itself. It is possible to either perform a post–mortem analysis using the core file generated when a program aborts, or start a live run under *dbx*.

Assuming that you have caused the linker to leave the symbol table intact (by passing it the *–g* flag), *dbx* can search, list and evaluate even complex C structures. It understands the high level languages used under SunOS, and can edit and compile programs without leaving the debugger.

dbx Command Summary

```
patience% dbx
(dbx) help
    Command Summary

Execution and Tracing
   catch       ignore      run         stop
   clear       next        status      trace
   cont        rerun       step        when
   delete

Displaying and Naming Data
   assign      dump        undisplay   where
   call        print       up          whereis
   display     set81       whatis      which
   down

Accessing Source Files
   cd          func        pwd         /
   edit        list        use         ?
   file

Miscellaneous
   alias       detach      make        sh
   dbxenv      help        proc        source
   debug       kill        quit

Dbxtool
   button      toolenv     unbutton    unmenu
   menu

Machine Level
   nexti       stepi       stopi       tracei

The command 'help <cmdname>' provides additional
help for each command
(dbx)
```

dbx Command Summary

Running dbx

dbx is normally run by typing *dbx program* on the command line. You can specify a process ID to attach *dbx* to an already–running process, or specify the program you wish to debug from within *dbx* using the *debug* command. If a core file is present in the current directory, *dbx* displays source at the location of the crash.

Until you are familiar with *dbx*, *help* is the most common command you will use. *dbx* has extensive built–in help functions. *help* without a command name displays a list of commands. *help command* displays the syntax of the command.

Debugging Using *dbx*

stop at <line> [<if cond>]	– Stop execution at the line
stop in <proc> [<if cond>]	– Stop execution when <proc> is called
stop <var> [<if cond>]	– Stop when value of <var> changes
	NOTE: Optional <if cond> causes execution to stop only if condition <cond> is true when appropriate stopping point is reached
	eg: stop at 100 if i == 5
stop <if cond>	– Stop if condition true
list	– List 10 lines
list <first>, <last>	– List source lines from <first> to <last>
list <proc>	– List the source to <proc>
status	– Print trace's, when's, and stop's in effect
status > <filename>	– The same, but redirect output into file <filename>
	eg: status > foo
clear	– Clear all breakpoints at the current stopping point
clear <lineno>	– Clear all breakpoints at <lineno>
delete <number> ...	– Remove trace's, when's, or stop's of given number(s)
delete all	– Remove all trace's, when's, and stop's
run	– Begin execution of the program with the current arguments
run <args>	– Begin execution of the program with new arguments

Debugging Using *dbx*

Breakpoints

You set breakpoints using the *stop* command. The main variants are *stop in* and *stop at.* The command *stop in* is followed by a function name. This sets a breakpoint at the first executable statement in that function. The *stop at* command is followed by a line number. To find out the line number, use the *list* command. To list a range of lines, provide *dbx* a starting and stopping line number. The default use of *list* (no arguments) is to list the next ten lines starting at the current location.

Breakpoints can be displayed by using the *status* command and cleared by using the *clear* command if you are stopped at that line. Otherwise you must use the optional line number argument. The *delete* command with the event number given by *status* will also remove a breakpoint. The *clear* command does nothing if the execution of the program is not currently stopped on a line where the breakpoint was set. Once you have a breakpoint in place, it is safe to run the process using the command *run*. If arguments are provided to this command, they are passed through to the program you are debugging as though passed on the command line. The count of these arguments is put in *argc* and pointers to the text are in *argv.*

Debugging Example

patience% **cc –g dbx.demo1.c –o dbx.demo1**

patience% **cc –g dbx.demo2.c –o dbx.demo2**

patience% **dbx.demo1**

est

patience% **dbx dbx.demo1**

Reading symbolic information...
Read 39 symbols

(dbx) **run**

Running: dbx.demo1
est

execution completed, exit code is 0
program exited with 0

(dbx) **stop at 4**

(2) stop at "/home/sunray/student1/dbx.demo1.c":4

(dbx) **status**

(2) stop at "/home/sunray/student1/dbx.demo1.c":4

Debugging Example

patience% **cat dbx.demo1.c**

```
/*      dbx.demo1.c     */
char c_array[] = "test";
main()
{       putchar(c_array[1]);
        putchar(c_array[2]);
        putchar(c_array[3]);
        putchar(c_array[4]);
        putchar('\n');
}
```

Debugging Example (con't.)

```
(dbx) run
Running: dbx.demo1
stopped in main at line 4 in file "dbx.demo1.c"
   4  {       putchar(c_array[1]);
(dbx) print c_array
c_array = "test"
(dbx) print c_array[1]
c_array[1] = 'e'
(dbx) print c_array[0]
c_array[0] = 't'
(dbx) clear 4
(dbx) status
(dbx) debug dbx.demo2
Reading symbolic information...
Read 42 symbols
(dbx)
```

Debugging Example (con't.)

```
patience% cat dbx.demo1.fixed.c

/*      dbx.demo1.c     */
char c_array[] = "test";
main()
{       putchar(c_array[0]);
        putchar(c_array[1]);
        putchar(c_array[2]);
        putchar(c_array[3]);
        putchar('\n');
}

patience% cc dbx.demo1.fixed.c -o dbx.demo1.fixed

patience% dbx.demo1.fixed

test

patience% cat dbx.demo2.c

/*      dbx.demo2.c     */
char message[] = "leaving now..." ;
main()
{
        int index = 0;

        while (index++ < 5) ;
        {
                printf("index = %d \n", index);
        }
        puts(message);
}

patience%
```

Debugging Using *dbx*

print <exp> ... – Print the value of the expression(s) <exp> ...

Sample *print* commands:

print j+4	display the sum of *j* and *4*
print j/i	display the quotient of *j* and *i*
print argv[2]	display the string pointed to by the second argument vector
print &argv	display the address of the argument vector
print string	display the characters at *string* up to the first null
print string[1]	display the second element of the array *string*
print struct–>member	where *struct* points to a structure, print the *member*
print struct	print the entire structure and current contents

Debugging Using *dbx*

Printing variables

dbx has a very powerful function for displaying the current value of a variable or expression. You can *print* the value of a variable, a pointer, an address, a structure, an element of a structure, or the value of an expression (which may include function calls). *dbx* decides how the data is to be displayed.

When you use *dbxtool*, you can select a section of source code for printing. If you have misunderstood the rules of precedence, misused a pointer, or referenced a structure incorrectly, the problem appears when you *print* the expression.

Debugging Using *dbx*

debug	– Print the name and args of the program being debugged
debug [–k] prog [core]	– Begin debugging <prog> An arbitrary process may be debugged by specifying its process–id instead of <core> Kernel debugging is initiated with –k
step	– Single step one line (step INTO calls)
step <n>	– Single step <n> lines (step INTO calls)
stepi	– Single step one machine instruction (step INTO calls)
stepi <n>	– Single step <n> machine instructions (step INTO calls)
next	– Step one line (skip OVER calls)
next <n>	– Step <n> lines (skip OVER calls)
display	– Print the list of expressions being displayed
display <exp> ...	– Display the value of expressions <exp> ... at every stopping point
cont [sig <signo>]	– Continue execution with signal <signo>
cont at <line> [sig <signo>]	– Continue execution at line <line> with signal <signo>

Debugging Using *dbx*

Running the Process

When the process is halted at a breakpoint, any variables you have chosen for continuous display are shown and the *(dbx)* prompt is displayed. Execution can be continued with the *step* command. This executes one line of source (one *step*) or one line of assembler (*stepi*). Adding an optional *count* as an argument to *step* allows you to proceed for a certain number of lines of code. An interesting variant on the step command is the *next* command. This actually *steps* over the function calls and stays within the current function. Thus, instead of watching the details of each line of code executed (as in *step*), you maintain a sense of the flow of your program at the current level of detail.

Trace and *display* are more expensive in terms of processing but are often invaluable. *Trace* allows you to watch a variable or the execution of a function. *Display* displays the current contents of variables continuously (in *dbxtool*) or at every stopping point *(dbx)*. The debugger must interrupt the process after each step to see if it needs to alter the display. Since you are allowed to set conditions on *trace*, the subprocess may run very slowly.

Use the command *cont* to run to the next breakpoint. To interrupt processing type <CNTRL–C>. At any time you can use the *run* command to restart the program without altering the breakpoints in place. The qualifier *all* with the *delete* command restores breakpoints and traces to their state upon startup of the debugger.

Debugging Example (con't.)

(dbx) **run**

```
Running: dbx.demo2
index = 6
leaving now...

execution completed, exit code is 0
program exited with 0
```

(dbx) **list 1,10**

```
 1  /*      dbx.demo2.c     */
 2  char message[] = "leaving now..." ;
 3  main()
 4  {
 5          int index = 0;
 6
 7          while (index++ < 5) ;
 8          {
 9                  printf("index = %d \n", index);
10          }
```

(dbx) **trace index**

(2) trace index

(dbx) **stop at 7**

(3) stop at "/home/sunray/student1/dbx.demo2.c":7

Debugging Example (con't.)

Debugging Example (con't.)

```
(dbx) run
initially (at line 5):  index = 0
stopped in main at line 7 in file "dbx.demo2.c"
   7            while (index++ < 5) ;
(dbx) step
after line 7:   index = 2
stopped in main at line 7 in file "dbx.demo2.c"
   7            while (index++ < 5) ;
(dbx) step
after line 7:   index = 4
stopped in main at line 7 in file "dbx.demo2.c"
   7            while (index++ < 5) ;
(dbx) step
after line 7:   index = 6
stopped in main at line 9 in file "dbx.demo2.c"
   9                    printf("index = %d \n", index);
(dbx) step
index = 6
stopped in main at line 11 in file "dbx.demo2.c"
  11            puts(message);
(dbx) step
leaving now...
stopped in main at line 12 in file "dbx.demo2.c"
  12  }
```

Debugging Example (con't.)

```
patience% cat dbx.demo2.fixed.c

/*      dbx.demo2.c     */
char message[] = "leaving now..." ;
main()
{
        int index = 0;

        while (index++ < 5)
        {
                printf("index = %d \n", index);
        }
        puts(message);
}

patience% cc -g dbx.demo2.c -o dbx.demo2.fixed

patience% dbx.demo2.fixed

index = 1
index = 2
index = 3
index = 4
index = 5
leaving now...

patience%
```

Debugging Using *dbx*

func	– Print the name of the current function
func <filename>	– Change the current function
func <proc>	– Change the current function to function or procedure <proc>
dump	– Print all variable local to the current procedure
dump <proc>	– Print all variables local to <proc>
up	– Move up the call stack one level
up <number>	– Move up the call stack <number> levels
down	– Move down the call stack one level
down <number>	– Move down the call stack <number> levels
whatis <name>	– Print the declaration of the <name>
which <name>	– Print full qualification of <name>
where	– Print a procedure traceback
where <num>	– Print the <num> top procedure in the traceback

Debugging Using *dbx*

Setting the Scope

What if you reuse a variable name locally in different functions? *dbx* gives you explicit control over the scope of *print*, *display* and *trace*. Variables can be fully qualified with backquotes:

filename 'function1'local_variable

This overrides the scope rules normally used by *dbx*. *dbx* has a notion of the current file (which can be overridden by *file*) and a notion of the currently active function.

func is used to change or display the currently active function. The current stack frame is dumped by the command *dump* displaying all the local variables. This notion of the current function may also be changed by using *up* to move upward to the calling function and *down* to return to the called function. These commands allow you to examine functions and variables (despite name conflicts) which are not currently active.

The commands *whatis, which* and *where* provide information, if you are unsure of what you are looking at. *whatis* displays the original declaration of the variable from the source file. *which* displays the fully qualified name of the variable. *where* displays a stack back trace, which names the function that called the active one, and continues to name functions all the way back to *main.*

Debugging Example (con't.)

```
(dbx) quit
patience% cc -g dbx.demo3.c -o dbx.demo3
patience% dbx dbx.demo3
Reading symbolic information...
Read 47 symbols
(dbx) run
Running: dbx.demo3
1
The index is 3

execution completed, exit code is 0
program exited with 0
(dbx) stop in pfun
(2) stop in pfun
(dbx) run
Running: dbx.demo3
1
stopped in pfun at line 17 in file "dbx.demo3.c"
  17            printf("The index is %d \n", value);
(dbx) where
pfun(value = 3), line 17 in "dbx.demo3.c"
main(0x1, 0xefffd20, 0xefffd28), line 10 in "dbx.demo3.c"
```

Debugging Example (con't.)

patience% **cat dbx.demo3.c**

```
/*      dbx.demo3.c      */
main()
{       int index = 0;

        while (++index <= 3)
        {
                printf("%d\n",index);
                if (index = 3)
                {
                        pfun(index);
                }
        }
}
pfun(value)
int value;
{
        printf("The index is %d \n", value);
}
```

patience%

Debugging Example (con't.)

```
(dbx) dump
value = 3
(dbx) func main
(dbx) display index
index = 3
(dbx) set index = 1
(dbx) cont
The index is 3
2
stopped in pfun at line 17 in file "dbx.demo3.c"
   17           printf("The index is %d \n", value);
(dbx) quit
patience%
```

Debugging Example (con't.)

```
patience% cat dbx.demo3.fixed.c
/*      dbx.demo3.c      */
main()
{       int index = 0;

        while (++index <= 3)
        {
                printf("%d\n",index);
                if (index == 3)
                {
                        pfun(index);
                }
        }
}
pfun(value)
int value;
{
        printf("The index is %d \n", value);
}

patience% cc -g dbx.demo3.fixed.c -o dbx.demo3.fixed

patience% dbx.demo3.fixed

1
2
3
The index is 3

patience%
```

A More Complicated Debugging Example

- Program: *buggy.c*

  buggy.c is a program comprised of several functions which accomplish a variety of tasks depending on the option specified. *buggy* may perform string copies and reverse printing, count to 20, print a string, change a string to upper case, and compute factorials.

- Functions:

  bounds()

  clobber()

  dangler()

  for_null()

  loop_forever()

  macro_gotcha()

  stack_blow()

  print_usage()

A More Complicated Debugging Example

For a complete source code listing of the files used in these modules refer to Appendix D.

Debugging *buggy.c* Using *dbx*

```
patience% cc -g buggy.c -o buggy
patience% dbx buggy
Reading symbolic information
Read 268 symbols

(dbx) run s

Running: buggy s
in stack_blow
Enter an integer to be factored:3
signal SEGV (segmentation violation) in factor at line 160 in file
"buggy.c" 160   }

(dbx) list 160, 175
 160   }
 161
 162   /*************************************
 163       *        stack_blow routine
 164   *************************************/
 165
 166   factor(i)
 167   int i;
 168   {
 169       int next = 0;
 170       next = i - 1;
 171       if (next = 0)
 172           return 1;
 173       else
 174           return (i* factor (next));
 175   }
```

Debugging *buggy.c* Using *dbx*

Let's try running *buggy s* and see what happens. *buggy* should prompt us to enter the integer we want to find the factorial of. After entering the integer, 3, we get a segmentation violation. The message states that the error occurred in the function *factor* of *buggy.c*.

Notice that the *run* command is followed by arguments to be passed to *buggy* (via *argc and argv*) as if they were on the shell command line. The message indicates that *dbx* has suspended *buggy* at the first executable line of the function *factor*.

Debugging *buggy.c* Using *dbx*

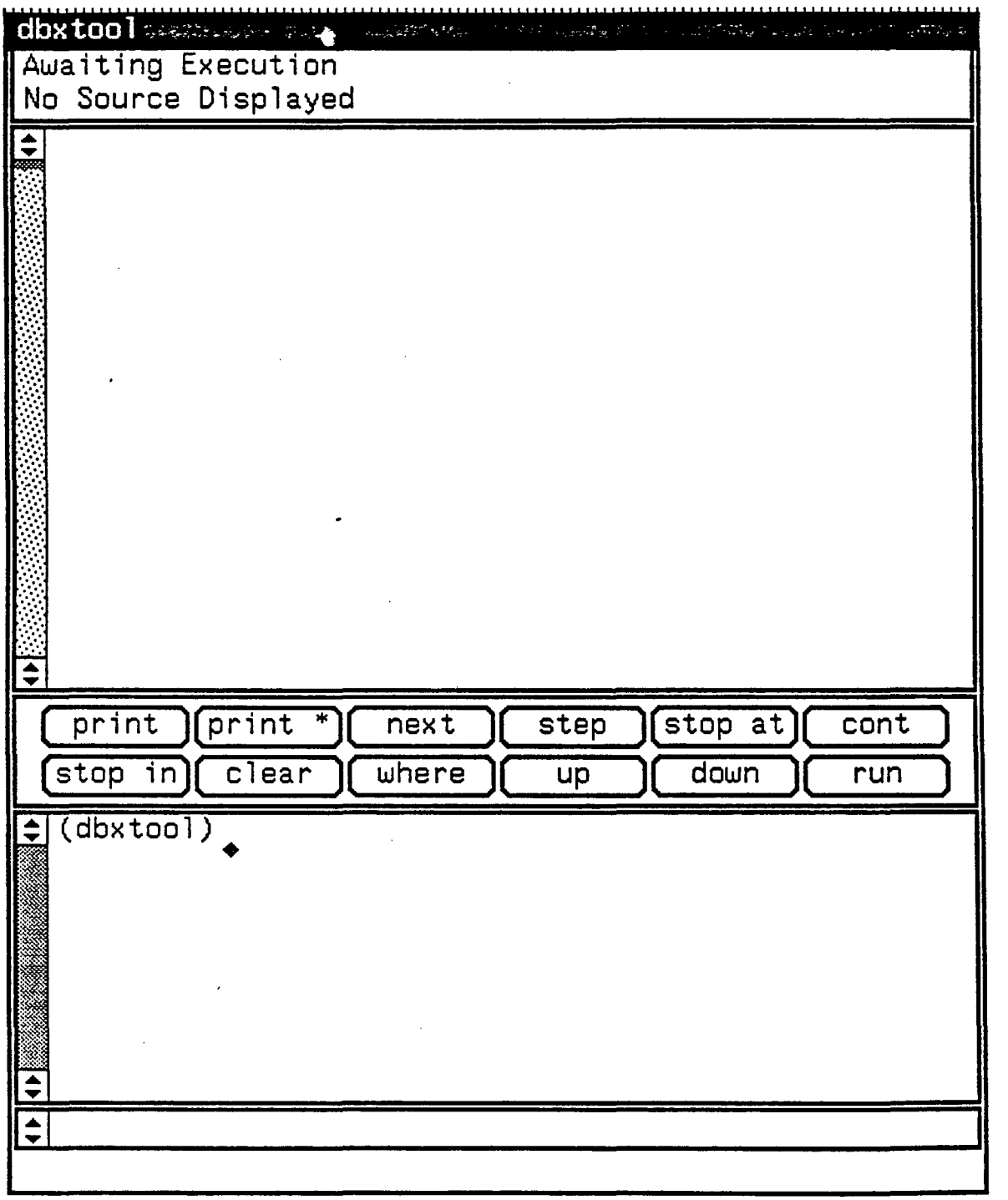
```
(dbx) stop in factor
(2) stop in factor
(dbx) run s
Running: buggy s
in stack_blow
Enter an integer to be factored:5
stopped in factor at line 169 in file "buggy.c"
  169       int next = 0;
(dbx) step
stopped in factor at line 170 in file "buggy.c"
  170       next = i - 1;
(dbx) print i
'buggy' factor' i = 5
(dbx) step
stopped in factor at line 171 in file "buggy.c"
  171       if (next = 0)
(dbx) step
stopped in factor at line 174 in file "buggy.c"
  174           return (i* factor (next));
(dbx) print next
next = 0
(dbx) quit
patience%
```

Debugging *buggy.c* Using *dbx*

Since the segmentation violation occurred in *factor*, let's set a *stop* (breakpoint) in *factor* and then run the program again to try to pinpoint the problem. The number in parenthesis, 2, is the event number. Each breakpoint is assigned a unique event number.

Several single-steps later we see that *next* is already set to zero. We are not returning the value one as expected (by executing line 172) but rather continuing to recursively call "factor." Looking at the condition of the *if* statement at line 171 we can see that *next* will always be zero and the statement should read: if (next == 0).

Introducing *dbxtool*

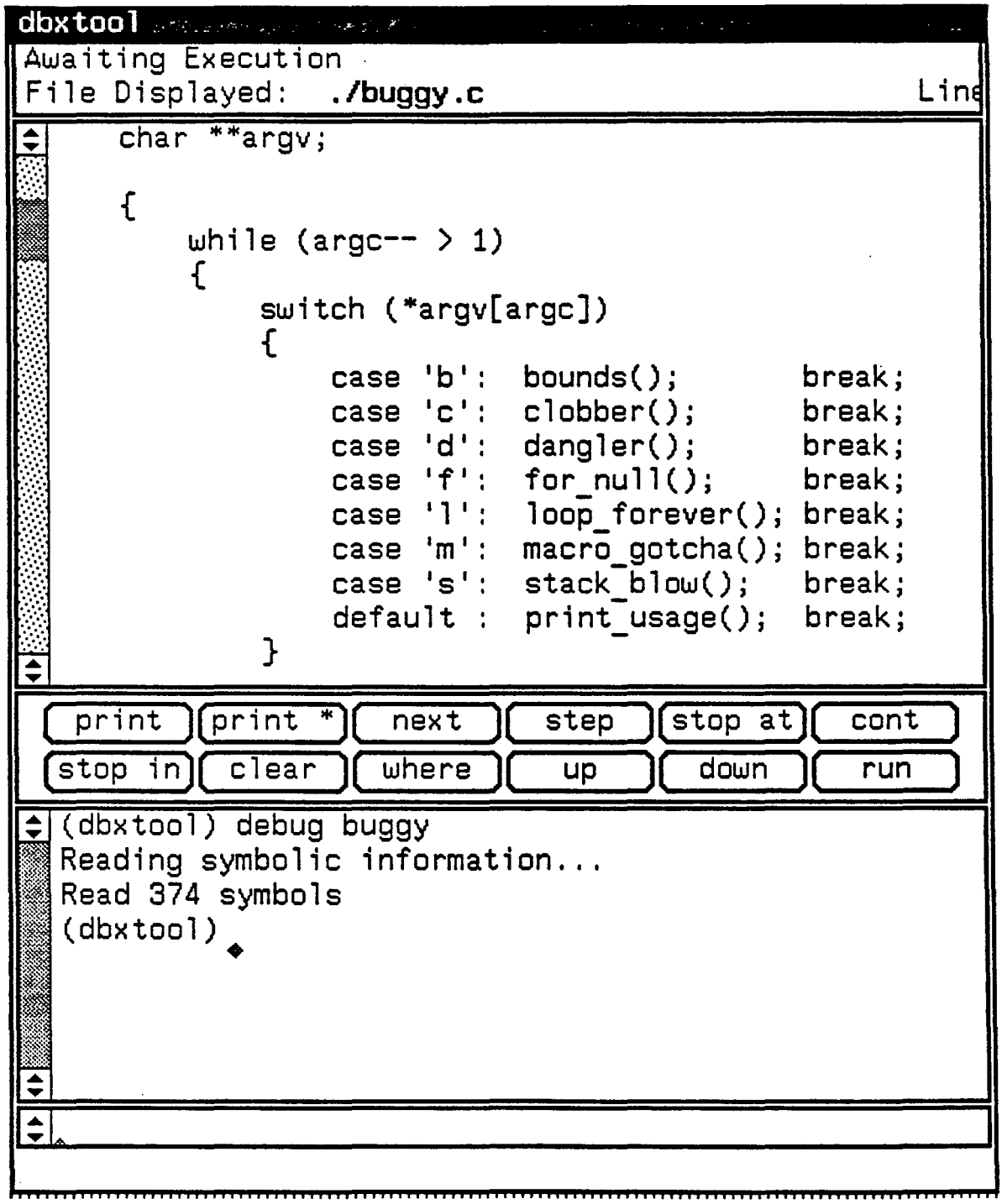


Introducing *dbxtool*

The window version of *dbx* adds a easy to use interface with windows which display:

- Information about the file being debugged
- Continuous source code listing
- Common command buttons
- User command window
- Dynamic object/expression display window

Debugging *buggy.c* Using *dbxtool*

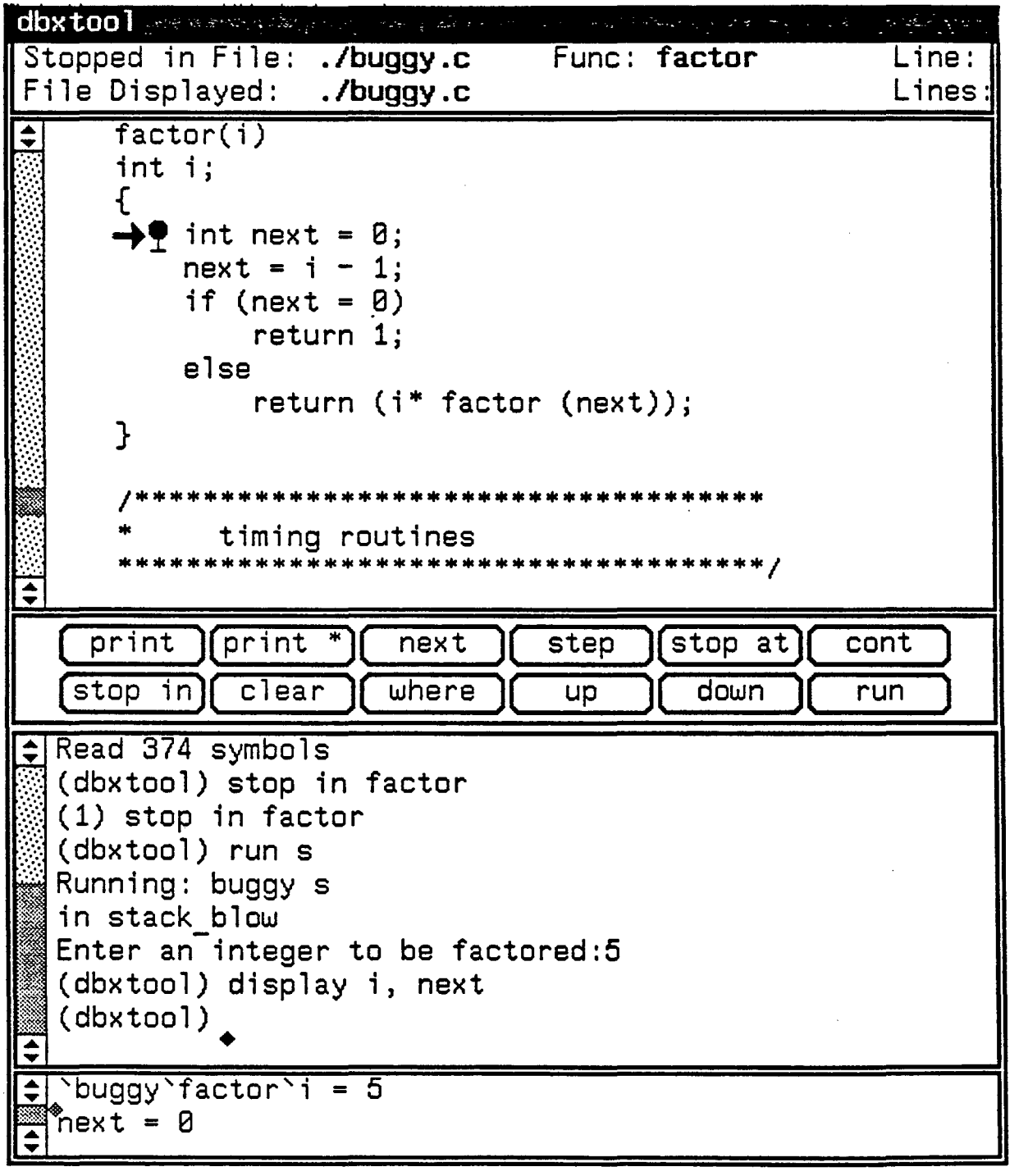


Debugging *buggy.c* Using *dbxtool*

SunView's *dbxtool* can display considerably more information simultaneously. The commands are identical to *dbx,* the difference is you have the option of using the mouse to select addresses and objects in the upper window and actions by clicking in the middle band of buttons. The window directly below the buttons is a keyboard interface to *dbx.* Commands are entered and output is displayed in this window, exactly as you would use *dbx* outside of windows and/or *dbxtool. dbxtool* can evaluate common C expressions (including long arithmetic expressions) as well as display the contents of variables. It understands structures and correctly evaluates statements such as *struct–>element* in even the most complex structures.

When *dbxtool* finds a *core* file containing the same name as the program invoked on the command line, it brings up the code (assuming that the program is compiled with the *–g* flag) and points to the location at which the crash occurred.

Debugging *buggy.c* Using *dbxtool*



Debugging *buggy.c* Using *dbxtool*

The procedure for debugging our program is similar to the activity just conducted with *dbx*. We type *run s* to invoke *buggy* under control of the debugger. This time we ask for a continuous display of the variables *i* and *next* which appear in the lowest window.

We set the breakpoint by clicking on the line of source code containing the *factor* function definition, and then click on the *stop in* button. After clicking three times on the *step* button in the button window, the value of *next* is clobbered in the *if* statement.

Debugging *buggy.c* Using *dbxtool*

patience% **buggy l**
Enter number of loops: *500*
^Z
Stopped

patience% **bg**
[1] buggy l &

patience% **ps | grep buggy**
6849 p2 R 1:34 buggy l

patience% **dbxtool buggy 6849**

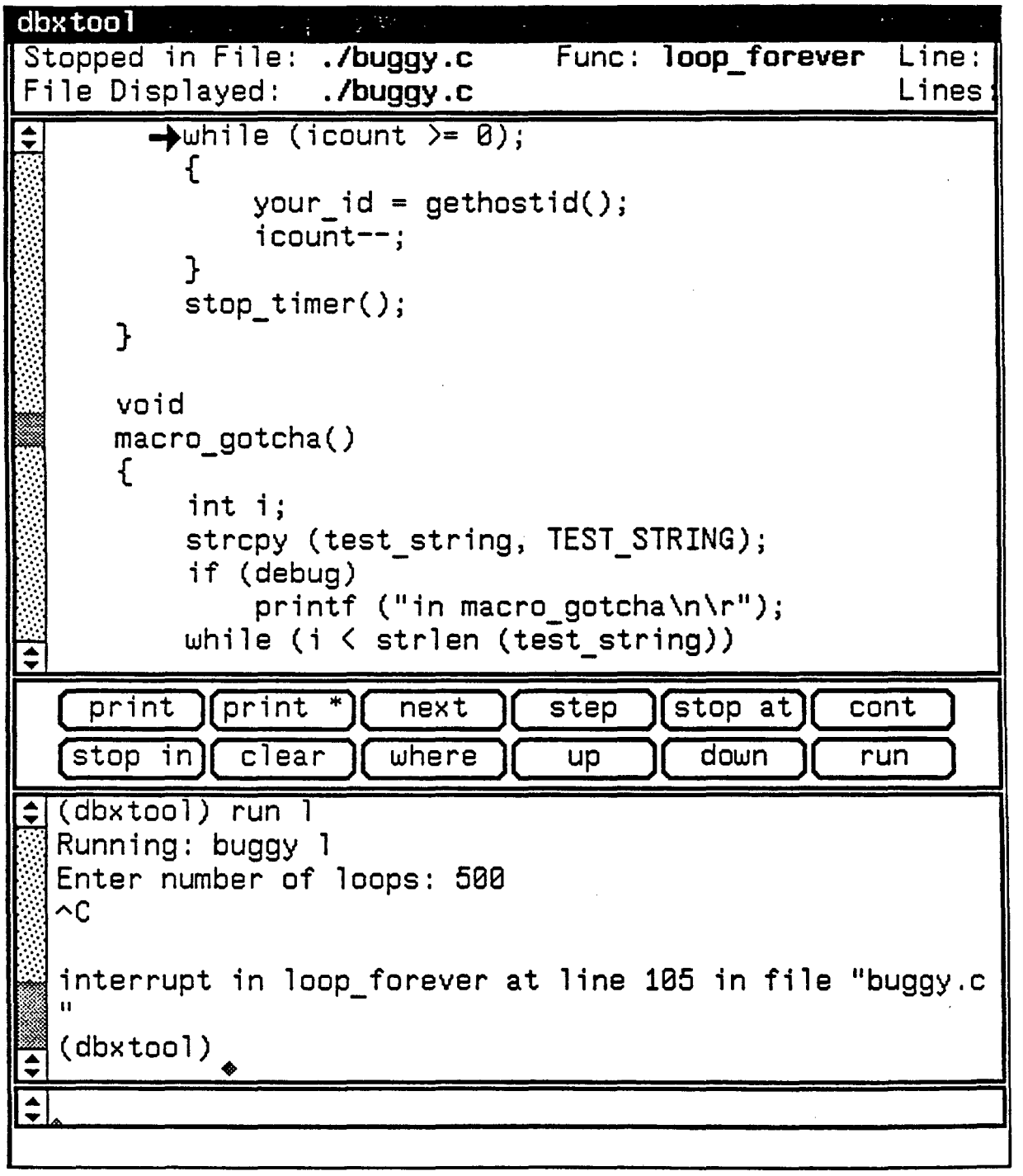
Debugging *buggy.c* Using *dbxtool*

Next, *buggy* is run with the command flag *l* to invoke its loop count routine. The program is suspended and placed in the background when *buggy* has not exited after a long wait.

The *ps* command shows us that *buggy* is running and consuming resources but is not completing.

Now that we know *buggy*'s process id, we attach *dbxtool* to the running program in order to examine it.

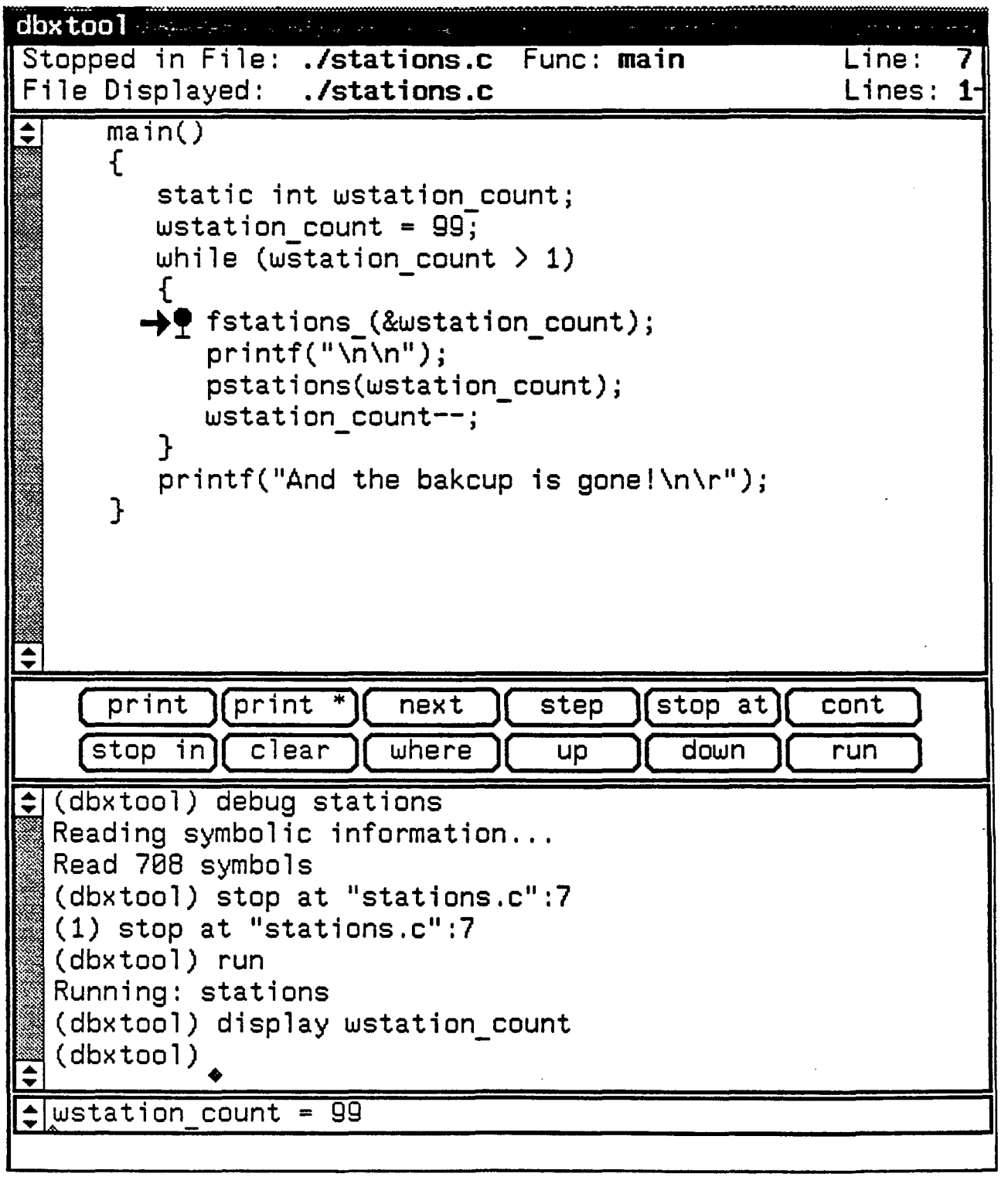
Debugging *buggy.c* Using *dbxtool*



Debugging *buggy.c* Using *dbxtool*

The status line at the top of the display shows that the program is looping in function *loop_forever*.

Debugging *stations* Using *dbxtool*



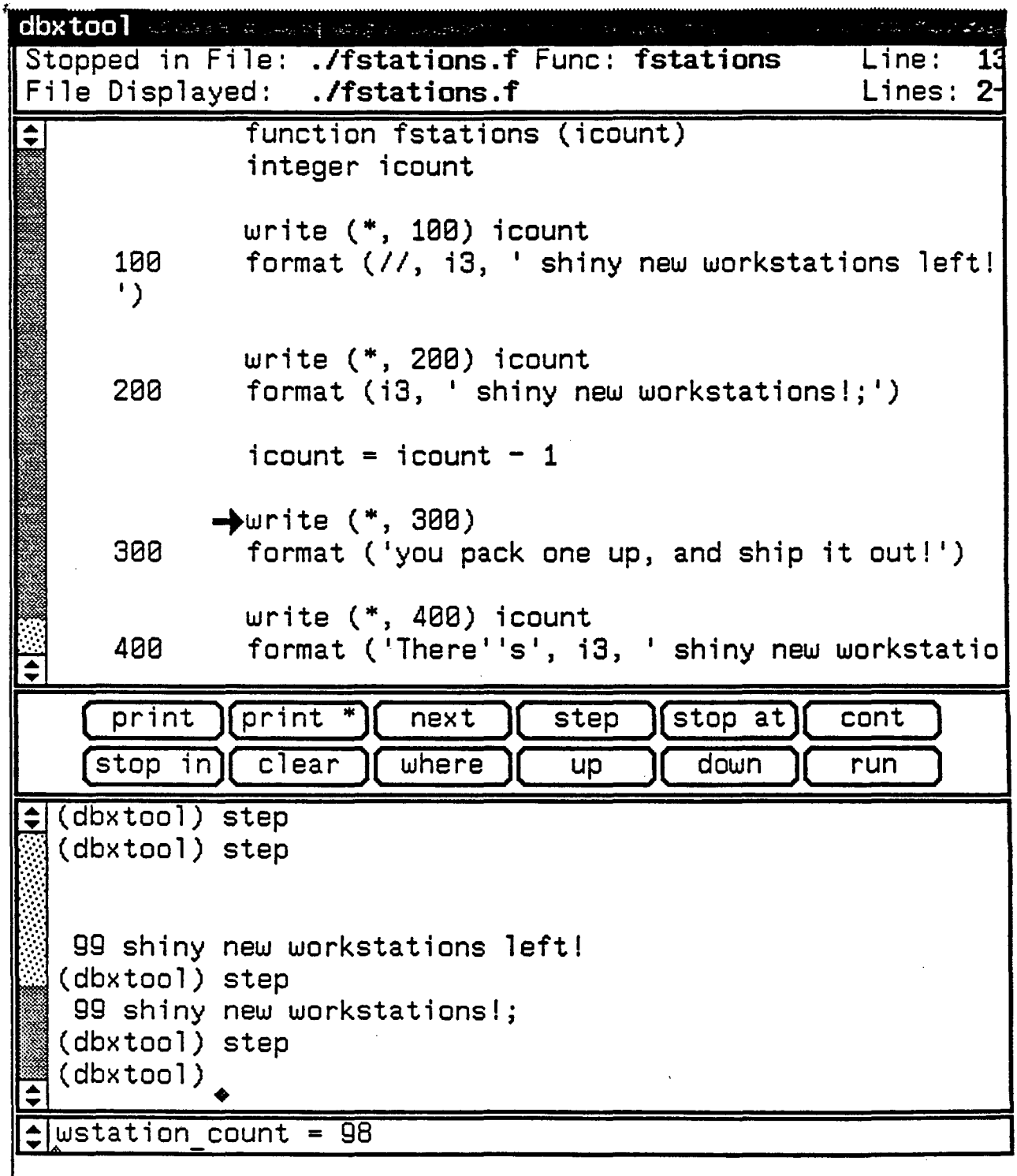
Debugging *stations* Using *dbxtool*

Here we step through the program *stations* to see how well *dbxtool* handles various source languages.

The objective of the program *stations*, is to display modified lyrics to the well-known favorite, *"99 Bottles of Beer on the Wall"*.

A breakpoint is also set at the FORTRAN routine, followed by a continuous display of the variable *wstation_count*.

Debugging *stations* Using *dbxtool*

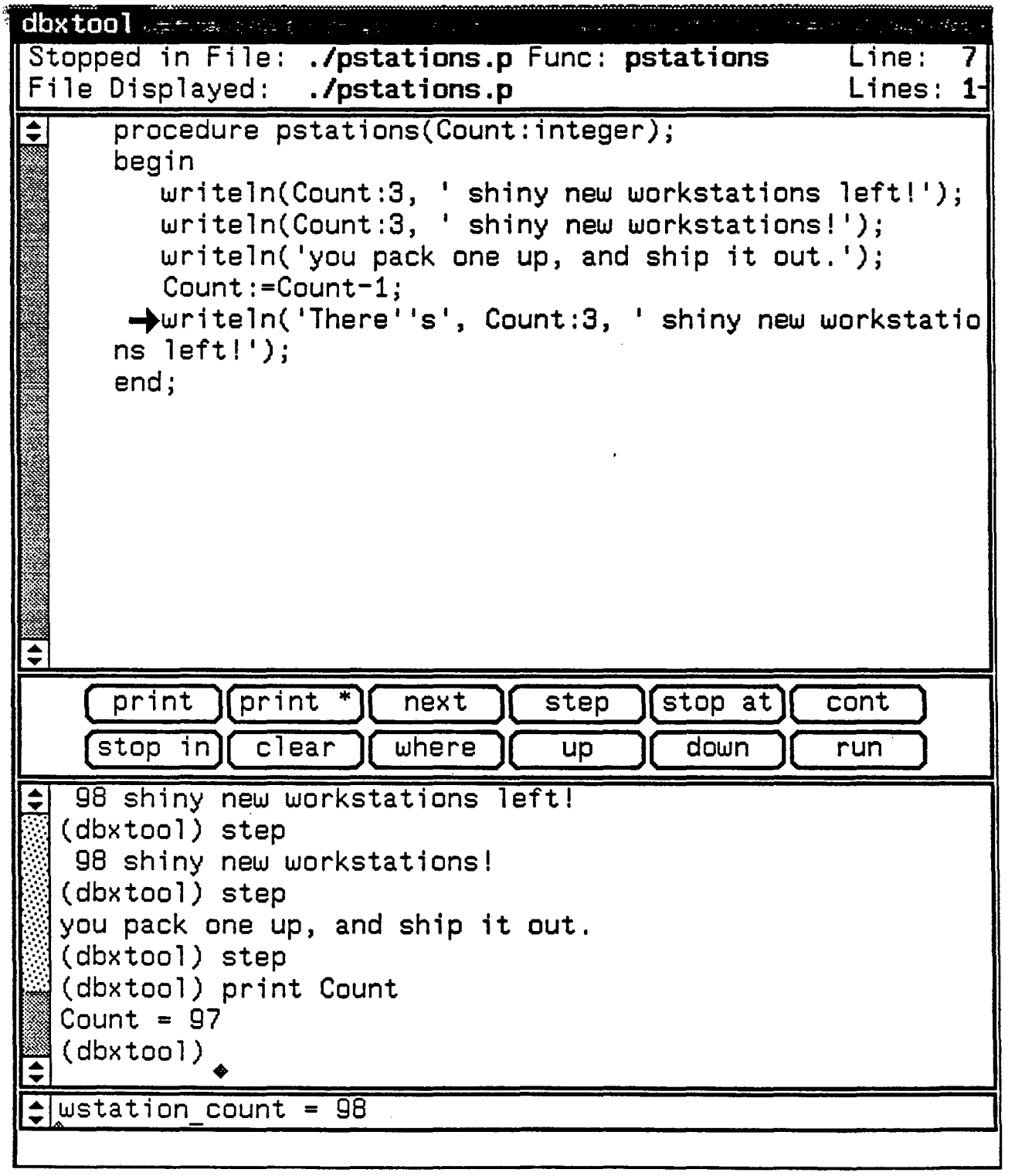


Debugging *stations* Using *dbxtool*

Now we single-step a few times in the FORTRAN routine. Notice how the decrement of *wstation_count* within the routine affects the variable in *main*.

FORTRAN passes arguments by address (by reference) instead of putting a copy on the stack (by value, as do C and Pascal). This is one reason why a FORTRAN subroutine can cause side effects.

Debugging *stations* Using *dbxtool*

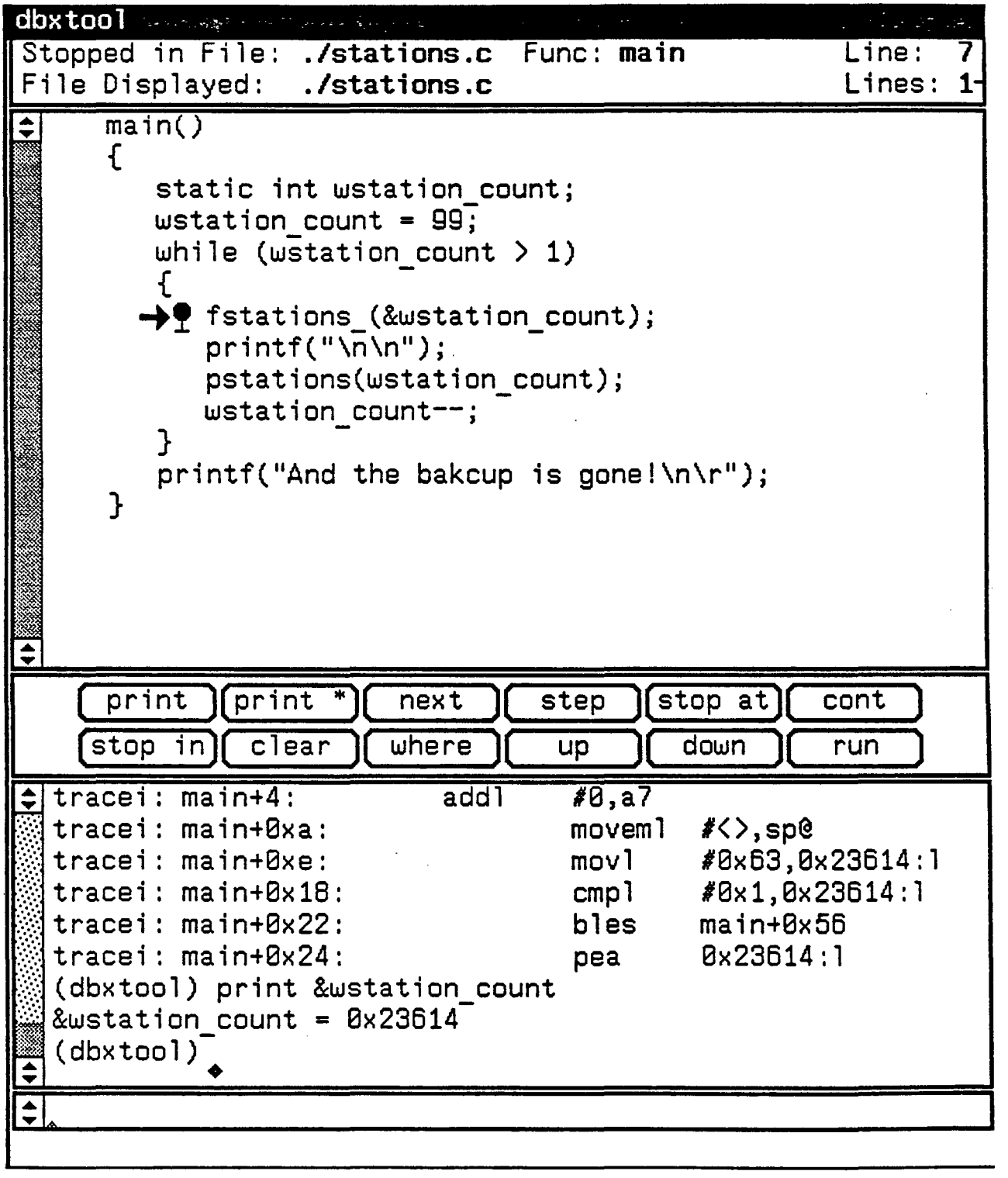


Debugging *stations* Using *dbxtool*

Pascal, like C, is passing the count by value and the decrement which occurs in the subroutine does not affect the variable in the calling routine.

In general, it is not difficult to integrate other language routines into C programs under SunOS. Complications, if any, are due to the differences in string handling, array passing and ordering.

Debugging *stations* Using *dbxtool*



Debugging *stations* Using *dbxtool*

If you are trying to determine how arguments are passed between two languages, it's possible to examine the assembler output one line at a time. Try this on the simplest possible program which meets your needs before working in a more complex environment.

In *stations,* we set the display to *tracei.* This displays a continuous trace of the assembly instructions as they execute. A breakpoint at the call to the *fstations* subroutine, followed by the command *run,* produced this display.

Recall that the song begins, "Ninety-nine shiny new workstations left..." The move of *99 (hex 63) to 23614 hex*, leads us to suspect that this is the location of the variable *wstation_count*. This is confirmed when the address of *wstation_count* is displayed.

Next, the program compares this value with one and does not take the conditional jump on the following line. The clue as to how the arguments are passed is in the next line. Just before a branch to subroutine *fstations*, we see a *pea* (push effective address) *of 23614*. This pushes the address of *wstation_count* onto the stack.

dbxtool and Child Processes

- Use these steps to attach *dbxtool* to a child process that needs debugging.
  - Include code within the parent that will obtain the child's PID.
  - Delay child code execution by inserting a *sleep(20)* in the child process path.
  - Link the program using the *–N* flag in order to make the executable nonshared.
  - Start *dbxtool* on the parent process.
  - Start another copy of *dbxtool* and at the prompt type:

    debug filename

    but do not enter a carriage return!
  - Start the execution of the parent code. Obtain the PID of the child.
  - Enter the child's PID in the second *dbxtool* window, after *filename,* and enter a carriage return.

dbxtool and Child Processes

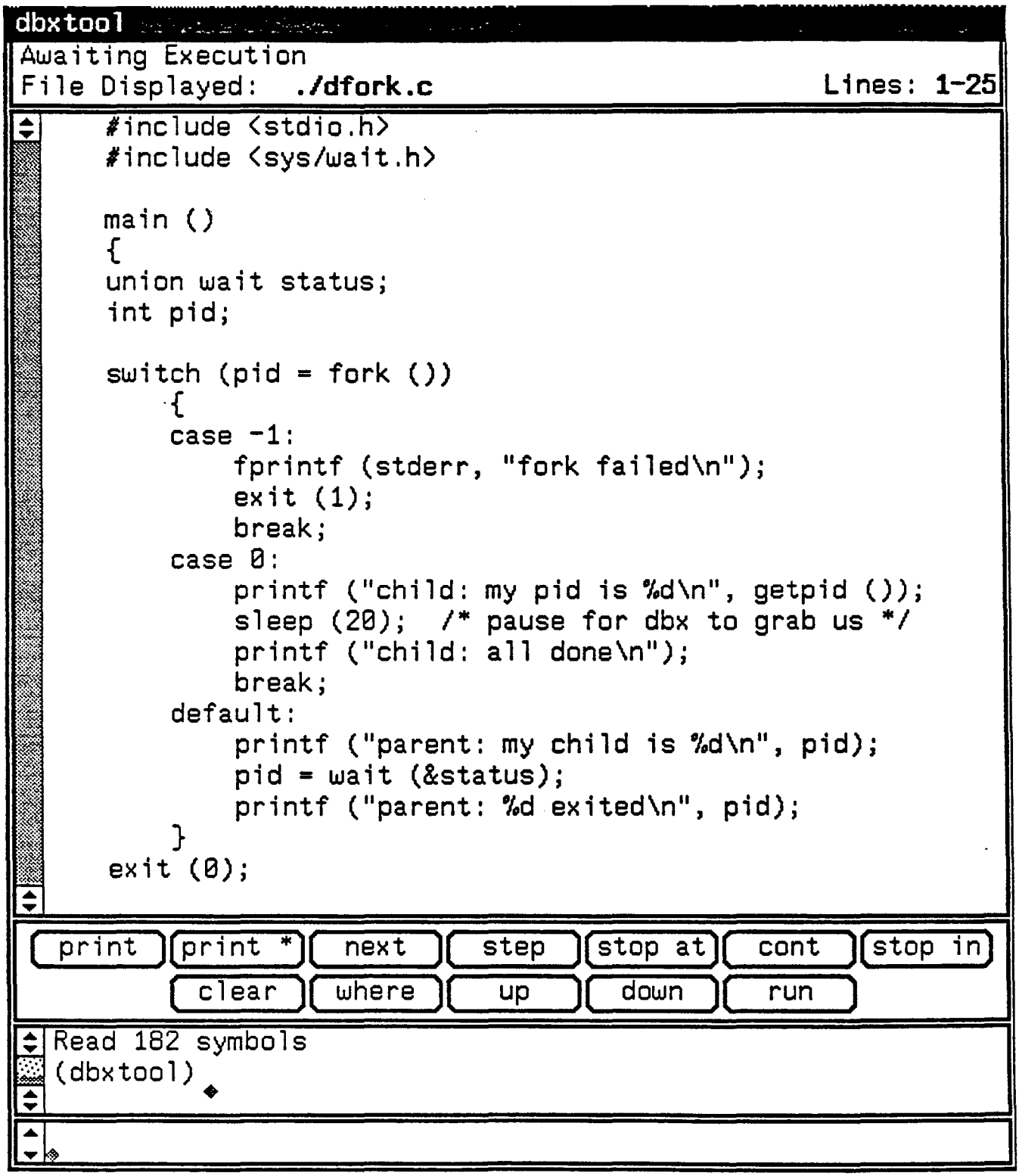
dfork.c

```
#include <stdio.h>
#include <sys/wait.h>

main ()
{
    union wait status;
    int pid;

    switch (pid = fork ())
    {
    case -1:
        fprintf (stderr, "fork failed\n");
        exit (1);
        break;
    case 0:
        printf ("child: my pid is %d\n", getpid ());
        sleep (20);  /* pause for dbx to grab us */
        printf ("child: all done\n");
        break;
    default:
        printf ("parent: my child is %d\n", pid);
        pid = wait (&status);
        printf ("parent: %d exited\n", pid);
    }
exit (0);
}
```

dfork.c

dfork.c is a simple program that creates a child process. If successful, it will print the parent's PID and the child's PID, sleep for 20 seconds, and print messages that the child is done executing and that the child exited. Note that the first two steps to attaching the debugger to the child process have already been completed (i.e. the parent will obtain the child's PID and a sleep exists within the child process path). The third step, link with the *–N* option, would look like: *cc –N –g dfork.c –o dfork*.

Attaching *dbxtool* to a Child Process



Attaching *dbxtool* to a Child Process

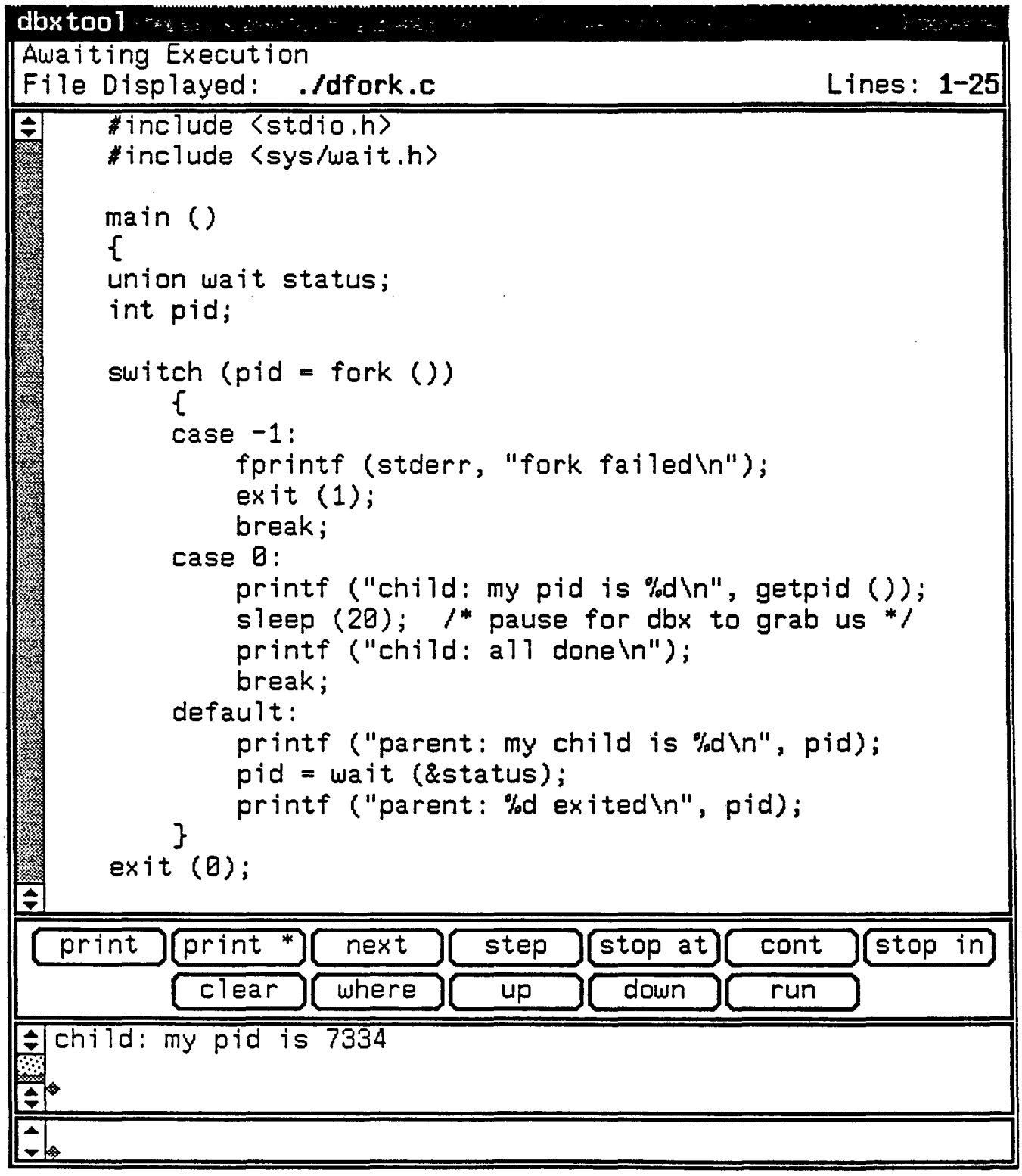
The fourth step is to start *dbxtool* on the parent process:

patience% **dbxtool**
(dbxtool) **debug dfork**

Complete the fifth step by starting another *dbxtool* for the child (do not execute the *debug* command yet by entering a carriage return):

patience% **dbxtool**
(dbxtool) **debug dfork**

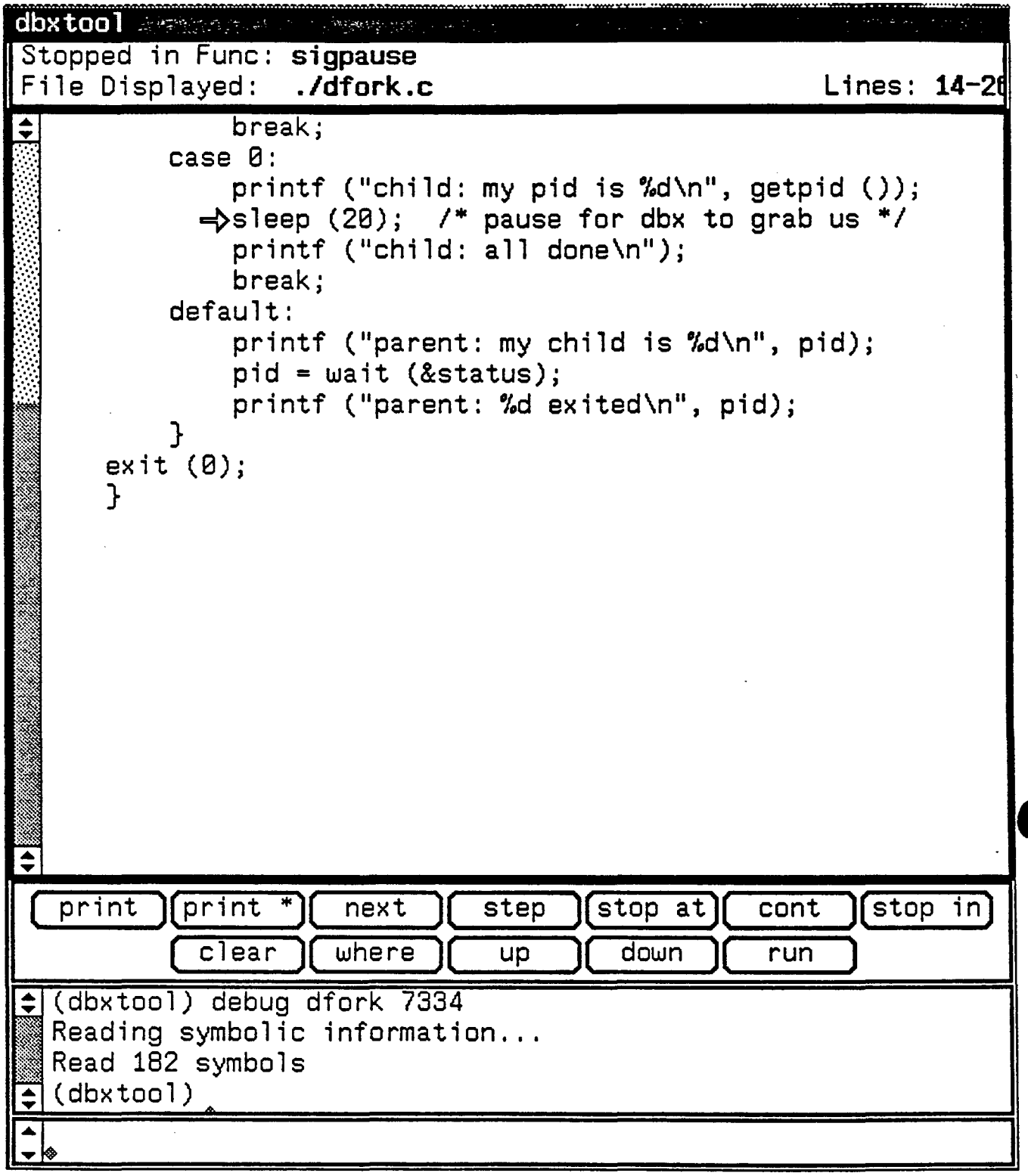
Attaching *dbxtool* to a Child Process



Attaching *dbxtool* to a Child Process

Start executing the parent within the first *dbxtool* window by executing *run* at the prompt. Note that the child's process PID is 7334.

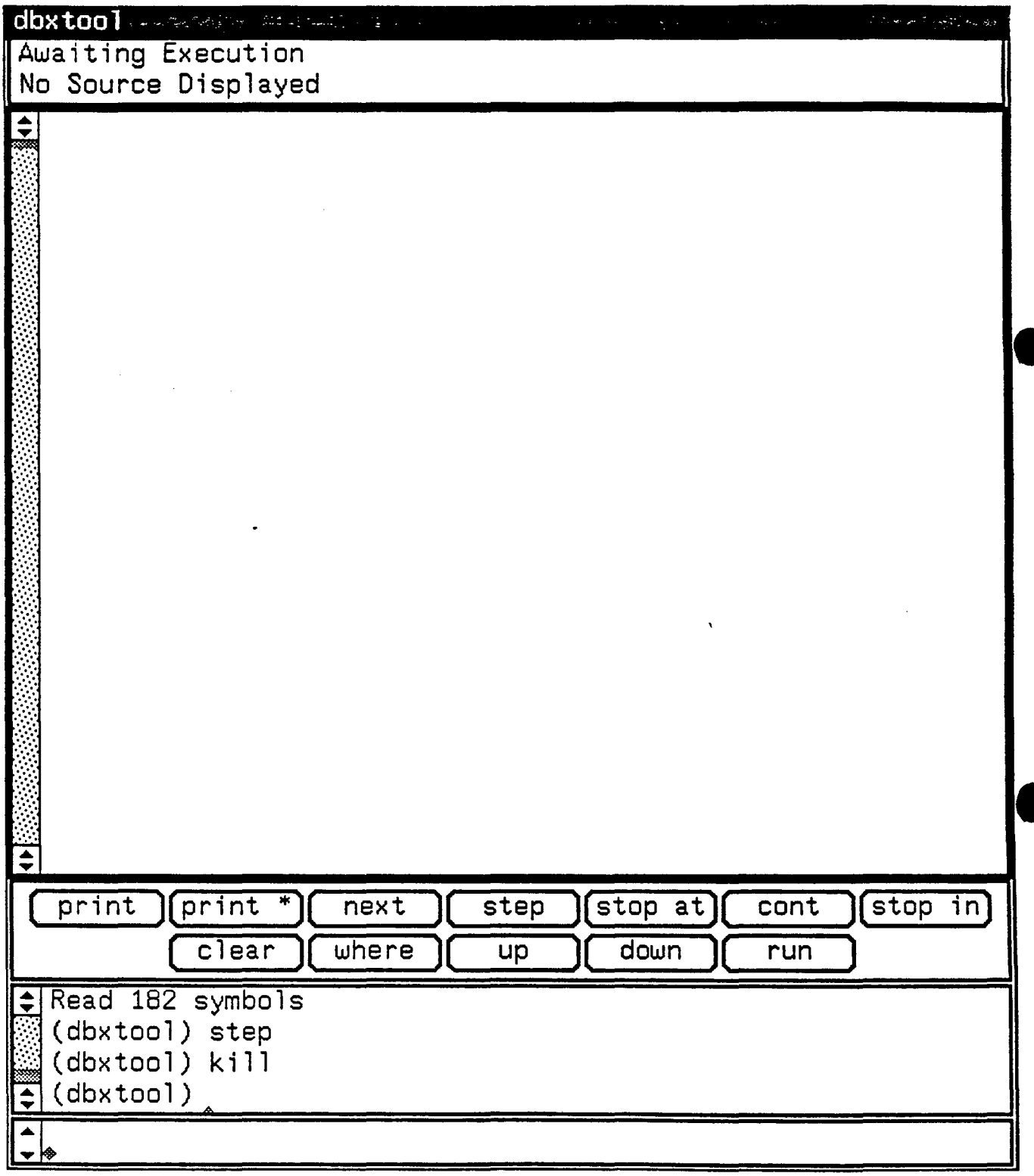
Attaching *dbxtool* to a Child Process



Attaching *dbxtool* to a Child Process

Now, to complete the final step, enter the child's PID in the second *dbxtool* window so that the command reads: *debug dfork 7334* (don't forget to enter a carriage return). At this point we have successfully attached *dbxtool* to the child process and it is ready to be debugged in this window.

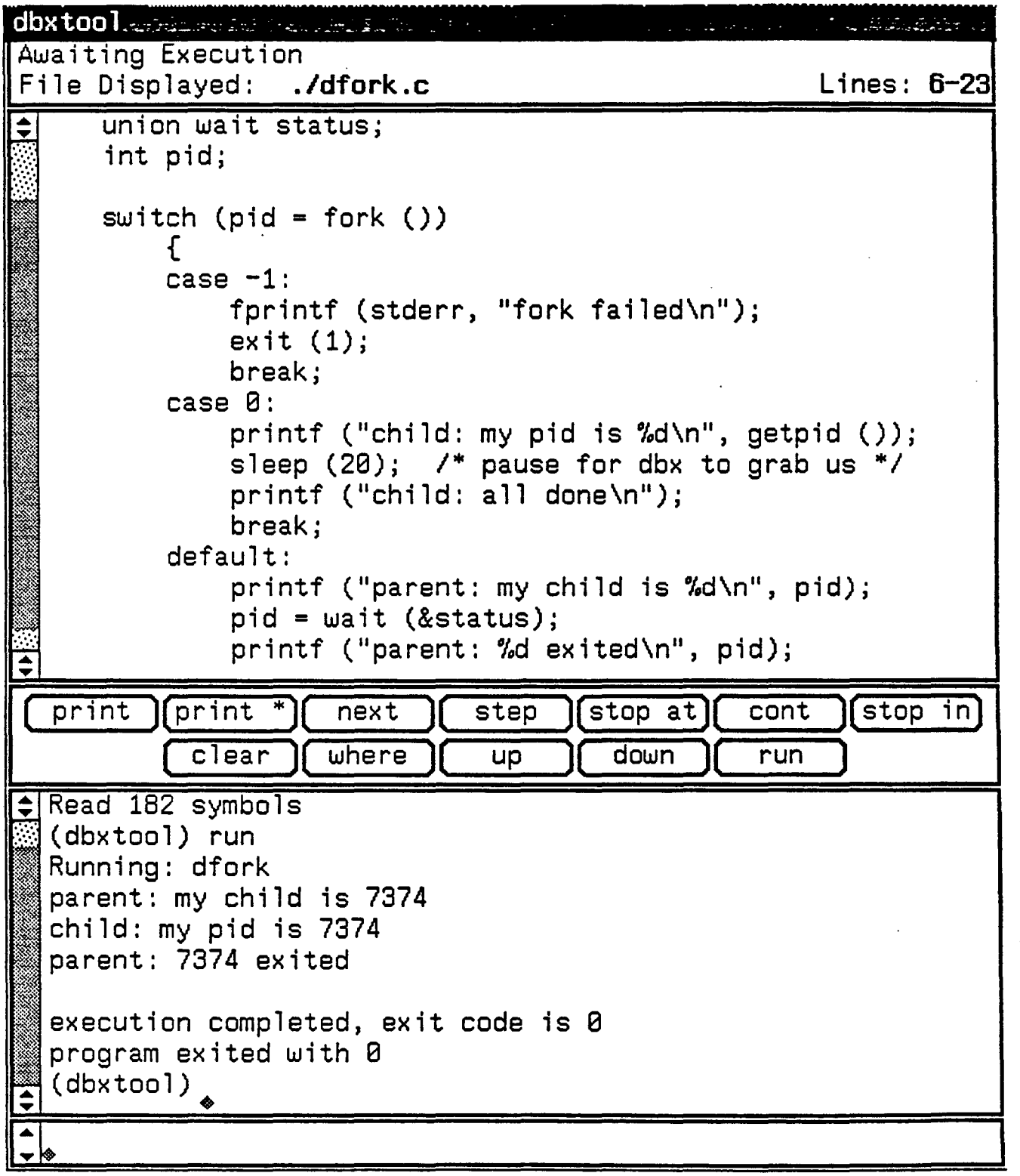
Attaching *dbxtool* to a Child Process



Attaching *dbxtool* to a Child Process

In this example, the child sleeps for twenty seconds and then prints a message that it is done executing. However, because the process is attached to *dbxtool,* the child has not really exited and sent the appropriate completion message to the parent; so the parent waits. Enter *kill* to destroy *dbxtool's* attachment to the child process.

Attaching *dbxtool* to a Child Process



Attaching *dbxtool* to a Child Process

This slide shows a synopsis of what occurs in the parent's *dbxtool* window when trying to attach *dfork's* child to *dbxtool.* Although this is a different run of the example (the PIDs are different) the outcome is otherwise the same.

Summary of *dbx* and *dbxtool* Commands

Execution and Tracing

catch ignore run stop clear next status trace
cont rerun step when delete

Displaying and Naming Data

assign dump undisplay where call print up
whereis display set81 whatis which down

Accessing Source Files

cd func pwd / edit list use ?
file

Miscellaneous

alias detach make sh dbxenv help proc source
debug kill quit

Dbxtool

button toolenv unbutton unmenu menu

Machine Level

nexti stepi stopi tracei

Module11
Performance Analysis

Objectives: Upon completion of this module, the student should be able to:

- Identify performance bottlenecks as an essential step toward the optimization of program performance, with particular attention to page faults and dynamic memory allocation.
- Apply the static analysis tools: *cxref* and *cflow* to source code files and interpret the output.
- Use the C-Shell's built-in *time* function to generate quick, gross analyses of resource consumption.
- Compile programs for analysis by the dynamic analysis tools: *tcov, prof, gprof* and run the tools on live processes, and interpret the output.
- Decide when to apply which tools, with an understanding of the advantages of each.
- Monitor memory usage with *vmstat.*

Evaluation:

Perform Lab 9 to 90% proficiency.

Reference information:

- *The Sun UNIX System: A Sun Technical Report,* Sun Microsystems, Inc. (p/n 800-1419-02).
- *SunPro, The Sun Programming Environment,* Sun Microsystems, Inc. (p/n FF144/20K).

Introduction

Costs of programs:

- real (wall clock) time
- compute–time
- time spent waiting for resources
- programmer's and user's time

} *time*

- main memory
- swap space
- executable image size

} *space*

- file (disk) I/O
- network load
- terminal I/O
- kernel services

} *I/O activity*

Introduction

Why be concerned with performance analysis? Software execution consumes real resources. As every programmer knows, identifying and measuring these behaviors can suggest ways to improve performance.

Software optimization techniques tend to be intimately connected with specific algorithms and data representations. Their application is often art rather than science; significant performance improvement tends to be the result of significant insight. The scope of this module is limited to the mechanics of using the available measuring tools.

Finding the Bottlenecks

Every program has a bottleneck. Finding the bottlenecks requires a combination of knowledge and intuition. SunOS provides a number of tools which assist in the analysis of the performance of programs. Once a program reaches the stage in its development where it compiles correctly, it becomes a candidate for the application of these tools to guide the programmer in improving its efficiency. These analysis tools can be categorized as static or dynamic. Static tools are those which examine and display information about the program source code (*cflow* and *cxref)*. Dynamic tools monitor the program as it runs (*tcov, prof,* and *gprof)*. The results yielded by dynamic tools are often more difficult to grasp, especially since they can be influenced by the particular input data which is likely to vary from one run to the next. However, they are likely to offer more useful insight into actual run–time bottlenecks.

The program *pager.c*, on the following page, counts printable ASCII characters from standard input and prints the results. It is intentionally poorly–designed to demonstrate the usage of performance analysis tools to improve *pager's* performance.

Program Source Code: *pager.c*

```
/* pager.c — Program to be optimized.                    */
/* Reads stdin and counts the occurrence of ASCII chars. */
#define LINES   1000
#define LINELEN 100
#define NUMCHAR 128
char *valloc(), *calloc();
void countmem(), clearmem(), setcount(), printcount();

main()
{
        char **line_array;
        int  *count_array;
        int  lines;
        line_array = (char **) calloc(LINES, sizeof(char *));
        count_array = (int *) calloc(NUMCHAR, sizeof(int));
        clearmem(count_array, NUMCHAR * sizeof(int));

        for (lines = 0; lines < LINES; lines++)
                line_array[lines] = valloc(LINELEN);

        for (lines = 0; lines < LINES; lines++)
                if (!(gets(line_array[lines])))
                        break;

        countmem(lines, line_array, count_array);
        printcount(count_array);
        exit(0);
}
```

Program Source Code: *pager.c*

The program *pager.c*, on the previous page, counts printable ASCII characters from its standard input and prints the results.

Lines 3–7

The program counts only *LINES* number of lines with *LINELEN* or fewer characters. The number of ASCII chars is *NUMCHAR*. To prevent compiler and *lint* warnings, all the functions are declared.

Lines 14–16

Memory is allocated for an array of line pointers (line_array) and the output array (count_array) using *calloc()* (a memory allocation system call related to *malloc()*). The output array is zeroed by the *clearmem()* function.

Lines 18–19

Storage for each line is allocated using *valloc()* and the line pointer is put in *line_array*. *valloc()* is also a system call for memory allocation and is related to *malloc()* and *calloc()*.

Lines 21–23

Data is read line–by–line from *stdin* until *line_array* is full or the input is exhausted.

Line 25

The job of the *countmem()* function is to count each character in *line_array* and put the results in *count_array*.

Line 26

printcount() prints *count_array* to *stdout*.

Line 27

Unless the program exits normally, many of the performance analysis tools will not function correctly.

Program Source Code: *pager.c* (con't.)

```
30
    void                /* count chars in line_array */
    countmem(numlines, line_array, count_array)
    int  numlines;
    char **line_array;
35  int  *count_array;
    {
        int  lines, chars;

        for (chars = 0; chars < LINELEN; chars++)
40          for (lines = 0; lines < numlines; lines++)
                if (line_array[lines][chars] != 0)
                    setcount(count_array,
                        line_array[lines][chars]);
    }
45

    void                /* set memory to 0 for num char */
    clearmem(cp, num)
    char *cp;
50  int  num;
    {
        while (num--)
        *cp++ = 0;
    }
55
```

Program Source Code: *pager.c* (con't.)

Lines 31–44

The *countmem()* function is a compound loop. It goes through line_array character–by–character and line–by–line. Every time it finds a valid character, it calls *setcount()* to make an entry in *count_array.*

Lines 47–54

The *clearmem()* function is a simple loop.

Program Source Code: *pager.c* (con't.)

```
    void            /* increment the count array for c */
    setcount(count_array, c)
    int  *count_array;
60  char c;
    {
        count_array[c] += 1;
    }

65
    void            /* print content of char count array */
    printcount(array)
    int  *array;
    {
70      int  c;

        for (c = ' '; c < '~'; c++) {
            if (!(c % 8))
                printf("\n");
75          printf("\t%c %d", c, array[c]);
        }
        printf("\n");
    }
```

Program Source Code: *pager.c* (con't.)

Lines 57–63

The *setcount()* function increments the *count_array* position which corresponds to the ASCII code for the character. Each character has a unique value, so this keeps a running count of each character's occurrences.

Lines 66–78

The *printcount()* function prints *count_array*. It starts at the array position which corresponds to the value of the space character (' ') and continues until it reaches the value of the tilde ('~') character. On the first line, and on every eighth line after that, it prints a new–line to make a presentable tabulated output. For each array position, it prints the character and its count.

Static Analysis using *cflow*

- *cflow* builds a static call graph (or flow graph), representing the relationships among functions which are defined in *.c, .i, .l, .y, .s*, and *.o* files.

- To track the relationships among functions defined in library files, first extract *.o* files from the *.a* file using *ar()*.

- The output lines are numbered and indentation shows hierarchy.

- The original appearance of a function name is identified by source file and line, if known. If not, it is represented by <>. (Line numbers aren't appropriate for *.o* files.) The type returned by the function is also displayed.

- Successive references to a function use the original line number to identify it.

- *cflow* recognizes the following C preprocessor flags: *–I, –D,* and *–U.*

- Command Format:

*cflow [–r] [–ix] [–i_] [–d*num*] filename(s)*

Static Analysis Using *cflow*

Note that the *.c, .l,* and *.y* files are passed through the C preprocessor before analysis but the *.i* files are not. The *.s* files are first assembled and then treated as *.o* files. If you need to run *cflow* over files containing *lex* or *yacc* programs, run it over the source files, not the output.

cflow accepts the following C preprocessor options:

- *-I* – specify the search path for *#include* files which are specified with angle brackets. (e.g. *<sys/file.h>*)
- *-D* – define a symbol
- *-U* – undefine a symbol

The *cflow* options are:

- *-r* – Invert the normal graph, from the normal *calls* ("which functions does this function call"), to *called-by* ("which functions call this function").
- *-ix* – Report on data references.
- *-i_* – Include references to functions which have names beginning with the underscore character. By default, it is assumed that these are private functions, and no reference to them are included in the graph.
- *-d* – ("down") allows you to reduce the information by restricting the graph to the first couple of levels. This exists to reduce the lexical complexity of the output.

  Warning! Information is lost by "cutting off" cflow output.

Sample Files

- Contents of file *foo.c:*

```
1   int n;

    main (argc, argv)
    int   argc;
5   char *argv[];
    {
       n = foo();
       exit(n);
    }
10
    int foo()
    {
       return (bar(17));
    }
```

- Contents of file bar.c

```
    int bar(n)
    {
       return (n + n);
    }
```

Sample Files

This example is composed of three simple functions: *main, foo,* and *bar. main* calls *foo* and *foo* calls *bar* with the integer 17 as a parameter. *bar* doubles this integer and returns. If everything works correctly, the returned value will be 34.

cflow Examples

- Running *cflow* on an incomplete program:

```
patience% cflow foo.c
1       main: int(), <foo.c 6>
2               foo: int(), <foo.c 12>
3                       bar: <>
4               exit: <>
```

- Running *cflow* on a complete program:

```
patience% cc -o foo foo.c bar.c
patience% echo $status
34
patience% cflow foo.c bar.c
1       main: int(), <foo.c 6>
2               foo: int(), <foo.c 12>
3                       bar: int(), <bar.c 2>
4               exit: <>
patience% cflow bar.c foo.c
1       main: int(), <foo.c 6>
2               foo: int(), <foo.c 12>
3                       bar: int(), <bar.c 2>
4               exit: <>
patience%
```

cflow Examples

The output from running *cflow* on *foo.c* is only four lines long. As for all *cflow* output, each line begins with a reference number and the indentation indicates the depth of the function call. Note that the general format for presenting functions is:

function_name: definition, <source_file line_number>

as in "main: int()". The rest of the line, delimited by angle brackets, is the name of the source file and the line number of the definition. If this information is undefined, no data appears between the angle brackets, as in "bar: <>". The reason this is undefined in this example is because only *foo.c* was input to *cflow.*

Both *foo.c* and *bar.c* are input to *cflow* in the second example. Now *cflow* can figure out the name of the source file and the line number of its definition for the function *bar.c:* "bar: int(), <bar.c 2>".

The third *cflow* example illustrates that the output of *cflow,* the call graph, is independent of the order in which the input files are specified.

cflow Examples (con't.)

```
patience% cflow -r bar.c foo.c
1       bar: int(), <bar.c 2>
2               foo : <>
3       exit: <>
4               main : <>
5       foo: int(), <foo.c 12>
6               main : 4
7       main: int(), <foo.c 6>
patience% ar x /lib/libc.a exit.o
patience% cflow exit.o foo.c
1       main: int(), <foo.c 6>
2               foo: int(), <foo.c 12>
3                       bar: <>
4               exit: text, exit.o
5                       _cleanup: <>
6                       _exit: <>
patience% cflow bar.c exit.o foo.c
1       main: int(), <foo.c 6>
2               foo: int(), <foo.c 12>
3                       bar: int(), <bar.c 2>
4               exit: text, exit.o
5                       _cleanup: <>
6                       _exit: <>
```

cflow Examples (con't.)

This next example illustrates the output of *cflow –r* which reverses the output such that function names are listed on the first level and the calling functions on the second level.

After this we extract a *.o* file from the appropriate library and use it as input to *cflow* along with *foo.c*, and next with both *foo.c* and *bar.c*. Note that the only information extracted by *cflow* from the *.o* file is whether the name is in the text segment or the data segment.

cflow Examples (con't.)

```
patience% cflow -ix bar.c exit.o foo.c
1       main: int(), <foo.c 6>
2               foo: int(), <foo.c 12>
3                       bar: int(), <bar.c 2>
4               exit: text, exit.o
5                       _cleanup: <>
6                       _exit: <>
7               n: int, <foo.c 1>

patience% cflow -r -ix bar.c exit.o foo.c
1       _cleanup: <>
2               exit : <>
3       _exit: <>
4               exit : 2
5       bar: int(), <bar.c 2>
6               foo : <>
7       exit: text, exit.o
8               main : <>
9       foo: int(), <foo.c 12>
10              main : 8
11      main: int(), <foo.c 6>
12      n: int, <foo.c 1>
13              main : 8
```

cflow Examples (con't.)

The *–ix* option tells *cflow* to also specify the external and static data symbols. In these simple files there is only one: *n*. The *–ix* switch is dependent on the *–r* switch, as is shown in the last example.

Running *cflow* on *pager.c*

```
patience% cflow pager.c

1       main: int(), <pager.c 10>
2               calloc: <>
3               clearmem: void*(), <pager.c 51>
4               valloc: <>
5               gets: <>
6               countmem: void*(), <pager.c 36>
7                       setcount: void*(), <pager.c 61>
8               printcount: void*(), <pager.c 69>
9                       printf: <>
10              exit: <>
```

Running *cflow* on *pager.c*

Static Analysis Using *cxref*

- Generates a listing of every use of every symbol in the input files. This includes every *#defined* symbol, and every formal parameter for each parameterized macro.

- Useful for discovering random references, loops, possible replacements of function calls by macros, uses of function names which are not replaceable by macros.

- Output is voluminous.

- This is the *only* tool which can be used to track references to macro names.

- Command Format:

  cxref [–c] [–w [num]] [–o filename] [–t] [–s] filename(s)

Static Analysis Using *cxref*

The *cxref* utility produces a cross-referenced listing of all the symbols used in a source module and its include files.

An example of its invocation:

cxref -o xref prog.c

This produces a cross-reference of *prog.c* in a file named *xref.* The cross-reference is voluminous and indicates where, by source code line number, each symbol is defined and used.

The *cxref* options are:

-w [num]	limit the output to *num* columns.
-o filename	write the output to a named file instead of to standard output.
-t	same as *-w80.*
-c	combine the output from all the input files.
-s	don't report on the name of each input file; operate silently.

Running *cxref* on *foo.c*

patience% **cxref foo.c**
foo.c:

SYMBOL	FILE	FUNCTION	LINE
argc	foo.c	—	3
	foo.c	main	*4
argv	foo.c	—	3
	foo.c	main	*5
bar	foo.c	foo	13
exit	foo.c	main	8
foo()			
	foo.c	—	11
	foo.c	main	7
main()			
	foo.c	—	*3
n	foo.c	—	*1
	foo.c	main	7 8

Running *cxref* on *foo.c*

cxref produces a cross–reference listing of the symbols used in the program. It prints a table ordered by the symbol name.

For each symbol, it shows in which file, in which function, and on which line the symbol appears. Line numbers marked with an asterisk ('*') indicate places where the symbol is declared.

Notice that local names are included, and that each place they are defined is marked. This type of listing is helpful when analyzing the effects of modifying a function or declaration. It displays where the target object is so you can find out how it is used.

Running *cxref* on *pager.c*

patience% **cxref pager.c**

pager.c:

SYMBOL	FILE	FUNCTION	LINE
LINELEN	pager.c	—	*4 19 39
LINES	pager.c	—	*3 14 18 21
NUMCHAR	pager.c	—	*5 15 16
array	pager.c	—	67
	pager.c	printcount	*68 75
c	pager.c	—	58
	pager.c	printcount	*70 72 73 75
	pager.c	setcount	*60 62
calloc()			
	pager.c	—	*6
	pager.c	main	14 15
chars	pager.c	countmem	*37 39 41 43
clearmem()			
	pager.c	—	*7 48
	pager.c	main	16
count_array	pager.c	—	32 58
	pager.c	countmem	*35 42
	pager.c	main	*12 15 16 25 26
	pager.c	setcount	*59 62
countmem()			
	pager.c	—	*7 32
	pager.c	main	25
cp	pager.c	—	48
	pager.c	clearmem	*49 53

Running *cxref* on *pager.c*

Running *cxref* on *pager.c* (con't.)

exit	pager.c	main	27
gets	pager.c	main	22
line_array	pager.c	—	32
	pager.c	countmem	*34 41 43
	pager.c	main	*11 14 19 22 25
lines	pager.c	countmem	*37 40 41 43
	pager.c	main	*13 18 19 21 22 25
main()			
	pager.c	—	*9
num	pager.c	—	48
	pager.c	clearmem	*50 52
numlines	pager.c	—	32
	pager.c	countmem	*33 40
printcount()			
	pager.c	—	*7 67
	pager.c	main	26
printf	pager.c	printcount	74 75 77
setcount()			
	pager.c	—	*7 58
	pager.c	countmem	42
valloc()			
	pager.c	—	*6
	pager.c	main	19

Running *cxref* on *pager.c* (con't.)

Using *time (csh)*

- Command Format:

  time [command]

- Output is displayed as:

  u s min:sec XX% k io pf+0w

- Example:

```
patience% time wc /usr/man/man1/csh.1
   3077   13353   76185 /usr/man/man1/csh.1
0.5u 0.2s 0:00 86% 0+80k 11+0io 10pf+0w
```

Using *time (csh)*

The *time* command has a number of varieties. The most useful is built into the C–Shell itself.

The fields indicate:

u the amount of time, in seconds, used by the *user* portion of the process' code (as opposed to the *system* portion, that is, the amount of time spent by the operating system satisfying the process' system calls).

s the amount of *system* time used on behalf of the process.

min:sec
the actual running time.

XX% percentage of machine usage.

k the average amount of shared+unshared memory, in kilobytes.

io the number of input and output block operations.

pf+w the number of page faults and process swaps.

In the example we time how long it takes to count the number of lines, words, and characters in the man page for *csh*. The first line of output shows the count, respectively. The second line of output is from the *time* command.

Compiling and Timing the Pager Program

patience% **cc –o pager pager.c**

patience% **time pager < /usr/dict/words**

```
 0     ! 0     " 0     # 0     $ 0     %0     & 1     ' 1
 ( 0   ) 0     * 0     + 0     , 0     - 0     . 0     / 0
 0 0   1 0     2 0     3 0     4 0     5 0     6 0     7 0
 8 0   9 0     : 0     ; 0     < 0     = 0     > 0     ? 0
 @0    A 219   B 1     C 2     D 0     E 0     F 0     G 0
 H 0   I 1     J 0     K 1     L 1     M 3     N 1     O 0
 P 0   Q 0     R 0     S 2     T 0     U 0     V 0     W 0
 X 0   Y 0     Z 0     [ 0     \ 0     ] 0     ^ 0     _ 0
 ` 0   a 1270  b 204   c 331   d 292   e 675   f 89    g 174
 h 117 i 516   j 15    k 26    l 462   m 269   n 544   o 375
 p 78  q 16    r 381   s 279   t 457   u 178   v 64    w 18
 x 17  y 95    z 7     { 0     | 0     } 0
11.0u 231.6s 59:36 6% 0+968k 4+0io 102349pf+0w
```

Compiling and Timing the Pager Program

The *pager* program is compiled normally the first time around. Then it is run on the contents of the *spell()* dictionary, using the C shell built–in *time* command. After some period of time, *pager* prints its results and time shows how long it took (keep in mind that the program is very inefficient).

Note:

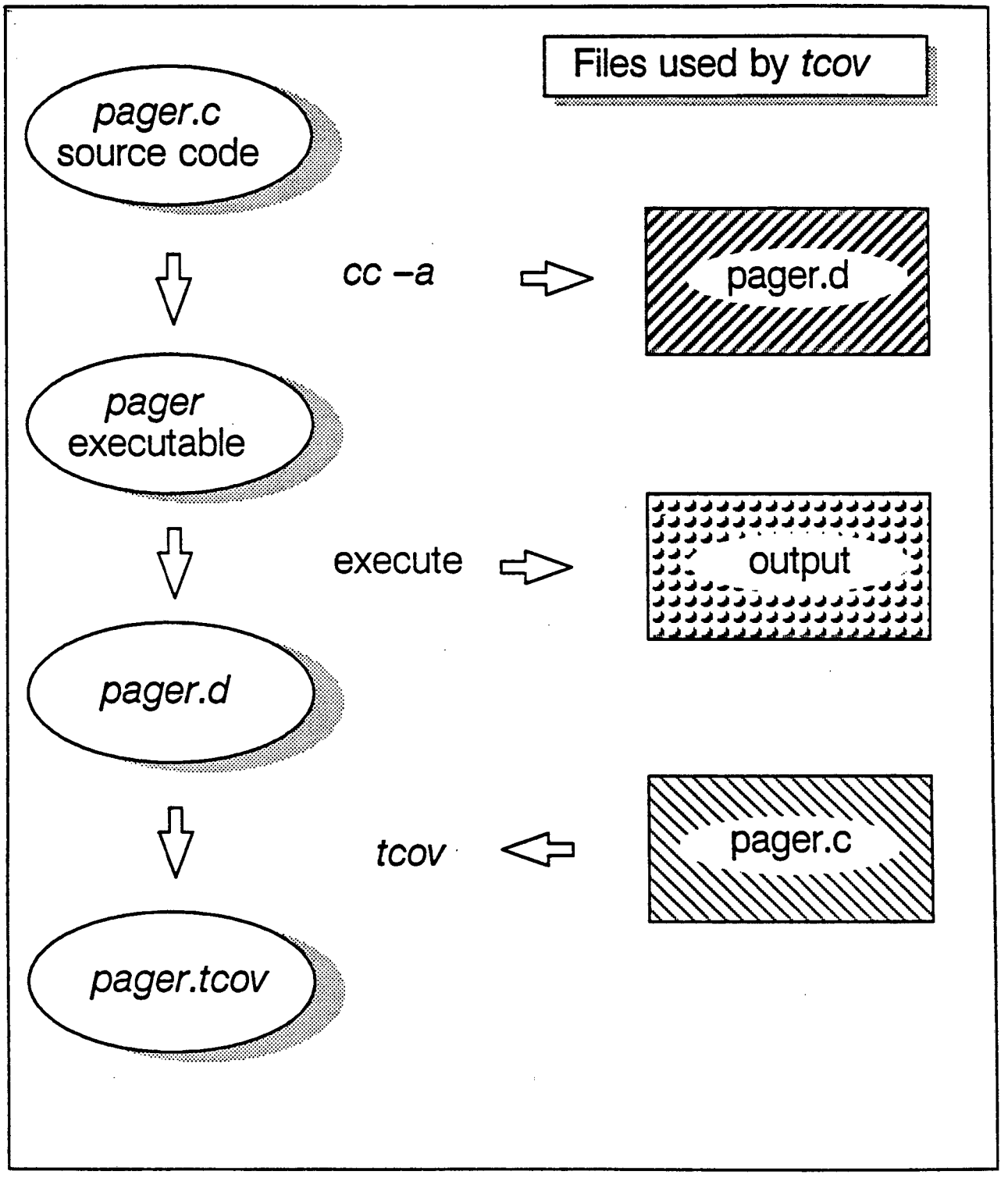
The demonstrations presented in this module were run on a Sun 3/60LE client with 4MB memory served by a Sun 3/160 with 8 MB of memory running SunOS revision 4.0.1. On different machine configurations or under different loads, you are likely to see different results.

Notice that the overall execution time is 59:36 minutes. Note that more time was spent in system calls (231.6s) than in user functions (11.0u). Considering that most of the work should be performed in counting characters, this indicates an area for possible optimization.

Next, look at the number of page faults: 102349pf. This program has to reload many pages of the program from secondary storage.

A large number of page faults, as in the example above, is an indication that the program is accessing virtual memory in a very inefficient way.

Dynamic Analysis Using *tcov*



Dynamic Analysis Using *tcov*

tcov produces a test coverage analysis and statement-by-statement profile of a C or FORTRAN program. When a program in a file named *file.c* or *file.f* is compiled with the *-a* option, a corresponding *file.d* file is created. Each time the program is executed, test coverage information is accumulated in *file.d*.

tcov takes source files as arguments. It reads the corresponding *file.d* file and produces an annotated listing of the program with execution statistics in *file.tcov*. Each straight-line segment of code (or each line if the *-a* option to *tcov* is specified) is prefixed with the number of times it has been executed; lines which have not been executed are prefixed with *#####*.

Note: The profile produced includes only the number of times each statement was executed, not execution times; to obtain times for routines use *gprof()* or *prof()*.

tcov command format:

tcov [-options] srcfile

Dynamic Analysis Using *tcov*

```
patience% cc -a -o pager pager.c
patience% ls -l
total 27
-rwxr-xr-x 1 student1      24576  Mar  2 09:09 pager
-rw-r--r--1 student1        1586  Mar  2 09:03 pager.c
-rw-r--r--1 student1         125  Mar  2 09:09 pager.d
patience% pager < /usr/dict/words
    ....
    (output)
    ....
patience% ls -l
total 27
-rwxr-xr-x 1 student1      24576  Mar  2 09:09 pager
-rw-r--r--1 student1        1586  Mar  2 09:03 pager.c
-rw-r--r--1 student1         163  Mar  2 09:48 pager.d
patience% tcov pager.c
patience% ls -l
total 30
-rwxr-xr-x 1 student1      24576  Mar  2 09:09 pager
-rw-r--r--1 student1        1586  Mar  2 09:03 pager.c
-rw-r--r--1 student1         163  Mar  2 09:48 pager.d
-rw-r--r--1 student1        2940  Mar  2 10:05 pager.tcov
```

Dynamic Analysis Using *tcov*

The command, *cc –a –o pager pager.c* produces a statistics file named *pager.d* which is updated after the successful execution of *pager.*

The command:

tcov pager.c

produces a file named *pager.tcov* which contains the original source annotated with the execution statistics. Such statistics can point out areas of the program which are used heavily or not executed at all.

The file *pager.tcov* is displayed and discussed on the following pages.

Dynamic Analysis Using *tcov*

```
                /* pager.c — Program to be optimized.                    */
               /* Reads stdin and counts the occurrence of ASCII chars. */
               #define LINES   1000
               #define LINELEN 100
               #define NUMCHAR 128
               char *valloc(), *calloc();
               void countmem(), clearmem(), setcount(), printcount();

                main()
    1->        {
                  char **line_array;
                  int  *count_array;
                  int  lines;
                  line_array = (char **) calloc(LINES, sizeof(char *));
                  count_array = (int *) calloc(NUMCHAR, sizeof(int));
                  clearmem(count_array, NUMCHAR * sizeof(int));

 1000 ->          for (lines = 0; lines < LINES; lines++)
 1000 ->            line_array[lines] = valloc(LINELEN);

    1,    1000 ->     for (lines = 0; lines < LINES; lines++)
 1000 ->              if (!(gets(line_array[lines])))
##### ->                  break;

    1 ->          countmem(lines, line_array, count_array);
                  printcount(count_array);
                  exit(0);
               }
```

Dynamic Analysis Using *tcov*

The numbers before the arrows (e.g. 1000 –>) indicate the number of times the indicated block of code was executed. Lines preceded by ##### were never executed.

Notice that the *gets()* call never returns zero, hence the *if* statement always fails, so the break; is never executed. This means that the program exhausted *line_array* before it ran out of lines to read.

Notice also that the line containing the *valloc()* and the one containing *gets()* are executed the same number of times (with the same array index).

Dynamic Analysis Using *tcov*

```
                void                /* count chars in line_array */
                countmem(numlines, line_array, count_array)
                int  numlines;
                char **line_array;
                int  *count_array;
      1 ->      {
                    int  lines, chars;

    100 ->      for (chars = 0; chars < LINELEN; chars++)
    100,  100000 ->         for (lines = 0; lines < numlines; lines++)
 100000 ->              if (line_array[lines][chars] != 0)
                          setcount(count_array,
   7182 ->                        line_array[lines][chars]);
      1 -> }


                void                /* set memory to 0 for num char */
                clearmem(cp, num)
                char *cp;
      1 ->      {
                int num;
                    while (num--)
    512 ->              *cp++ = 0;
      1 ->      }


                void                /* increment the count array for c */
                setcount(count_array, c)
                int  *count_array;
                char c;
   7182 ->          {
                        count_array[c] += 1;
                    }
```

Dynamic Analysis Using *tcov*

The *countmem()* function has one loop embedded in another. Its action, *setcount()*, is executed few times relative to the number of times the inner loop is executed. The inner loop is executed 1,000 times for each execution of the outer loop, and the total (100,000) is very close to the number of page faults that the time output showed. This could be a clue.

Dynamic Analysis Using *tcov*

```
        void                /* print content of char count array */
        printcount(array)
         int  *array;
 1 ->    {
         int  c;

94 ->       for (c = ' '; c < '~'; c++) {
94 ->           if (!(c % 8))
12 ->               printf("\n");
94 ->           printf("\t%c %d", c, array[c]);
            }
 1 ->       printf("\n");
        }
```

Top 10 Blocks

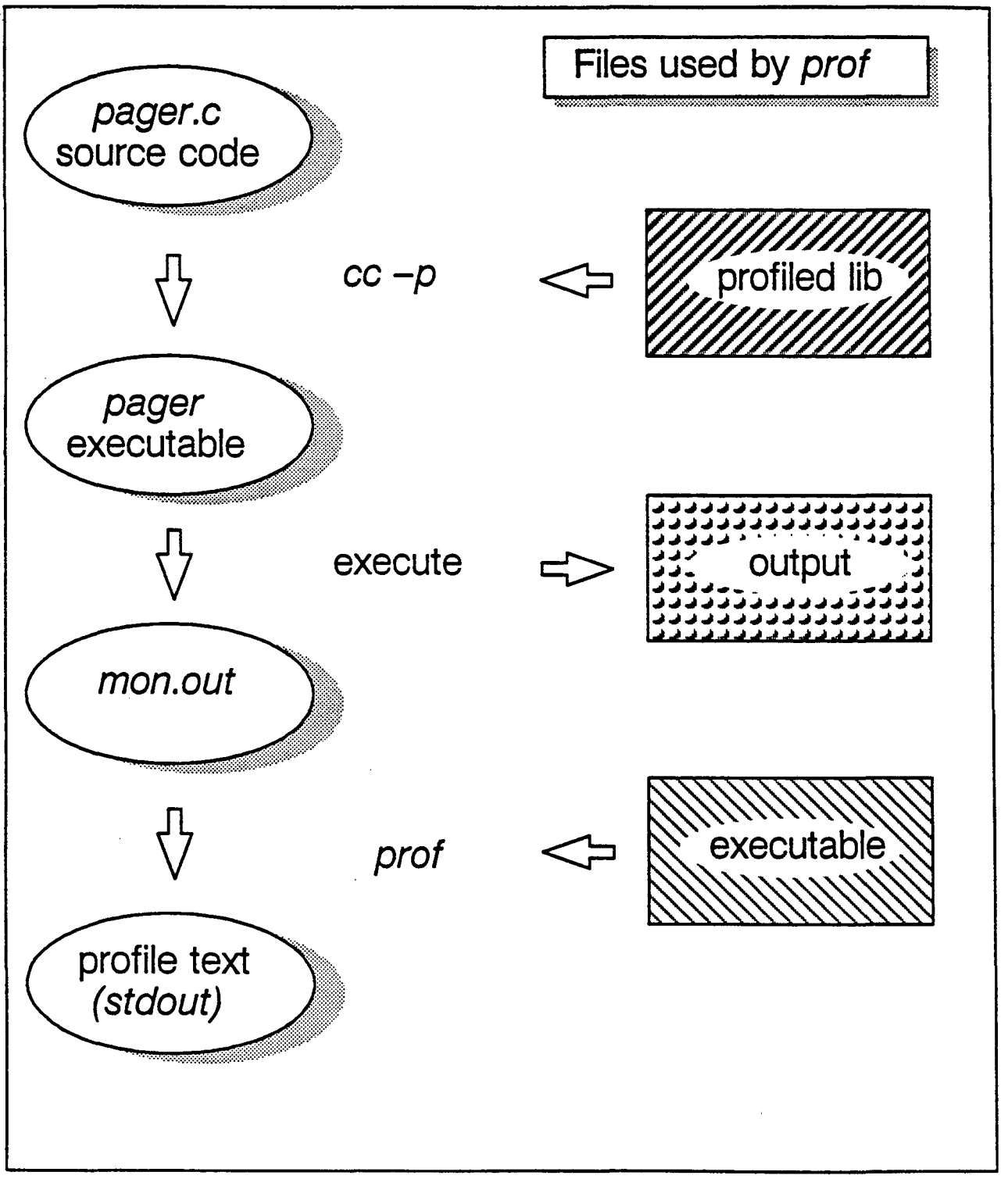
Line	Count
40	100000
41	100000
43	7182
61	7182
18	1000
19	1000
21	1000
22	1000
53	512
39	100

```
    25  Basic blocks in this file
    24  Basic blocks executed
 96.00  Percent of the file executed

219379  Total basic block executions
8775.16  Average executions per basic block
```

Dynamic Analysis Using *tcov*

The "Top 10" listing shows that lines 40 and 41 (the inner loop of *countmem()*) and lines 43 and 61 (the *setcount()* call) are ready for optimization by reducing the number of times they are executed.

Dynamic Analysis Using *prof*



Dynamic Analysis Using *prof*

prof is another post-processor which requires a specially-compiled source program. It produces a profile of the execution times of functions called. This reveals the execution-time hogs, which may not be the same as the most-executed functions revealed by *tcov.*

The target program must be compiled and linked with the *-p* option. This causes a profiled library to be used with the program, and a *mon.out* file to be produced when it exits. When *prof* is run with the executable program text, it matches up the symbols in the program to locations in the *mon.out* file and produces the output execution profile.

Then *prof* can be used to produce a human-readable version of this file. The output is written to standard out and can be redirected into a file for later use. It contains a summary of the number of calls and the amount of time used for every function in the program. The functions which use the most time are prime targets for recoding.

prof command format:

prof [-options] [image-file [profile-file]]

Example:

patience% **cc -p -o pager pager.c**

patience% **pager < /usr/dict/words**

patience% **ls**

mon.out pager pager.c

patience% **prof pager**

The output to this last command is shown on the next page.

Dynamic Analysis Using *prof*

%time	cumsecs	#call	ms/call	name
45.2	10.18	1	10179.14	_countmem
29.8	16.90	2502	2.69	_free
21.3	21.70			_getenv
1.2	21.98	504	0.56	_sbrk
0.7	22.14	1000	0.16	_memccpy
0.6	22.28			mcount
0.4	22.36	1005	0.08	_malloc
0.3	22.42			_realloc
0.2	22.46	7182	0.01	_setcount
0.1	22.48	107	0.19	__doprnt
0.1	22.50	1	20.00	_main
0.1	22.52	1000	0.02	_memalign
0.1	22.54	13	1.54	_write
0.0	22.54	1	0.00	__filbuf
0.0	22.54	2	0.00	__findbuf
0.0	22.54	1	0.00	__wrtchk
0.0	22.54	13	0.00	__xflsbuf
0.0	22.54	2	0.00	_bzero
0.0	22.54	2	0.00	_calloc
0.0	22.54	1	0.00	_clearmem
0.0	22.54	1	0.00	_exit
0.0	22.54	1	0.00	_finitfp_
0.0	22.54	1	0.00	_fstat
0.0	22.54	2	0.00	_getpagesize
0.0	22.54	1000	0.00	_gets
0.0	22.54	2	0.00	_ioctl
0.0	22.54	2	0.00	_isatty
0.0	22.54	107	0.00	_memchr
0.0	22.54	1	0.00	_on_exit
0.0	22.54	1	0.00	_printcount
0.0	22.54	107	0.00	_printf
0.0	22.54	1	0.00	_profil
0.0	22.54	1	0.00	_read
0.0	22.54	1000	0.00	_valloc

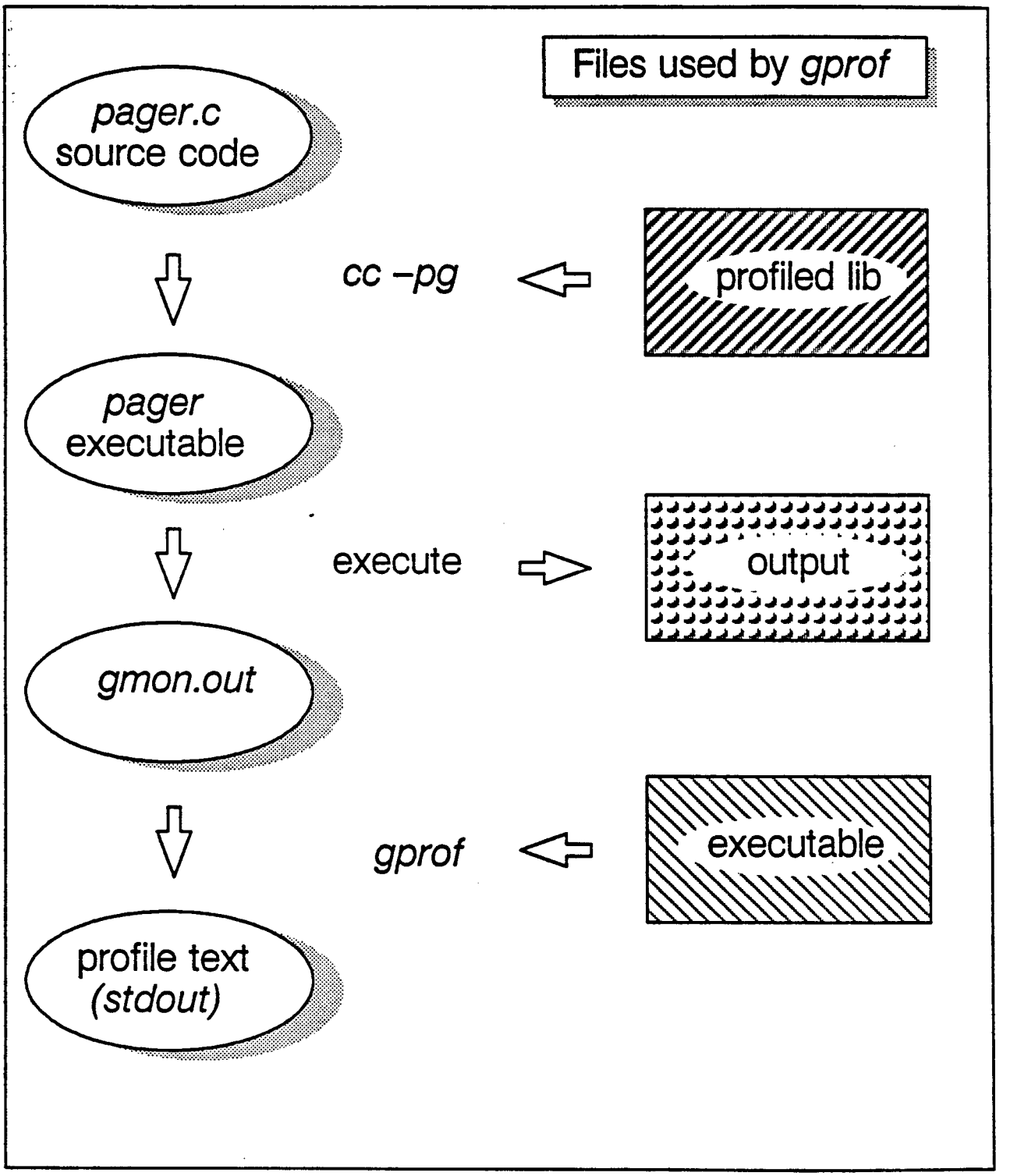
Dynamic Analysis Using *prof*

The *prof* output is a tabular listing of functions and their execution times. The *%time* column shows the percentage of total execution time used by each function. The *cumsecs* column shows the number of seconds used by this function and all the preceding ones. The *#call* column shows how many times this function was called, similar to *tcov*. The *ms/call* column shows the average time elapsed in each call. The *name* column shows the function name.

Many of the functions displayed are not explicitly in the source code. These are standard library calls, used by *calloc(), valloc(), gets(),* and *printf()* functions. Notice that the *profil()* function itself, which is used by *prof*, takes quite a bit of time.

This listing reveals that *countmem()* is the bottleneck. The next user function listed is *setcount()*. Both of these functions are candidates for optimization.

Dynamic Analysis Using *gprof*



Dynamic Analysis Using *gprof*

gprof provides more detailed profiling than *prof*. It summarizes execution times similarly to *prof*, but it also breaks down each function, as called from every other function. This level of detail can be used to find subroutines that are inefficient only in certain cases.

As with *tcov* and *prof*, the program has to be compiled and linked with a particular option: *-pg*. This causes a profiled library to be used with the program, and a *gmon.out* file to be produced after the executable program is run.

Running *gprof* on the program produces the human-readable profile output listing. The output is written to standard out and can be redirected into a file for later use. Similar to *prof* output, it contains the number of calls and the amount of time used for every function in the program. Along with the summary, it produces a detailed listing of every function and its descendants. This detailed information can be used to find functions which are called in inefficient ways by other functions. These are candidates for rewriting.

gprof command format:

gprof [-options] [image-file [profile-file]]

Example:

patience% **cc -pg -o pager pager.c**

patience% **pager < /usr/dict/words**

patience% **ls**

gmon.out pager pager.c

Dynamic Analysis Using *gprof*

patience% **gprof -b pager > outfile**

patience% **more outfile**

granularity: each sample hit covers 2 byte(s) for 0.08% of 25.26 seconds

index	%time	self	descendents	called/total called+self called/total	name	parents children index
						<spontaneous>
[1]	99.8	0.00	25.22		start [1]	
		0.02	25.20	1/1		_main [2]
		0.00	0.00	1/1		_on_exit [24]
		0.00	0.00	1/1		_finitfp_ [203]
		0.02	25.20	1/1		start [1]
[2]	99.8	0.02	25.20	1		_main [2]
		0.00	12.88	1000/1000		_valloc [4]
		12.18	0.02	1/1		_countmem [5]
		0.00	0.06	1000/1000		_gets [13]
		0.00	0.04	1/1		_printcount [17]
		0.00	0.01	2/2		_calloc [21]
		0.00	0.00	1/1		_clearmem [201]
		0.00	0.00	1/1		_exit [202]
		0.08	12.80	1000/1000		_valloc [4]
[3]	51.0	0.08	12.80	1000		_memalign [3]
		5.74	3.49	2000/2502		_free [6]
		0.14	3.44	1000/1005		_malloc [7]

Dynamic Analysis Using *gprof*

When *gprof* is run with the *-b* option a lot of descriptive text is suppressed. This text is useful when first starting out but soon becomes just more clutter. The output shown is edited from about 475 lines of the original (without the descriptions).

The *index* column is the number of the function being profiled in this block, and relates each function in the leftmost column to its own detail block.

The *%time* column shows the time spent in this function as a percentage of the total.

The *self* column shows the number of seconds spent in this function. The *descendants* column shows seconds spent in the subfunctions. The *called* column shows the count of nonrecursive calls for the function. The *name* column shows the function's position in the calling stack, with who calls whom.

Dynamic Analysis Using *gprof*

```
                0.00       12.88   1000/1000          _main [2]
[4]     51.0    0.00       12.88   1000               _valloc [4]
                0.08       12.80   1000/1000          _memalign [3]
                0.00        0.00      1/2             _getpagesize [198]
-------------------------------------------------------------------

                12.18       0.02      1/1             _main [2]
[5]     48.3    12.18       0.02      1               _countmem [5]
                 0.02       0.00   7182/7182          _setcount [19]
-------------------------------------------------------------------

                 1.44       0.88    502/2502          _morecore [9]
                 5.74       3.49   2000/2502          _memalign [3]
[6]     45.7     7.18       4.36   2502               _free [6]
                 3.42       0.00   2502/2502          _insert [8]
                 0.94       0.00   1499/1499          _delete [10]
-------------------------------------------------------------------
```

Dynamic Analysis Using *gprof*

Notice that *valloc() [4]* is a resource hog because it calls *memalign() [3].*

Dynamic Analysis Using *gprof*

granularity: each sample hit covers 2 byte(s) for 0.08% of 25.56 seconds

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
47.7	12.18	12.18	1	12182.03	12202.03	_countmem [5]
28.1	19.36	7.18	2502	2.87	4.61	_free [6]
13.4	22.78	3.42	2502	1.37	1.37	_insert [8]
3.7	23.72	0.94	1499	0.63	0.63	_delete [10]
3.3	24.56	0.84	1005	0.84	0.84	_demote [11]
1.2	24.86	0.30				mcount (142)
1.1	25.14	0.28	504	0.56	0.56	_sbrk [12]
0.5	25.28	0.14	1005	0.14	3.58	_malloc [7]
0.3	25.36	0.08	1000	0.08	12.88	_memalign [3]
0.2	25.42	0.06	1000	0.06	0.06	_memccpy [14]
0.2	25.46	0.04	107	0.37	0.41	__doprnt [15]
0.2	25.50	0.04				_moncontrol [18]
0.1	25.52	0.02	7182	0.00	0.00	_setcount [19]
0.1	25.54	0.02	502	0.04	5.21	_morecore [9]
0.1	25.56	0.02	1	20.00	25220.62	_main [2]
0.0	25.56	0.00	2502	0.00	0.00	_getfreehdr [25]
0.0	25.56	0.00	1499	0.00	0.00	_putfreehdr [193]
0.0	25.56	0.00	1000	0.00	0.06	_gets [13]
0.0	25.56	0.00	1000	0.00	12.88	_valloc [4]
0.0	25.56	0.00	107	0.00	0.00	_memchr [194]
0.0	25.56	0.00	107	0.00	0.41	_printf [16]
0.0	25.56	0.00	13	0.00	0.00	__xflsbuf [195]
0.0	25.56	0.00	13	0.00	0.00	_write [196]
0.0	25.56	0.00	2	0.00	3.58	__findbuf [20]
0.0	25.56	0.00	2	0.00	0.00	_bzero [197]
0.0	25.56	0.00	2	0.00	3.58	_calloc [21]
0.0	25.56	0.00	2	0.00	0.00	_getpagesize [198]
0.0	25.56	0.00	2	0.00	0.00	_ioctl [199]
0.0	25.56	0.00	2	0.00	0.00	_isatty [200]

Dynamic Analysis Using *gprof*

The *gprof* summary page is shown here, also edited somewhat from the original. It gives a little more detail than the *prof* summary.

Again, notice that *countmem()* is the heaviest time user, followed by *setcount()*.

Optimizing the Pager

patience% **time pager < /usr/dict/words**

....
(same output)
....

11.0u 231.6s 59:36 6% 0+968k 4+0io 102349pf+0w

Perform Fix Number 1

patience% **cc –o pager pager.c**
patience% **time pager < /usr/dict/words**

....
(same output)
....

8.2u 11.3s 3:56 8% 0+816k 2+0io 2043pf+0w
patience%

Perform Fix Number 2

patience% **cc –pg –o pager pager.c**
patience% **time pager < /usr/dict/words**

....
(same output)
....

0.9 u 0.0s 0:01 80% 0+88k 1+2io 2pf+0w
patience%

Optimizing the Pager

Fix number 1.

The program places each line of text on a separate page of memory, because the *valloc()* call is used to get each line buffer. The *countmem()* function then goes through the line array repeatedly, counting one character from each line. This causes the program to page fault many times. It might help to change the order of the loops in *countmem()*:

```
for (lines=0; lines < numlines; lines++)
    for (chars=0; chars < LINELEN; chars++)
        if (line_array[lines][chars] != 0)
            setcount(count_array,
                line_array[lines][chars]);
```

The case study shows this fix's effect on execution time. Speedup is by about 20:1. Equally important, the number of page faults is reduced by 50:1, and the system time is much lower.

Fix number 2.

Each input line needn't occupy an entire page of memory. We can replace the *valloc()* call with *calloc()* in the *main()* routine using the command:

```
line_array[lines] = calloc(LINELEN,1);
```

Now there are only two page faults and the execution time is less than one second.

Optimizing the Pager

patience% **gprof pager**

```
....
-----------------------------------------------

                0.48    0.02       1/1          _main [2]
[3]     73.5    0.48    0.02         1        _countmem [3]
                0.02    0.00   7182/7182      _setcount [13]

-----------------------------------------------

....
-----------------------------------------------

                0.02    0.00   7182/7182      _countmem [3]
[13]     2.9    0.02    0.00      7182        _setcount [13]

-----------------------------------------------

....
granularity: each sample hit covers 2 byte(s) for 0.08% of 25.56 seconds

  %         cumulative   self              self     total
 time         seconds   seconds    calls  ms/call  ms/call  name
 53.3            0.48     .48          1   480.08   500.08  _countmem [3]
 24.4            0.70     0.22                              mcount (136)
  4.4             .74     .04       1002     .04      .10  _calloc [4]
```

Optimizing the Pager

Running *gprof* on the version of *pager* incorporating fixes 1 and 2 shows that *countmem()* is still the bottleneck. This is because it calls *setcount()* to perform a trivial function.

Optimizing the Pager

Fix Number 3

patience% **cc -pg -o pager pager.c**
patience% **time pager < /usr/dict/words**

...
(same output)
...

0.7u 0.2s 0:01 69% 0+88k 3+2io 6pf+0w

patience% **gprof pager**

....

		0.46	0.00	1/1	_main [2]
[3]	71.9	0.46	0.00	1	_countmem [3]

....

granularity: each sample hit covers 2 byte(s) for 0.08% of 25.56 seconds

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
59.0	0.46	.46	1	460.08	460.08	_countmem [3]
18.0	0.60	0.14				mcount (136)
5.1	.64	.04	1	40.01	640.07	_main [2]

Optimizing the Pager

Fix number 3.

Replacing the *setcount()* function with an in–line macro:

```
#define setcount(ca,c) (++((ca)[c]))
```

produces the results shown on the previous page. Notice that the *countmem()* function takes about the same amount of time without the added *setcount()* execution time.

Monitoring Memory Usage: *vmstat()*

procs *(Current Process States)*	**r** Runnable **b** Blocked(on I/O or paging) **w** Runnable but swapped out
memory *(Virtual Memory Use*)*	**avm** Active virtual Kbytes **fre** Size of free list **averaged over 20 sec.*
page *(Paging and Swapping activity)*	**si** Swap-ins **so** Swap-outs **pi** Page-ins(per sec avg over 5 sec) **po** Page-outs(per sec avg over 5 sec) **fr** Pages freed(per sec) **de** Anticipated pages scanned(per sec) **sr** Reclaim pages scanned(per sec)
disk (operations per sec)	**x0-3** Controllers 0 to 3
faults (Trap Interrupt Rate)*	**in** Non-clock interrupts **sy** System calls **cs** Context switches
cpu (CPU usage percentage)	**us** *User processes* **sy** *System* **id** *Idle*

```
patience% vmstat -S
procs      memory          page                    faults                  cpu
r b w   avm  fre    si so  pi po fr de  sr s0 s1 s2 s3   in  sy   cs us sy id
0 0 0  2296 5376   40 70   0  4  4 72   2  1  0  0  0   10  91   11  5  3 92
```

Monitoring Memory Usage: *vmstat()*

The *vmstat* utility displays a set of statistics related to virtual memory. It reports on processes and also memory, disk, and cpu usage. The amount of virtual memory in use, and the number of page and swap in/outs is displayed. When a system is operating under load this information can be used to track program execution and find clues about why they failed.

The –*S* option reports on process swapping instead of more paging details. The *procs*, *memory*, and swapping/paging fields provide useful information which identifies the effects that processes have on memory.

Monitoring Memory Usage: *vmstat()*

```
patience% echo 10000000 10 | getmem&
[1] 411 412
getmem: get 10000000 bytes then sleep 10 sec.
```

```
patience% vmstat -S
procs     memory          page                    faults                  cpu
r b w   avm   fre    si so  pi po fr de sr x0 x1 x2 x3  in sy  cs  us sy id
0 0 0  3752 4384  40 70  0  4  4 72  2  1  0  0  0   10  95 11  5  3 92
```

```
patience% vmstat -S

procs     memory          page                    faults                  cpu
r b w   avm   fre    si so  pi po fr de sr x0 x1 x2 x3  in  sy  cs us sy id
1 1 0 13536  624  40 73  0  4  4 80  2  1  0  0  0   10  96 11  5  3 92
```

Monitoring Memory Usage: *vmstat()*

To further demonstrate *vmstat's* capabilities, the program *getmem* (which we discussed in the module on *Processes and Memory Management*), is run in the background on a large chunk of memory.

Most Sun's have 8 megabytes of memory or less, so this command forces swapping and paging activity. The amount of memory utilized is displayed in the *avm* field. Notice that the response time of the system dips when the swap fields *si* and *so* begin to increment. This is the result of processes being swapped out to secondary storage.

Monitoring Memory Usage: *vmstat()*

```
patience% vmstat -S
procs      memory          page                    faults                  cpu
r b w   avm    fre  si so  pi po fr de sr x0 x1 x2 x3  in  sy  cs us sy id
2 0 0 12112  776  46 78  0  4  4 72  2  1  0  0  0  10  96  11 5  3 92


patience%
[1]    Done                 getmem


patience% vmstat -S
procs      memory          page                    faults                  cpu
r     b w   avm   fre   si  so pi po fr de sr x0 x1 x2 x3  in  sy cs us sy id
311 0 0   3552 4936  48 79  0  5  4 72  2  1  0  0  0  10  99 11  5  3  92
```

Monitoring Memory Usage: *vmstat()*

Module 12
SCCS

Objectives: Upon completion of this module, the student should be able to:

- Create an *SCCS* database. Update and retrieve any version of a text file using the commands: *sccs create, sccs edit and sccs get.*
- Apply the *SCCS* concepts and techniques of: version and release numbering, creating deltas and branches, using ID keywords, creating s–files and comma files.
- Use the commands: *sccs delget, sccs info, sccs check, sccs diffs, sccs what, sccs delta,* and *sccs sccsdiff.*
- Use *SCCS* in conjunction with *make.*

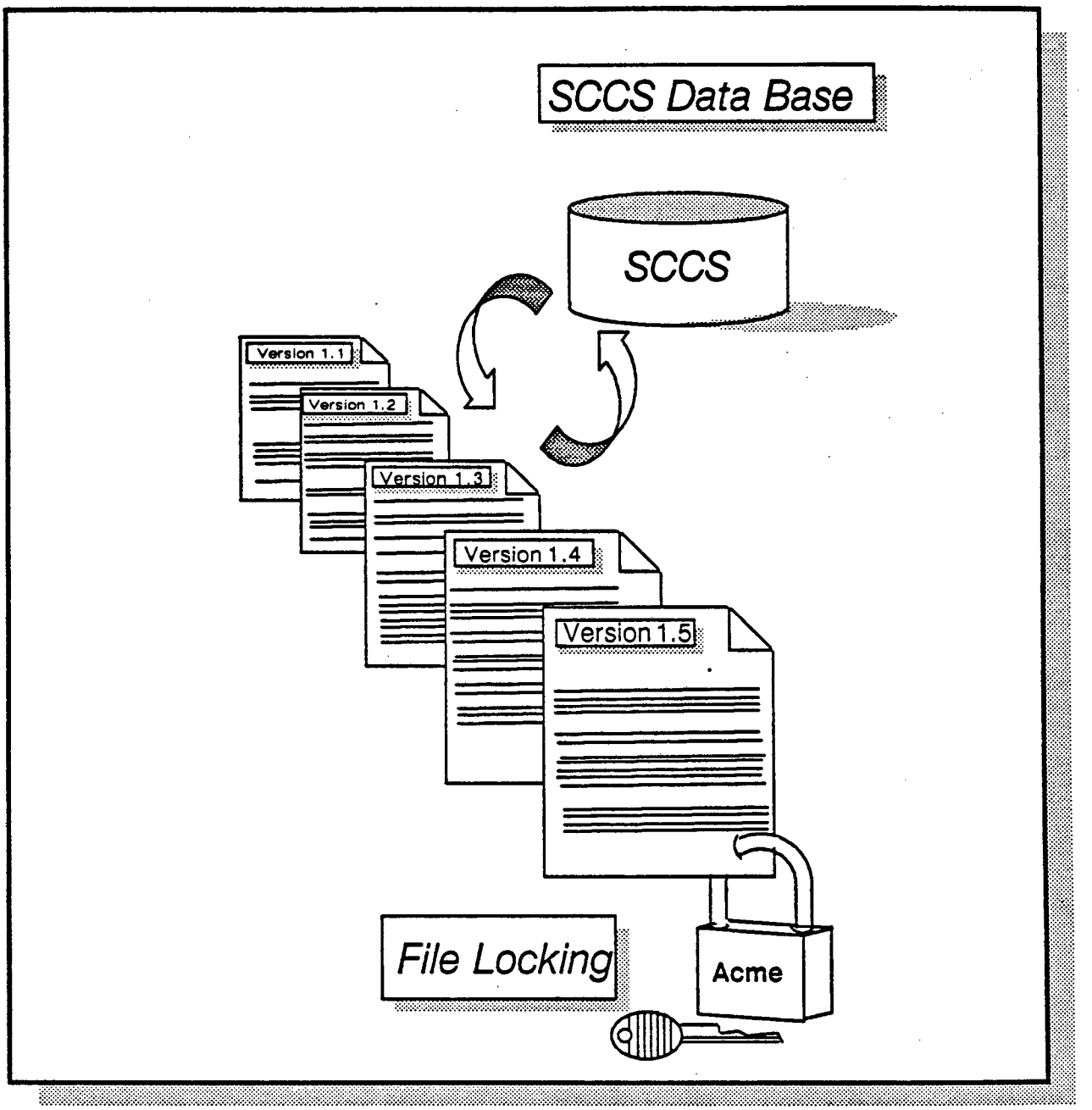
Evaluation:

Perform Lab 10 to 90% proficiency.

Reference information:

- *Doing More with SunOS: Beginner's Guide* (p/n 800–1710–10), Section 6.3.
- *SCCS – Source Code Control System* in *Programming Utilities and Libraries* (p/n 800–1774–15).
- *SunPro, The Sun Programming Environment,* Sun Microsystems, Inc. (p/n FF144/20K).

Introduction



Introduction

SCCS is a file version maintenance utility. *SCCS* provides services essential to any program development project no matter how large or small.

SCCS Benefits

SCCS takes custody of text files and, when changes are made, identifies and stores those changes in a database. By managing update information, *SCCS* can retrieve the original version of a file, or any revision. *SCCS* also provides a mechanism for logging an explanation with each change and can generate an audit report showing the history of the development of a program.

File Protection

The beneficial side-effect of enabling the retrieval of previous versions of a file is an additional level of file protection. For example, if the command: *rm -f * .o* is executed by accident, the space between the '*' and the '.' causes object files *and* all the other files in the directory to be removed. But since you have all of the files under *SCCS*_control, they can all be restored.

Using *SCCS,* recent versions of the files can instead be retrieved from the database, in place of reverting to a tape archive.

Project Management

SCCS is also a powerful tool for organizing software projects involving many engineers. Such projects typically involve several engineers with independent copies of a common set of files. These copies are then merged together at various stages of the project. When two engineers make changes to the same file, merging those changes can be a difficult task. Using *SCCS,* each engineer can access the *SCCS* database for the project and let *SCCS* provide the necessary coordination.

SCCS Terminology

Term	Description
s–file	A single file in the SCCS database containing all the different versions of a text file
delta	A set of changes to a text file
SccsID	A number that represents a delta
branch	A separate line of development independent to development along the main trunk.
ID keywords	Special symbols that can be placed into a text file for expansion by SCCS to help identify the file

SCCS Terminology

There a few terms that need to be defined in order to describe the *SCCS* system.

s–file

The *s–file* is a single file in the *SCCS* database directory containing information about all the different versions of a single text file. Additional administrative information is also stored in this file, including the comments which describe the reason for each update. This provides one common place for all of the *SCCS* information on a single text file.

delta

A *delta* (equivalent to a version number) is one set of changes inserted into an *s–file*. While a *delta* is only one set of changes, it is generally applied with respect to all previous *deltas*. However, it is possible to get a file with one or more *deltas* eliminated, this is equivalent to retrieving an earlier version.

SID (SccsID)

An *SID* is a number that represents a *delta*. This is a two–part number (e.g. *1.2*) where the first part is the *release number* (i.e. *1*), and the second part is a *level* (i.e. *2*). Since all *deltas* to a file are normally applied, it is common to refer to a *delta* as a version number of a file.

SCCS Terminology

Term	Description
s–file	A single file in the SCCS database containing all the different versions of a text file
delta	A set of changes to a text file
SccsID	A number that represents a delta
branch	A separate line of development independent to development along the main trunk.
ID keywords	Special symbols that can be placed into a text file for expansion by SCCS to help identify the file

SCCS Terminology

branch

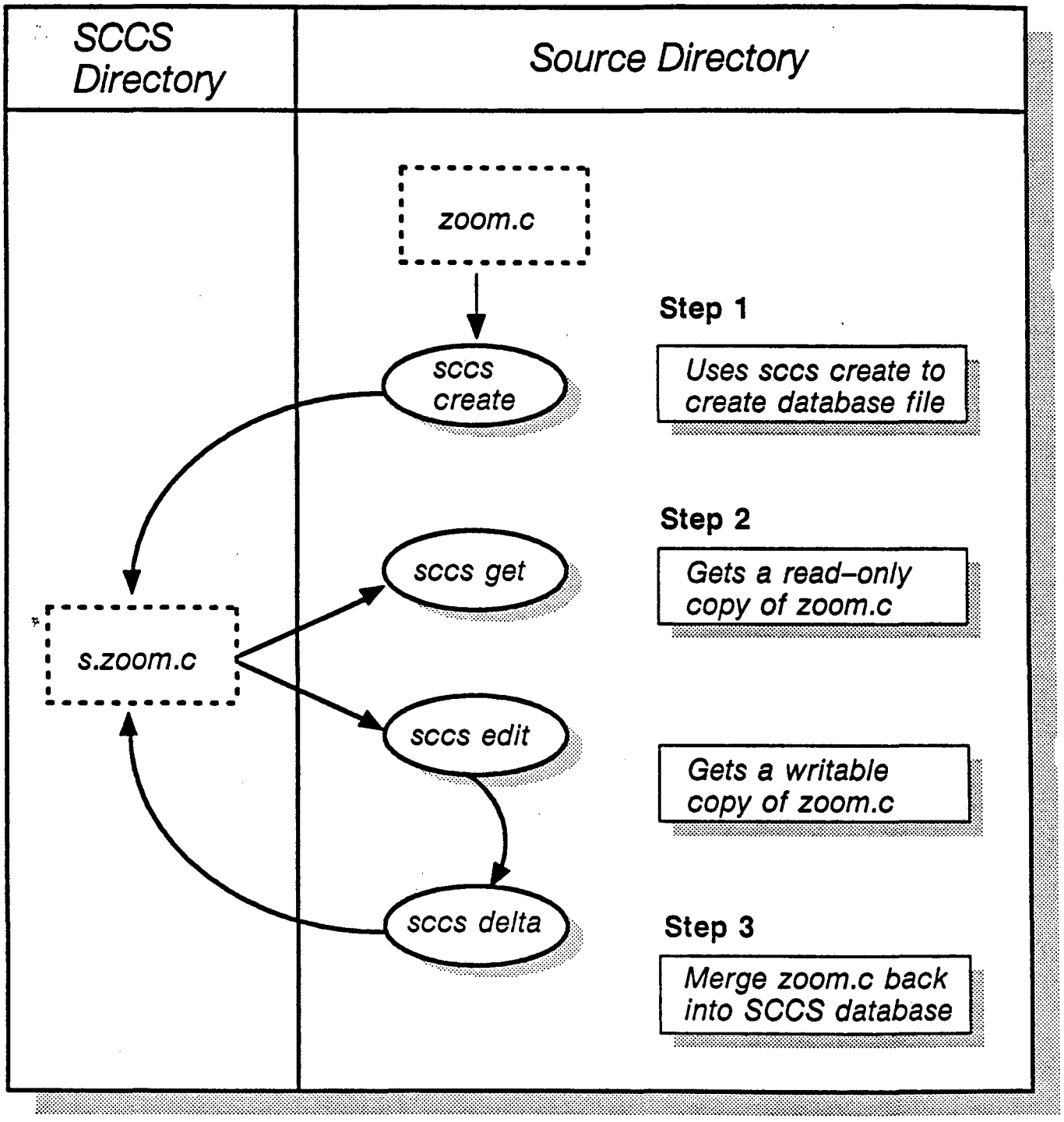
A *branch* is a line of development independent of the main line. The most common uses of branches are for bug fixes, custom versions, and experimental development.

ID Keywords

ID keywords are special symbols that can be included in text files. When files are retrieved for production purposes using *sccs get* (i.e. not for development with *sccs edit*), these keywords are expanded by *SCCS*. This allows information such as the name and *delta* number of the file to be compiled into programs.

ID keywords are not expanded when files are retrieved for editing. It is a good idea to include *ID keywords* in all files in order to incorporate the version number(s) of the source file(s) into the executable modules themselves. This is accomplished by expanding the keywords with *sccs get* before compiling.

Basic SCCS Operations



Basic SCCS Operations

SCCS is designed for ease of use and to augment other software development tools (e.g. *make*). The *SCCS* database is stored as a set of files in a subdirectory named *SCCS*. For each source file, there exists a corresponding file in the *SCCS* directory.

Three steps are added to the program development process to set-up *SCCS*:

Step 1: Creating the SCCS database

Before *SCCS* can be used, entries in the database called *s-files,* must be created. This is performed for each text file (source code, documentation, etc.) being developed. All other *SCCS* services operate on *s-files.*

To create the database, first create the *SCCS* directory and then invoke the command *sccs create* for each source file. For example:

sccs create zoom.c

You do not have to *cd* to the SCCS directory first because SCCS will automatically prefix the filename with "SCCS/s.".

Step 2: Retrieving source files for compilation or editing.

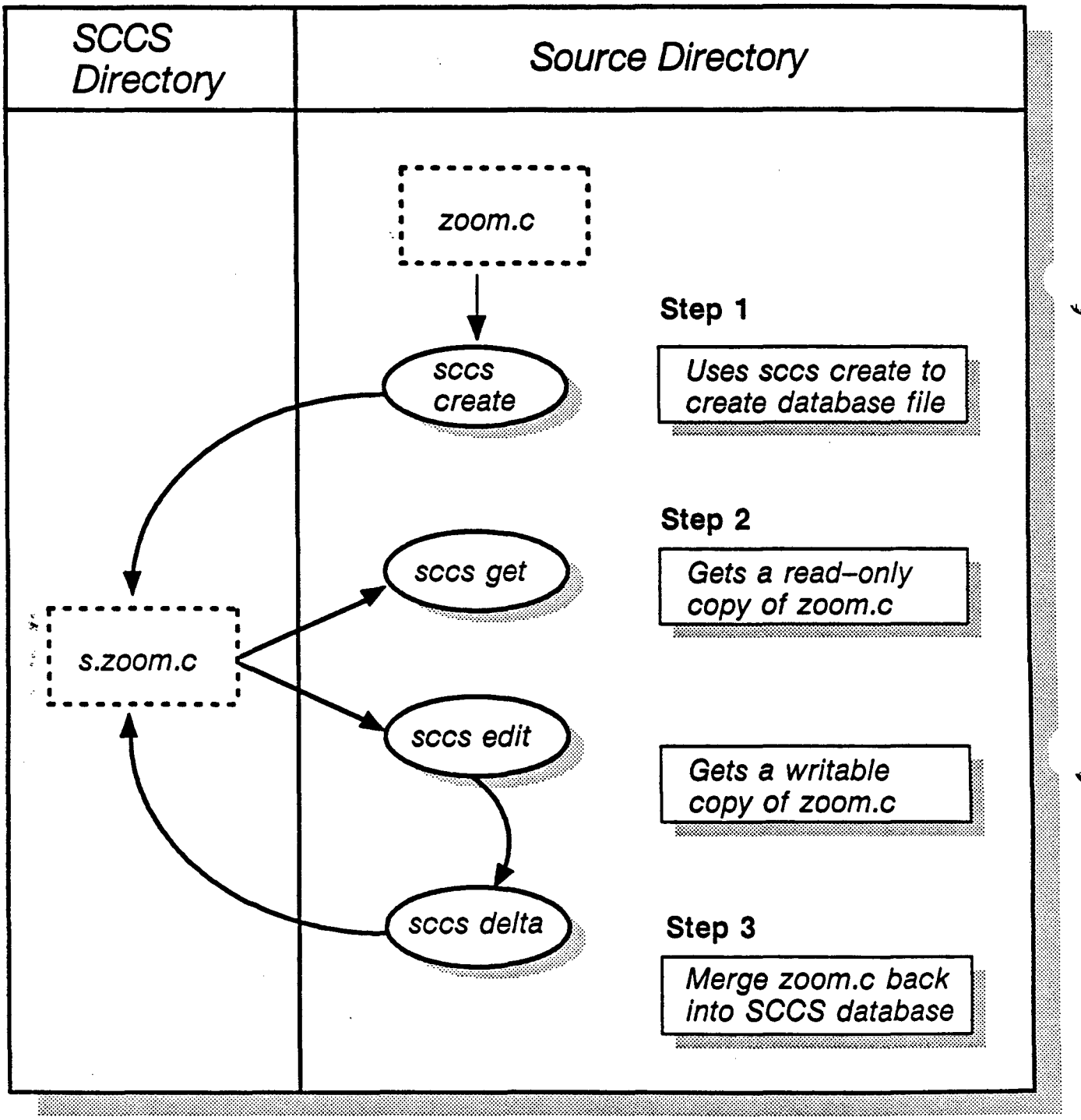
a. Get a read-only copy from an s-file (sccs get)

By default, the latest version, incorporating all *deltas*, is retrieved. This copy is not intended to be to edited or changed in any way; it should be used only for compiling or printing (the file's write permissions are disabled).

To retrieve the latest version of the C source code for compilation use the command *sccs get.* For example:

sccs get zoom.c

Basic SCCS Operations



Basic SCCS Operations

Step 2: Retrieving source files for compilation or editing (con't.).

b. Get a copy from the s-file for editing (sccs edit)

By default, the latest version, incorporating all *deltas*, is retrieved. This operation is similar to the previous *get* operation except that this copy of the file is intended to be changed and then incorporated back into the *s-file*. Only one person can be editing an *s-file* at one time, and *SCCS* enforces this policy. This is essential for coordinating software projects where several engineers share a set of sources. File system write permissions are granted only to the person who retrieved the *s-file*.

To obtain a source file for editing use the command *sccs edit*. For example:

sccs edit zoom.c

Step 3: Incorporate changes back into the s-file (sccs delta)

This command produces a new *delta* (and *delta* number) which is stored in the *s-file* along with user supplied comments indicating the reason for the *delta*. Incorporating a *delta* is the companion operation to the previous *edit* operation.

To make a *delta*, use the command is *sccs delta*. For example:

sccs delta zoom.c

Note that only the user who performed the *sccs edit* can execute an *sccs delta*.

SCCS in Action

useful.h

```
/*  some useful definitions          */
/*        %W% %G%                    */

/* so I can say for ever and get an infinite loop */
#define  ever     (;;)

#define  when     break; case
#define  wlse     break; default
#define  TRUE     1
#define  FALSE    0

/* the name used to invoke the program */
extern   char  *arg0;
```

SCCS in Action

useful.h

The programs *convert.c* (next page) and *useful.h* may be used to convert base ten numbers into numbers of other bases (binary, octal, hex). It is with these two programs that we will demonstrate the capabilities of *SCCS.*

The header file *useful.h* contains some C preprocessor definitions. The *when* and *wlse* are used in *main()'s* switch statement. These two symbols simplify case labels since most alternatives end with a *break* statement. The definition of *ever* writes an infinite loop. Such loops are usually terminated by conditions detected somewhere inside the loop.

useful.h also contains some SCCS ID keywords which will be explained later in the module.

SCCS in Action

convert.c

```
/* main     example program to convert base ten numbers to
 *          different bases
 * usage    print command line usage and exit
 * printit  prints number in different bases
 */

static char SccsID[] = "%W% %G%";

#include <stdio.h>
#include "useful.h"

char
        synopsis[] = "usage: %s [-both] number\n",
        bflag = 0,      /* binary output      */
        oflag = 0,      /* octal output       */
        tflag = 0,      /* decimal output     */
        hflag = 0,      /* hex output         */
        *arg0;

main(argc, argv)
int argc;
char **argv;
{
        int number;
```

SCCS in Action

convert.c

Program Function Calls: convert.c

The program consists of three functions. *Main()* performs the command line processing. If there is an error in the command line options, the function *usage()* is called to print the command's synopsis, and exits with an error code returned to the shell. The number conversion and output is performed by the function *printit()*.

Note the declaration and initialization of the variable *SccsID*. The symbols %*W*% and %*G*% are *ID keywords*. The expansion of these keywords will be demonstrated later in this module.

The *synopsis* variable indicates that the program recognizes four command line options. These are:

–b	binary output
–o	octal (base eight) output
–t	decimal (base ten) output
–h	hexadecimal (base 16) output

The flag variables are set when the corresponding option is found on the command line.

SCCS in Action
convert.c (con't.)

```
        /* command line processing */

        arg0 = *argv;
        while (*++argv)
        {
           if (**argv == '-')
           {
              while (*++*argv)
              {
                 switch (**argv)
                 {
                    when 'b':
                       ++bflag;
                    when 'o':
                       ++oflag;
                    when 't':
                       ++tflag;
                    when 'h':
                       ++hflag;
                    wlse:
                       usage(1);
                 }
              }
           }
           else
           {
              number = atoi(*argv);
              printit(number);
           }
        }
     } /* end of main() */
```

SCCS in Action
convert.c (con't.)

The command line processing section shows how each of the flags is set. The outer *while* statement loops over each command line parameter. If a parameter begins with a dash ('–'), a nested *while loop* examines the individual flag characters. Thus, one dash can be used to set several flags. For example, the parameter *–oh* causes the program to produce octal and hexadecimal output.

The *switch* statement distinguishes among the different flag characters. For example, when the character is a *'b'*, the variable *bflag* is incremented. In the outer *while loop*, if the argument does not begin with a dash, it is assumed to be a number to be converted.

The C library function *atoi()* converts the ASCII string to the internal format of an integer. This number is then passed to the routine *printit()* for display in the requested bases.

SCCS in Action
convert.c (con't.)

```
/* print the specified number in the desired bases */
printit(number)
int number;
{
        if (bflag) /* complicated so don't do it yet */
           { }
        if (oflag)
           printf("%o\n", number);
        if (tflag)
           printf("%d\n", number);
        if (hflag)
           printf("%x\n", number);
        return;
} /* end of printit() */

/* print command line syntax and exit */
usage(code)
int code;
{
        fprintf(stderr, synopsis, arg0);
        exit(code);
} /* end of usage() */
```

SCCS in Action
convert.c (con't.)

The routine *printit()* prints out the value passed in through the parameter *number* in the bases specified by the various flags.

The C library *printf()* function converts the variable *number* to octal, decimal, and hexadecimal automatically, as shown. However, to print the value in binary requires the conversion explicitly by the program.

When an error is detected during command line processing, *main()* calls the function *usage()* to display the command line syntax and exit.

This provides a consistent way of reporting errors. Note that it also prints the variable *arg0* which contains the name of the invoking command itself.

Creating the SCCS Database

```
patience% make convert
cc      -sun3 -o convert convert.c
patience% convert -h 37
25
patience% convert -o 53
65

patience% mkdir SCCS
patience% sccs create convert.c useful.h

convert.c:

useful.h:

SCCS/s.convert.c:
1.1
71 lines

SCCS/s.useful.h:
1.1
13 lines

patience% ls -a
,convert.c      .       SCCS            convert.c
,useful.h       ..      convert         useful.h

patience% convert -both 117
165
117
75

patience% rm ,convert.c ,useful.h

patience%
```

Create SCCS data base

Creating the SCCS Database

The two shell-level commands, *mkdir* and *sccs create,* create the database. Checking is then performed, via the *ls* command, to verify the information in the database. Note that SCCS did not delete any files.

The initial writing of the source files *useful.h* and *convert.c* is completed with a text editor. Note that *SCCS* can be used for any text files, not just those containing C source code. For example, all of Sun Microsystems' manual pages are under *SCCS* control.

Note that when the *mkdir* command is run, *SCCS* is spelled with capital letters. This is essential. Note also that it is possible to make entries for many text files with one *sccs create* command.

The *sccs create* command creates an entry for each of the files we want to include in a program. In general, anywhere the *sccs* command accepts a file name, a list of files can be used and the operation will be performed on each file separately. The *delta* is displayed, with the file size in lines. This is in accordance with the text-oriented philosophy of *SCCS. (SCCS* can manage any kind of text files, not just source code).

As the *sccs create* command is executing, *SCCS* verifies that the correct steps were performed: "s." files created (i.e. "s-dot files") in the SCCS directory, originals copied to "," files (i.e. "comma files"), and *sccs get* performed on each.

Now it is safe to delete the comma files created by *SCCS* using the *rm* command.

Creating and Editing Deltas

```
patience% sccs edit convert.c
1.1
new delta 1.2
71 lines

patience% vi convert.c
```

add error message: "binary output not yet implemented" in *printit()*.

```
patience% make convert
cc      -sun3 -o convert convert.c

patience% convert -b 127
convert: binary output not yet implemented

patience% ls -a
.       SCCS            convert
..      convert.c       useful.h

patience% sccs delta convert.c
comments?  added error message
1.2
3 inserted
1 deleted
70 unchanged

patience%  ls -a
.       SCCS            useful.h
..      convert

patience%
```

Creating and Editing Deltas

The commands to *edit* and *create deltas* are frequently used. The first step is to retrieve a writable copy of the source file:

sccs edit convert.c

This copy can be edited, compiled, and tested using customary program development techniques. Here, the *vi* editor is used to edit the file and *make* is used to compile and load the program. The edit–compile–test cycle is repeated as needed.

Once a program performs satisfactorily, it is time to create a new *delta:*

sccs delta convert.c

SCCS creates a new *delta* number and prompts for the reason for the change. *SCCS* then merges these changes back into the *s–file* and deletes the text file.

The latest version of the file is retrieved with the command:

sccs get convert.c

Note that after an *sccs edit* and when you *sccs delta*, the edited version is removed after the update is made.

Creating and Editing Deltas

```
patience% sccs delget convert.c
comments? added binary output not available message
1.2
3 inserted
1 deleted
70 unchanged
1.2
73 lines

patience% ls -l
 total 36
drwxr-xr-x      2 guru  512     Aug  4 22:44 SCCS
-rwxr-xr-x      1 guru  32768   Aug  4 22:43 convert
-r--r--r--      1 guru  1360    Aug  4 22:44 convert.c
-r--r--r--      1 guru  282     Aug  4 22:41 useful.h

patience% sccs edit convert.c
1.2
new delta 1.3
73 lines

patience% vi convert.c
```

(Note: modify output to display the base with the number)

```
patience% make convert
cc      -sun3 -o convert convert.c

patience% convert -h 35
 hex    0x23

patience%
```

Creating and Editing Deltas

The previous pages showed that *sccs delta* removes the copy from the current directory. However, because it is often helpful to have a copy of the source file in the working directory, there are two short cut methods to create a *delta* and retrieve a file in one step:

sccs delget

or

sccs deledit

This example shows the output assuming we had performed an *sccs delget* instead of an *sccs delta.* Note from the output of the *ls* command that *SCCS* does not grant write permission on files that have been retrieved with the *get* command.

The output of the number conversion program is still incomplete. It would be clearer if each number were displayed with its base. Therefore, another change must be made to the file *convert.c.* First, a writable copy is retrieved from *SCCS* using the command:

sccs edit convert.c

Next, the *vi* editor is used to make the changes to the file.

make is used to compile and link the program.

Now the executable is ready to be tested.

Obtaining Information from SCCS

```
patience% sccs info
convert.c: being edited: 1.2 1.3 student1 88/08/04 22:46:21
patience% sccs check
convert.c: being edited: 1.2 1.3 student1 88/08/04 22:46:21
patience% echo $STATUS
1
patience% sccs tell
convert.c
patience%
```

Obtaining Information from SCCS

The examples on the previous pages have demonstrated how to create and maintain an *SCCS* database. Now it is time to see which *SCCS* commands can be used to monitor this activity.

SCCS lists files currently retrieved for editing with the command:

sccs info

This displays who has retrieved the files and when. It is particularly useful when several people share the same files. The *info* command allows a team member to find out if a file is already being worked on.

sccs check

This command is essentially the same as the *info* command. The difference is that the *check* command is silent (produces no output) if no files are being edited and returns an error code to the shell when it completes. This makes it useful in shell scripts and makefiles.

sccs tell

prints a list of files that are being edited. Here's a typical application of the *tell* command:

sccs delta 'sccs tell'

Here, the shell's command substitution feature applies the *delta* command to the filenames generated by the *tell* command.

With one command it is possible to update every file in the *SCCS* database that has been retrieved for editing.

The *info, check,* and *tell* commands have two options:

-b	ignore branches (discussed later)
-u user	list only files edited by a specific user

The *user* argument defaults to the *ID* of the person who issued the command.

Obtaining Information from SCCS

```
patience% sccs delget 'sccs tell'
comments? added labels to output
1.3
3 inserted
3 deleted
70 unchanged
1.3
73 lines

patience% make convert
cc      -sun3 -o convert convert.c

patience% convert -oth 56
octal 070
decimal 56
hex 0x38

patience% sccs diffs convert.c

------- convert.c -------
59c59
<       printf("%o\n", number);
---
>       printf("octal\t0%o\n", number);
61c61
<       printf("%d\n", number);
---
>       printf("decimal\t%d\n", number);
63c63
<       printf("%x\n", number);
---
>       printf("hex\t0x%x\n", number);
patience%
```

Obtaining Information from SCCS

After performing a *delta* and retrieving a read-only copy of the file with the *sccs delget* command, *convert.c* is compiled, linked, and tested again.

sccs diffs

diffs compares the current version which has been *sccs edit*ed with a version *delta*ed previously. The -C flag is used to specify which previous version is to be compared.

Note that the less-than signs ("<") denote lines from the latest revision before the most recent *sccs edit* and the greater-than signs (">") denote lines from the file currently open for editing.

Using ID Keywords

```
patience% sccs what convert convert.c useful.h
convert:
        convert.c       1.3 8/4/88
convert.c:
        convert.c       1.3 8/4/88
useful.h:
        useful.h        1.1 8/4/88 */
```

Using ID Keywords

Among SCCS's useful features is its ability to insert identifying strings into text files and into the executable modules created from those text files. This provides a mechanism both for finding files by name and genealogy, and to identify strays. Termed *ID keywords*, these strings by convention begin with '*@(#)*' and are displayed with the command:

sccs what

For each such string found, the name of the file and the string are printed.

As used here, it also shows the delta numbers of the two source files. Note that *sccs what* has revealed that *delta 1.3* of the text file *convert.c* has been used to make the executable *convert*.

ID keywords have been used to identify the text files *convert.c* and *useful.h*. As can be seen by reviewing the source code, the technique for incorporating these strings into the executables is to put them in the corresponding *.c* files as data initialization strings. In *header (.h)* files they are included as comments. This keeps the data space of the program small. If keywords are put in header files as strings, then each string must have a unique name (e.g. each header file cannot define SccsID.)

Here is a partial list of some useful *ID Keywords:*

Keyword	Expansion
%Z%	@(#)
%M%	module name
%I%	highest SID applied
%W%	shorthand for %Z%%M%<tab>%I%
%G%	date of delta of highest SID applied
%R%	current release number (first part of SID)

Creating a New Release

patience% **sccs edit -r2 SCCS** ← *Create a New Release*

SCCS/s.convert.c:
1.3
new delta 2.1
73 lines

SCCS/s.useful.h:
1.1
new delta 2.1
13 lines

patience% **sccs delta SCCS**
comments? **starting new release**

SCCS/s.convert.c:
2.1
0 inserted
0 deleted
73 unchanged

SCCS/s.useful.h:
2.1
0 inserted
0 deleted
13 unchanged

patience% **ls**
SCCS convert

patience%

Creating a New Release

Once all the desired features have been added to a program and its operation has been verified, it is ready for release. Releasing a program requires copying it to a directory where all the users on a system can access it, e.g. */usr/local/bin*. It may also be necessary to produce distribution media, if the program is to be sold as a product. (*Note:* The process of releasing software does not directly involve *SCCS*.)

The use of *SCCS branches* also supports the post-program development phases of adding features and fixing bugs following release.

After releasing a product, it is recommended that *SCCS* is set up with a new release number for all text files used to create the product. For example, to begin work on release *2*, use the command:

sccs edit -r2 SCCS

This instructs *SCCS* to retrieve for editing, all of the files in the database. This command also specifies that the next *delta* will be in release *2* (i.e. *2.1*). Specifying *SCCS* (in capital letters) as the file name for any *SCCS* command indicates that the operation should be performed on all files in the database directory. *delta 2.1* can then be created by checking all the files back into *SCCS*.

The */s* command confirms that *SCCS* has removed all the text files after creating the *deltas*.

Working on the New Release

```
patience% make convert
sccs get -s convert.c -Gconvert.c
cc       -sun3 -o convert convert.c
convert.c: 10: Can't find include file useful.h
*** Error code 2
make: Fatal error: Command failed for target 'convert'

patience% sccs get useful.h
2.1
13 lines

patience% make convert
cc       -sun3 -o convert convert.c

patience% convert -both 237
convert: binary output not yet implemented
octal   0355
decimal 237
hex     0xed

patience%
```

Retrieve header file

Working on the New Release

Notice what happens when *make* is used to try to compile and link the program. First, it retrieves a version of *convert.c* from *SCCS*. Next, when it tries to compile this version, the C preprocessor can't find the include file *useful.h*.

Here we are relying on *make 's* default rules, rather than using a *makefile.* Therefore, *make* does not know our program includes a header file. If we had created a *makefile* in which *convert.o* depends on *useful.h, make* would have retrieved a read-only copy of the header file automatically.

We retrieve the *header* file by hand with the command:

sccs get useful.h

Now *make* can successfully compile and link the program.

Testing version *2.1* of the program confirms that it still works.

Adding a Feature to the New Release

- Add code to implement multiple base input:

```
patience% sccs edit convert.c
2.1
new delta 2.2
73 lines

patience% vi convert.c

patience% make convert
cc      -sun3 -o convert convert.c

patience% convert -both 0b1101
convert: binary output not yet implemented
octal   015
decimal 13
hex     0xd

patience% sccs delta convert.c
comments? added multiple base input capability
2.2
42 inserted
0 deleted
73 unchanged

patience%
```

Adding a Feature to the New Release

Adding a New Feature

Let's add the ability to accept input in bases other than decimal. This requires editing the program source to add more code.

First, a writable copy is retrieved from *SCCS*, then *vi* is used to make the changes. After testing the changes a new *delta* can be made. The *delta* demonstrates that we just added *forty two* lines of code.

Fixing a Bug on an Old Release

```
patience% sccs edit -b -r1.3 convert.c
1.3
new delta 1.3.1.1
73 lines

patience% vi convert.c
```

Creating Branch

Add code to implement binary output format:

```
patience% make convert
cc       -sun3 convert.c -o convert

patience% convert -both 134
binary  0b10000110
octal   0206
decimal 134
hex     0x86

patience% sccs delta convert.c
comments? implemented binary output
1.3.1.1
32 inserted
6 deleted
67 unchanged

patience%
```

Fixing a Bug on an Old Release

Recall that a *branch* is a line of development independent of the main line. The most common uses of branches are for bug fixes, custom versions, and experimental development.

Using branches to Fix Bugs

To fix the binary output bug in the released version (1.3) we create a branch at *delta 1.3*. The *branch* is created with the command:

sccs edit -b -r1.3 convert.c

Now this file can be edited and a new program created that includes none of the work completed since the release was made.

vi is used to add the code to implement the binary output feature. After testing the program, a *delta* is created, showing that *thirty two* lines were added and *six* deleted to implement the binary output capability.

Merging Changes from the Branch

```
patience% sccs edit -i1.3.1.1 convert.c
Included:
1.3.1.1
2.2
new delta 2.3
141 lines

patience% make convert
cc      -sun3 -o convert convert.c

patience% convert -both 0b10110110
binary  0b10110110
octal   0266
decimal 182
hex     0xb6

patience% convert -both 0x573
binary  0b10101110011
octal   02563
decimal 1395
hex     0x573

patience%
```

Merging Changes from the Branch

Now it's time to merge the changes made on the *1.3.1.1* branch into the main line of development. This is achieved with the command:

sccs edit -i1.3.1.1 convert.c

which creates a source file with the changes merged together.

The *-i* option specifies which *deltas* to include in the text file retrieved for editing.

Testing confirms that the merge was successful.

Note that any conflicts between revisions that are 'included' in this *sccs edit* will not be resolved by *sccs*. At the time the *sccs edit* command is run, error messages will report about unresolved conflicts; by specifying beginning and ending line numbers. It is the responsibility of the programmer to resolve the conflict. All lines (conflicting or otherwise) will be included in the resulting file.

Merging Changes from the Branch

```
patience% sccs delta convert.c
comments? merged in the binary output capability
2.3
0 inserted
0 deleted
141 unchanged

patience% make convert
sccs get -s convert.c -Gconvert.c
cc       -sun3 -o convert convert.c

patience% convert -both 07441
binary  0b111100100001
octal   07441
decimal 3873
hex     0xf21

patience%
```

Merging Changes from the Branch

After testing the program, a new *delta* can be made containing the merged source. *SCCS* indicates that no new changes were made to the file that was checked out when it creates the new *delta*.

A final test of the program verifies that it works.

Using *make* to compile and load the program again demonstrates *make's* ability to extract files from *SCCS*.

Displaying Development History

```
patience% sccs sccsdiff -r1.1 -r2.3 convert.c
3a4,5
>     tobinary — convert number to binary string
>     atoi ——— converts ascii to internal
50a53,55
>     char
>       binary[33];
>
 52, 54c57,58
<       /*  this is complicated so don't do it
<           yet
<       */
---
>       tobinary(number, binary);
>       printf("binary\t0b%s\n", binary);
57c61
<       printf("%o\n", number);
---
>       printf("octal\t0%o\n", number);
59c63
<       printf("%d\n", number);
---
>       printf("decimal\t%d\n", number);
61c65
<       printf("%x\n", number);
---
>       printf("hex\t0x%x\n", number);
```

Displaying Development History

The entire development history of the program is displayed with the command:

sccs sccsdiff -r1.1 -r2.3 convert.c

This extracts the two requested *deltas* and runs the *diff* command to produce a listing of all the changes made during development.

diff shows lines added*(a)* and changed*(c)*. Had lines been deleted from the text file, these would have been displayed, too.

Displaying Development History

```
62a67,132
> }
>
> /*  convert a number to an ascii string of its
>     binary equivalent (assumes ints are 32 bits)
> */
> tobinary(n, s)
>     int n;
>     char *s;
> {
>     int
>       mask = 0x80000000;
>
>     if (n == 0)
>       *s++ = '0';
>     else {
>       while ((n & mask) == 0) mask = (mask >>1) & 0x7fffffff;
>       while (mask) {
>           if (n & mask)
>               *s++ = '1';
>           else
>               *s++ = '0';
>           mask = (mask >> 1) & 0x7fffffff;
>       }
>     }
>     *s = 0;
> }
```

Displaying Development History

Displaying Development History

```
>
> /*  convert an ascii character string to
>      the computer's internal format
>        0b-- binary
>        0 -- octal
>        0t -- decimal
>        0x -- hex
>     no prefix also means base 10
> */
> atoi(s)
>     char *s;
> {
>     int
>       base = 10,          /* default base is 10 */
>       digit = 0,          /* current digit */
>       number = 0;         /* converted number */
>
>     if (*s == '0') {
>       switch (*++s) {
>         when 'b': base = 2;
>         when 't': base = 10;
>         when 'x': base = 16;
>         wlse: base = 8;
>       }
>       if (base != 8) ++s;
>     }
```

Displaying Development History

Displaying Development History

```
>     --s;
>     while (*++s) {
>       digit = 0;
>       if (*s >= '0' && *s <= '9')
>           digit = *s - '0';
>       else if (*s >= 'a' && *s <='f')
>           digit = 10 + *s - 'a';
>       else if (*s >= 'A' && *s <='F')
>           digit = 10 + *s - 'A';
>       else return number;
>       if (digit >= base) return number;
>       number = number*base + digit;
>     }
>     return number;

patience% sccs info
Nothing being edited

patience%
```

Displaying Development History

A final check with *sccs info* is performed to confirm that nothing is being edited.

Project Management Using SCCS

Techniques for coordinating development:

- Establish one *SCCS* database directory.
- Developers work in separate directories.
- Make symbolic links from each working directory to the *SCCS* directory.
- Use *sccs edit* to lock the s–file and retrieve the text file for editing.
- Create *deltas* only after thorough testing.
- Use *make* rather than shell scripts to compile and load programs.

Project Management Using SCCS

One of the challenges of project management arises when two people try to edit the same file. The changes made by one destroy the changes made by the other. *SCCS* helps to manage this problem by permitting only one outstanding *sccs edit* for any given file at any particular moment. This is achieved by locking the *s-file.*

Note: SCCS does not actually prevent programmers from changing the protections on files and then editing them. In this respect, the engineers working on the project must cooperate with each other.

SCCS Locking Mechanism

This locking mechanism can cause problems unless used wisely. For example, if one member of the team locks all the files with *sccs edit*, then no other member of the team can do any work. It is good etiquette to reserve only the files intended for immediate edit.

Another problem arises when one engineer is debugging the changes made to one file. If the rest of the team is using that copy of the file, they are held back until the bugs in it are fixed. The recommended solution is for each team member to work on his/her own copy of the source in his/her own working directory. Each working directory can contain a symbolic link to the the master *SCCS* directory. This allows each team member to use the same *SCCS* database—containing the shared *s-files*—while working with a private copy of the source. After thorough testing, changes can be incorporated back into the *SCCS* database and made available to other team members.

Using make with SCCS

If *make* is used, changes merged back into the *SCCS* database will be automatically used by other members of the team. The command *make* understands that source files depend on the corresponding *s-files* and automatically performs an *sccs get* when a *delta* is made by another engineer.

Appendix A
Special Shell Characters

PRIMARY COMMAND COMMAND LINE COMPONENTS		EXAMPLES
space	Command component delimiter	grep string file
-	Command modification flag	ls -lai
/	Root directory name	cd /
/	Path segment delimiter	cd /usr/games
\	Escape, turn-off a metacharacter	echo *
&	Background execution directive	sort file1 > file2&
( )	Execute enclosed command(s) in subshell	(date ; pwd) >> file

OUTPUT & INPUT REDIRECTION		
|	Pipe	ls | wc -w
cmd | tee filename		
	Send output both to terminal and file	date | tee file.date
|&	Pipe stdout and stderr	cmd |& tr 'C' 'c'
;	Compound command operator	ls ; pwd ; date
>	Redirect output to named file	cal 1986 > temp
>&	Redirect output and errors	cal 1986 >& temp
>!	Redirect output, override noclobber	cal 1986 >! temp
>&!	Redirect output and errors, override	date >&! temp
>>	Append output to named file	date >> temp
>>&	Append output and errors to file	date >>& temp
>>!	Append to file, override noclobber	date >>! temp
>>&!	Append output and errors, override	date >>&! temp
<	Redirect input from named file	mail john < file
<<	Here document, take input from script	ed $file << END_HD
1>&2	Redirect stdout to stderr (Bourne shell)	echo Usage: opps 1>&2

MATCHING & RESTRICTION OF MATCHING		
~	Expands to user's home directory	cp file ~/Dir
.	Current directory	cp Dir/file .
..	Default link name to parent directory	cp file ..
*	0 or more characters in a filename	cp file* Dir
?	Any 1 character in a filename	rm file?
[]	Any 1 character from this set in filename	rm file[a-g]
{}	List of filenames or filename patterns	rm file{acf}
' '	Disable metacharacter expansion (except for \ and !)	echo '*'
" "	Disable metacharacter expansion (except for $, ', \, and !)	echo "*"

CSH HISTORY OPERATORS

!	History operator	!56 newarg
!!	Rerun previous command	!! newarg
!str	Latest command starting with 'str'	!cc
!?st	Latest command containing 'st'	!?file_name
!-3	Third command back in history	!-5
!$	Last argument of previous command	echo !$
^	Past event string substitution	^old^new^
:	Past event argument selection	cc !56:3
:0	Zeroth argument from command	!47:0
:^	Argument one from selected command	cp !83:^ file
:3	Argument three from selected command	rm !14:3
:$	Last argument of selected command	ld !8:$ file.o
:x-y	Range of arguments	cc !38:4-7
:-x	Arguments zero through 'x'	cat !7:-4
:x*	Arguments 'x' through last	mv !5:8* Dir
:h	Head only from path name	cd !24:h
:r	Remove dot extension	make !57:r
:t	Tail portion of path name	od !22:t
:p	Print command only, don't execute	!13:r:p
:s	Substitute operation	!4:s/old/new/

GENERAL SCRIPT COMPONENTS

#	Comment delimiter – to end of line	# script name
#!	Declaration of execution shell	#! /bin/sh
` `	Cmd substitute, immediate execution	echo `date`
(exp)	Expression delimiters	@ v= ($x + $y)

SCRIPT – VARIABLE VALUE REFERENCING

$	Variable access operator		echo $variable
$$	Auto variable: current shell pid no.		echo $$
$#	Auto variable: number of arguments		if ($#path > 3) then
$0	Auto variable: script name		if ($0 == "cmd") then
$1	Auto variable: first argument passed		if ($1 == "-x") then
${v}x	Juxtaposition, isolation of variables		echo ${sh}ell
$*	All arguments passed to script		foreach v ($*)
$?	Is named variable set		if ($?v) then
$<	Take input from standard input	(csh)	set rsp = $<
$@	Same as $* except when delimited by ". All script arguments are treated as one.	(sh)	cat "$@"
$?	Exit status of previous command	(sh)	date ; x=$? ; echo $x
$!	PID of last process run in background	(sh)	date > /dev/null & echo $!
$-	Current flags passed by the host shell –in the called script put the line...	(sh)	sh -xv < script.name echo $-

PARAMETER SUBSTITUTION

${v-word} Value of v if set, else 'word'.	v remains unset.	echo ${v-word}
${v=word} Value of v if set, else 'word'.	v is set to word.	echo ${v=word}
${v+word} 'word' if v is set, else nothing.	v is not changed.	echo ${v+word}
${v?word} Value of v if set, else 'word'.	exit current shell.	echo ${v?word}

BOOLEAN OPERATORS

!	Truth or state inversion operator	if !($# > 3) then
||	Logical "or" operator	if ($a < 4 || $b < 4) then
&&	Logical "and" operator	if ($a < 5 && $b < 5) then
^	Logical "exclusive or" operator	if ($a < 6 ^ $b < 6) then
!(||)	Logical "nor" operator	if ! ($a < 4 || $b < 4) then
!(&&)	Logical "nand" operator	if ! ($a < 4 && $b < 4) then
!(^)	Logical "exclusive nor" operator	if ! ($a < 4 ^ $b < 4) then

MATH OPERATORS

=	Set equal to		set a = 4
@	Set equal to result of expression,	(csh)	@ a = ($a + $b)
++	Increment variable contents by 1,	(csh)	@ a++
--	Decrement variable contents by 1	(csh)	@ a--
+	Addition		@ a = ($a + $b) echo 'expr $a + $b'
-	Subtraction		@ a = ($a - $b) echo 'expr $a - $b'
*	Multiplication		@ a = ($a * $b)
	note: * may need to be escaped in sh script		echo 'expr $a * $b'
/	Division		@ a = ($a / $b) echo 'expr $a / $b'
%	Modulo		@ a = ($a % $b) echo 'expr $a % $b'
<<	Left shift operator (Multiply)		set a=2 ; @ b= ($a << 1) echo $b
>>	Right shift operator (Divide)		set a=2 ; @ b= ($a >> 1) echo $b

COMPARISON OPERATORS

~	Complement operator	@ a = (~ $a)
==	Evaluate for equality	if ($1 == "-v") then
=~	Equality test with metacharacters	if ($v =~ *.c) then
!~	Inequality test with metacharacters	if ($v !~ f.?) then
<	Less than	if ($a < $b) then
>	Greater than	if ($a > $b) then
>=	Greater than or equal to	if ($a >= $b) then
<=	Less than or equal to	if ($a <= $b) then
!=	Not equal to	if ($a != $b) then
!(<)	Not less than	if !($a < 3) then
!(>)	Not greater than	if !($a > 3) then

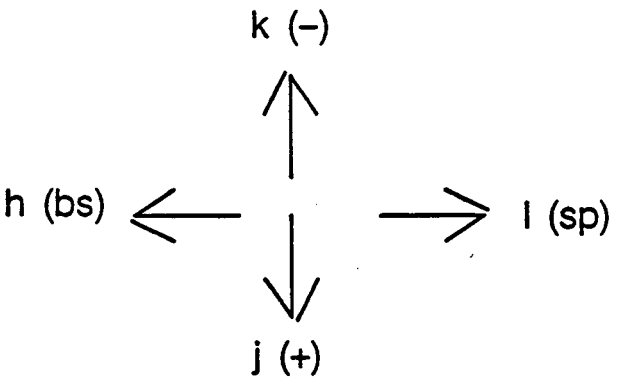
FILE TEST OPERATORS

-b	True if file exists & is a block device	if (-b file) then
-c	True if file exists & is a character device	if (-c file) then
-d	True if file exists & is a directory	if (-d file) then
-f	True if file exists & is a not a directory	if (-f file) then
-g	True if file exists & has setgid set	if (-g file) then
-h	True if file exists & is a symbolic link	if (-h file) then
-k	True if file exists & has sticky bit set	if (-k file) then
-r	True if file exists & is readable	if (-r file) then
-s	True if file exists & is > zero in size	if (-s file) then
-w	True if file exists & is writable	if (-w file) then
-x	True if file exists & is executable	if (-x file) then
-z	True if file exists & is zero is size	if (-z file) then

CONDITIONAL EXECUTION OPERATIONS

cmd && cmd	Execute second cmd only if first succeeds.	date && who date -x && who
cmd || cmd	Execute second cmd only if first fails.	date || who date -x || who

Appendix B
vi Quick Reference



Several ways to move the cursor:
h, j, k and l convenient for 'touch typists'
^H (backspace), + (or <CR>), –, space bar
cursor arrows as diagrammed above

Several ways to move the screen:

^f	move screen forward one page
^b	move screen backward one page
^d	move screen down (forward) one–half page
^u	move screen up (backward) one–half page

Several ways to move on the screen:

H	HOME, top of screen
M	MIDDLE of screen
L	LAST line of screen
G	GOTO last line in file
nG	GOTO *nth* line in file (or :n)
^G	GIVES file status

Several ways to move on the line:

w	forward one word
b	backward one word
e	end of word
0	beginning of line (or ^)
$	end of line

Searching:

/pattern	scan (/) forward for 'pattern' (i.e. /Let me out)
n	next occurrence of 'pattern' (N – previous occurrence)

Getting out:

:q!	"Let me out of here" QUIT with no save
:w	write file without exiting *vi*
:wq	write file and quit *vi*
ZZ	write file and quit *vi*

Insert mode:

Note: Use ESC (escape key) to escape form (end) insert mode.

i	insert text before cursor
I	insert text at beginning of line
a	append text after cursor
A	append text to end of line
o	Open line below current line
O	open line above current line
r	replace one character (does not require ESC)
R	replace current text until an escape is entered
cw	change word (c*n*w changes *n* number of words)
C	change from cursor to end of line
u	"Oops I didn't mean it" undoes last change
U	restores full line

Note: The characters i, a, o, r, c and s (in both upper and lower cases) all invoke insert mode. If insert mode is entered unintentionally, hit ESC and then u to undo.

Yanking (copying):

Y	yanks (copies) line (into a buffer blind to the user)
*n*Y	yanks *n* number of lines
*n*yy	yanks copy of *n* lines into an unnamed buffer

Deleting:

x	deletes characters (also 'd sp')
dw	deletes words
D	deletes from cursor to end of line
dd	deletes lines (into a blind buffer)
*n*dd	deletes *n* number of lines (i.e. *10dd* deletes *10* lines)

Putting:

p	puts the contents of the blind buffer on the next line
P	puts the contents on the line above
xp	transposes two characters

Interactive edit: (search and replace)

/pattern	find pattern to be replaced (as above)
cw	use a replacement command (cw, dw, r, s, etc.)
n	find next occurrence of 'pattern'
.	repeat command

Global replacement:

:1,$s/string1/string2/g — for lines 1 to end of file ($) substitute string2 for string1, e.g. :1,$s/sun/Sun/g

Global delete:

:g/pattern/d — search for lines containing a pattern and delete them
e.g. :g/###/d (to delete lines inserted by *cc file.c |& error-v)*

Reading in files:

:r file2 — read contents of file2 into file1 (at line cursor was on)

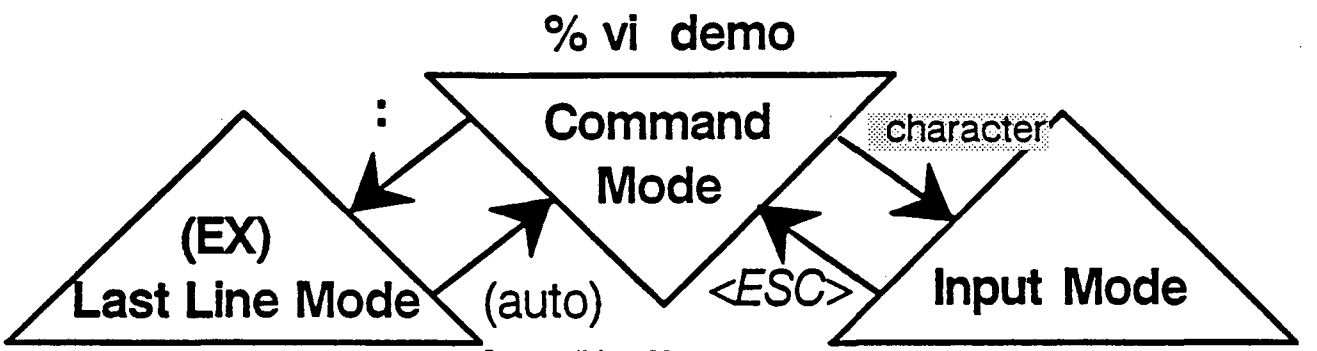
Editing between files: (not needed for Sun system users)

:w	write file1 to save before editing another file
:e file2	edit a second file
:w	write file2 before leaving to go back to file1 (if changed)
:e #	returns user to the original file (file1)

Miscellaneous commands:

:! cmd	execute a shell command from within the editor
~	(tilde or 'wavy'0 changes lower case to upper case and vice versa
%	matches parentheses, braces, brackets
mx	mark location with'x' / d'x execute to location marked 'x'
^V	allows for insertion of control characters (e.g. ^L)
?string	scan (/) backward for 'pattern'
:n,n w file	writes the range of number n,n to a file (e.g. 15,25 w file3)
J	joins the next line to the line the cursor is on
:set ai	editor will automatically insert tabs
:set list	shows special characters (i.e. non–printable characters)
:set nows	stop wraparound search
:set ts=*n*	set tab stops to be other than the default (8)
:set wm=*n*	set wrap margin (automatic carriage return insert at *n)*

example: setenv EXINT 'set ai wm=8 ts=4|map F W|map @ :w^M:e#^M'



Search Functions:

/exp	Go forward to exp
?exp	Go backward to exp

Move and Insert Text:

:3,8d	Delete lines 3–8
:4,9m 12	Move lines 4–9 to 12
:2,5t 13	Copy lines 2–5 to 13
:5,9w file	Write lines 5–9 to file

Save Files and Exit:

:w	Write buffer to disk
:w newfile	Write buffer to newfile
:w! file	Write Absolutely
:wq	Write buffer and quit
:q	Quit editor
:q!	Quit, discarding buffer
:e!	Reedit, discarding changes

Control Edit Session:

:set nu	Display line numbers
:set nonu	Turn off an option
:set all	Show all settings
:set list	display invisible chars
:set wm=5	Wrap lines 5 sp from right margin

Screen/Line Movement:

j↓ k↑ h← l→

0	Goto line start
$	Goto line end
%	Goto matching brace/paren
G	Goto last file line
3G	Goto line 3

Word Movement:

W	Go forward 1 word
3W	Go forward 3 words
B	Go back 1 word
3B	Go back 3 words

Search Functions:

n	Repeat previous search
N	Reverse previous search

Delete Text:

x	Delete one character
dw	Delete one word
dd	Delete one line
D	Delete to end of line
d0	Delete to begin of line
dG	Delete to end of file
4dd	Delete 4 lines

Cancel Edit Function:

u	Undo last change
.	Do last change again

Copy and Insert Text:

Y	Yank a copy of a line
5Y	Yank a copy of 5 lines
p	Put below cursor
P	Put above cursor

Word processing functions:

J	Join next line to current
4J	Join 4 lines to current
xp	Transposes 2 characters

Add/Append Text:

a	Append after cursor
A	Append at line end
i	Insert before cursor
5i	Insert text 5 times
I	Insert at beginning of line

Add new lines:

o	Open a line below cursor
O	Open a line above cursor

Change Text:

cw	Change a word
3cw	Change 3 words
C	Change line
r	Replace one character
R	Replace/Type-over a line

Appendix C
install

Introduction

Techniques used to install Software under SunOS:

- *install*
- *shell archives*
- *make and makefiles*

Introduction

The purpose of this appendix is to provide programmers with the knowlege necessary to install various types of software under SunOS with the *install* command. The two other methods, utilizing shell archives and/or makefiles, were presented in earlier modules.

install

An installation is the procedure whereby scripts, programs, data and other files are made accessible to users on the network. For example, the editor *vi* is installed so that it is available to all users. The process of installation ranges from simply copying a program to a standard location to performing many complex tasks including compiles, links, and data preparation.

Shell Archives

Shell archives are shell scripts that can be delivered by electronic mail systems and automate the installation of software. Each is composed of a combination of shell commands and embedded data that, when executed, takes the data residing in the file and places it into external files (self–extracting).

Make and Makefiles

make is a program that is often used to build and install systems. *make* reads highly–structured scripts named *makefiles* for instructions on how to perform its tasks. It applies default rules and rules contained in a *makefile* to perform an installation. *make* is good for programs which are compiled from several modules, libraries, etc.

Fundamentals of an Install

To *install* a program:

- The executable file must be copied *(cp)* to the target directory.
- Execute permission *(chmod)* must be provided to *other* network users.
- The command *install* performs both a *cp* and a *chmod.*

- Basic Command Format:

  install [–options] sourcefile destination

Fundamentals of an Install

Programs, (e.g. *games*) accessed by everyone on the network, are by convention often installed in the */usr/local* directory. This directory typically contains public domain software as well as application programs which are developed in-house.

Because *install* does not use special privileges to copy files from the destination to the source, you must:

- have permission to read the files to be installed.
- have permission to copy into the destination file or directory.
- have permission to change the modes on the final copy of file if you want to use the *-m* option to change modes.
- be superuser if you want to use the *-o* option to change ownership.

To *install* a program in this directory(or any other designated directory), two important steps are necessary:

(1) The executable file must be copied *(cp)* to the target directory.

(2) Execute permission must be provided to *other* network users.

The shell command *install* performs both steps, *cp* followed by *chmod (755 by default)* in a single command. For example, the command:

install game /usr/local/bin

*install*s the program *game* in the directory */usr/local/bin*. There are options to *install,* which are discussed in the section on *Installing Software with Special Privileges.*

Fundamentals of an Install

```
patience% ls -l
-rwx------    1 patience    16 Jun 17  09:05 why
patience% cat why
#! /bin/csh -f
echo Because
patience% why
Because
patience% echo $path
. /usr/local/bin /usr/ucb /bin /usr/bin

patience% install why /usr/local/bin

patience% ls -l /usr/local/bin/why
-rwxr-xr-x  1 patience   16  Jun 17 17:05 /usr/local/bin/why

patience% rm why

patience% why
why: Command not found.
patience% rehash

patience% why
Because
```

Fundamentals of an Install

This example presents the fundamental steps necessary to *install* files under SunOS in the directory */usr/local/bin*. It assumes that we have read access to *why* and write access to */usr/local/bin*.

We first verify that the C shell search path includes */usr/local/bin* to allow access to the *why* command. Executing *printenv* or *echo $path* confirms that */usr/local/bin* is in the search path for *csh.*

Next the *install* command copies *why* to the */usr/local/bin* directory and makes it executable by everyone.

install effectively performs the following steps:

cp why /usr/local/bin/why
chmod 755 /usr/local/bin/why

The *ls –l* command displays the results of the *install.*

Lastly, we attempt to execute the command *why.* This results in the error message *"why: Command not found."* This is because the *csh* builds a *hash* table of all commands specified in the path variable upon startup so that commands may be located quickly. Issuing the *csh* command *rehash* causes this table to be rebuilt. Other people already logged in must issue the *rehash* command or log out and back on in order to use the new *why* command.

Installing Software with Special Privileges

- *install* Extended Command Format:

*install [–***m** *mode] [–***o** *owner] [–***g** *group] binary destination*

Useful Options:

–m	*mode* (*755* by default)
–o	*owner* (defaults to current user id)
–g	*group* (defaults to current group id)

Installing Software with Special Privileges

It is often useful to write shell scripts and C programs which have special privileges. For example, only one user can access the tape drive at a time when running the *tar* command, otherwise the data written can become corrupted. In this case, the *root user* normally grants ownership of the device to a person using the tape unit. Therefore, a user named "joy" is granted exclusive access to *rst0* when the *root user* issues the following shell commands:

```
# chmod 600 /dev/rst0
# chown joy /dev/rst0
```

This could also be accomplished by a program or shell script that has superuser privileges.

A program can change the ownership of device *rst0* when the unit is not being used by anyone else. This program must execute in *Set User ID on Execution* (*setuid*) mode and be owned by root to allow nonprivileged users to change the ownership of */dev/rst0*. For example, the following commands allow the program named *grab* to allocate the tape unit:

```
# chown root grab
# chmod 4755 grab
```

This program has the privileges of root regardless of who runs it, because it is owned by root and executes in *setuid* mode.

The *install* command can be used to move *grab* to */usr/local/bin* and perform the above steps by issuing the command:

```
# install -m 4755 grab /usr/local/bin
```

A *setuid* Shell Script

Login: *payor*
Password:
patience% **ls –l personnel**
–rw————— 1 payor 133 Jan 21 16:16 personnel

patience% **cat personnel**

Employee	Phone	Salary
Adams, P	x1332	$56,550
Jones, E	x6523	$43,523
Smith, A	x4612	$64,130
Smith, J	x5621	$45,312
Young, B	x8763	$71,220

patience% **cat phone**

```
#!/bin/csh –feb
if ($#argv !=1) then
        /bin/echo "Usage: phone name"
        /bin/echo "e.g.: phone smith"
        exit(1)
endif
/bin/grep –i "^$1" ~payor/personnel | awk '{print $1, $3}'
```

A *setuid* Shell Script

This example illustrates how to write a C shell script that executes in *setuid* mode.

The *ls –l* displays the mode of a file named personnel. The contents of the file are accessible by the owner, *payor*. Other users cannot view the contents of the file because the *group* and *other* modes do not grant read privileges on this file.

A *cat* of the file *personnel* displays the employee name, phone extension and salary information. The shell script *phone* provides limited access to the file *personnel*. It accepts a *name* as an argument, and prints out all employees with that name and their phone extensions.

Line 1

It is necessary for the first line of the script to contain *–b* before the C shell script executes in *setuid* mode. Without the *–b* option, the shell will not run a set–user–ID script. The *–f* option causes the shell to start more quickly by skipping *.cshrc*. The *–e* option terminates the script if an error occurs for security reasons.

Line 7

The */bin/grep* command selects employees of interest and the *awk* command passes the first two fields to *stdout*.

A *setuid* Shell Script

```
patience% install -m 4711 phone /usr/local/bin
```

```
patience% ls -l /usr/local/bin/phone
-rws--x--x 1 payor  133 Jan 21 16:41 phone
```

A *setuid* Shell Script

The *install* command copies the script to */usr/local/bin* so it may be executed by other users. The mode of the script is set using the *–m 4711* option. This mode allows any user to access the personnel file through the *phone* command. This is accomplished by setting the mode to execute in *setuid* or *Set User ID on Execution* mode. Note that you only have permission to do this if you are the owner of the file or are superuser.

A program or script that executes in *setuid* mode assumes the privileges of the owner of the script for the duration of execution. In this example, the user *payor* owns both the *personnel* and the *phone* script. Anyone executing the *phone* script assumes the privileges of *payor* during execution of the script and may therefore access the *personnel* file.

The *ls –l /usr/local/bin/phone* command displays the modes as set by the *–m 4711* option. The *rws* indicates the script is readable and writable by *payor*, and the script runs in *setuid* mode. The two *—x's* allow execution by other users.

Note: Other users cannot view the contents of the *phone* script as they are not given read privileges.

A *setuid* Shell Script

```
patience% login anyone
Password:

patience% phone Smith
Smith, A        x4612
Smith, J        x5621

patience% cat ~payor/personnel
/usr/payor/personnel: Permission denied

patience% cat  /usr/local/bin/phone
/usr/local/bin/phone: Permission denied
```

A *setuid* Shell Script

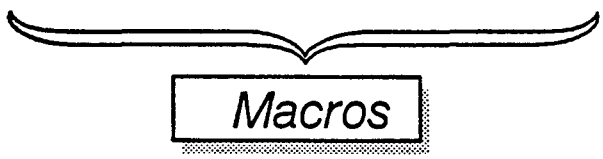
The script is tested by logging in as another user and issuing the command: *phone smith*.

Notice that security is also tested. An attempt to access either the personnel file or the phone script, produces a permission denied error message.

Note that programs written in compiled languages (e.g. C) may also execute in *setuid* mode.

Case Study: A *makefile* Install

```
patience% cat makefile
SOURCE=  fortune.c strfile.h strfile.c rnd.c \
        unstr.c scene
TFILES=  Troff.mac Troff.sed Do_troff
LIBDIR=  /usr/games/lib
BINDIR=  /usr/games
OW=      arnold
GRP=     arpa
DEFS=
CFLAGS=  -O $(DEFS)
SFLAGS=  -r
TDEV=    -Pver
DESTDIR=
```



Case Study: A *makefile* Install

Case Study: A *makefile* Install

○ *makefile* continued . . .


```
all:    fortune strfile unstr fortunes.dat

fortune: fortune.o rnd.o
        $(CC) $(CFLAGS) -o fortune fortune.o rnd.o

strfile:    strfile.o rnd.o
        $(CC) $(CFLAGS) -o strfile strfile.o rnd.o

unstr:    unstr.o
        $(CC) $(CFLAGS) -o unstr unstr.o

fortune.o strfile.o unstr.o: strfile.h

fortunes.dat: fortunes strfile
        ./strfile $(SFLAGS) fortunes

fortunes: scene
        cp scene fortunes
```

rules

Case Study: A *makefile* Install

Target: all

In this example, the *target "all"* depends on *fortune, strfile, unstr* and *fortunes.dat*. Examination of the *makefile* reveals that these four *dependencies* are also *targets*. A command is not placed after the *"all" target*, as all necessary processing is performed during evaluation of the four dependencies and their individual *targets*.

Target: fortune

The next *target* entry describes the dependencies of the *fortune* program itself. Once the dependencies have been satisfied, *fortune* is created with the command:

$(CC) $(CFLAGS) –o fortune fortune.o rnd.o

$(CC) is one of *make*'s default macros. It is defined as *cc* to use the C compiler by default. The *$(CFLAGS)* macro is defined in the macro definition section of the *makefile*. Combining them yields the command:

cc –O –o fortune fortune.o rnd.o

which is executed.

Hierarchical Relationship of Dependencies

Recall the dependencies must exist before *fortune* is created. The first dependency, *fortune.o*, is found to be a *target* a few lines down. *fortune.o* is created by that entry before the second dependency *rnd.o* is evaluated.

rnd.o must be created to satisfy *fortune's* dependency upon it. However, *rnd.o* does not appear as a target in the makefile. Since there are no rules to describe how it should be created, *make* creates *rnd.o* by compiling *rnd.c* according to default rules.

Case Study: A *makefile* Install

○ *makefile* continued . . .

```
lint:
    lint -hxb $(DEFS) fortune.c rnd.c \
        > fortune.lint 2>&1
    lint -hxb $(DEFS) strfile.c rnd.c \
        > strfile.lint 2>&1
    lint -hxb $(DEFS) unstr.c \
        > unstr.lint 2>&1
install: all install.data
    install -s -m 4711 -o daemon fortune \
        $(DESTDIR)$(BINDIR)
install.data: fortunes.dat
    install -m 600 -o daemon fortunes.dat \
        $(DESTDIR)$(LIBDIR)
clean: sort.clean
    rm -f fortune fortunes fortunes.dat \
        strfile unstr ? core *.o
sort: strfile unstr
    strfile -oi scene
    mv scene Oscene
    unstr -o scene
sort.clean:
    rm -f Oscene
```

Case Study: A *makefile* Install

Target: lint

This target executes the C program verifier. *lint* is not a dependency in any target entry. Issue the command *make lint* to execute the three command lines. These commands execute unconditionally since there are no dependencies in this target. Commands must be written using Bourne shell syntax.

Target: install

This target is executed by issuing the command: *make install.* Its first dependency, *all,* performs all processing necessary to *install* the software. The command line expands into:

install -s -m 4711 -o daemon fortune /usr/games

This command copies the *fortune* executable into the directory */usr/games.* The *-s* option strips the symbol table from the executable to save disk space. The *-m 4711* enables *setuid* mode, and ownership is given to *daemon.* (Note that the person installing *fortune* must be the *root user* for the ownership to be properly set. [See the *chown(8) man* page.])

Target: install.data

This target copies the file containing the fortunes, *fortunes.dat*, into the directory */usr/games/lib. Daemon* owns the file and the mode is set to read/write by owner. Any user can access the */usr/games/lib/fortunes.dat* file using the *fortune* command since it executes in *setuid* mode.

Targets: clean and sort

The *clean* target entry is used to delete intermediate files such as the object files once *fortune* has been properly installed. The *sort* target uses the programs *strfile* and *unstr* to sort the fortunes in alphabetical order saving the original file as *Oscene.*

Case Study: A *makefile* Install

```
patience% make
cc –O –c fortune.c
cc –O –c rnd.c
cc –O –o fortune fortune.o rnd.o
cc –O –c strfile.c
cc –O –o strfile strfile.o rnd.o
cc –O –c unstr.c
cc –O –o unstr unstr.o
cp scene fortunes
./strfile –r fortunes
"fortunes" converted to "fortunes.dat"
There were 1181 strings
Longest string: 642 bytes
Shortest string: 11 bytes
```

Case Study: A *makefile* Install

Building the fortune System

make builds the *fortune* system by executing these steps:

(1) First, *make* processes the *all* entry. The first dependency, *fortune*, is evaluated to determine if it has been satisfied. This dependency is found to be a target in the following entry and is processed next. The first dependency in the *fortune* entry is found to be *fortune.o*, which is also a target.

(2) The target entry *fortune.o* has one dependency, *strfile.h*. *make* searches the current directory and finds that the file *strfile.h* exists.

(3) *make* now creates the *fortune.o* target and searches the current directory for the file *fortune.c*. *make* then issues the command *cc –O –c fortune.c* to make *fortune.o*.

(4) make processes the second dependency, *rnd.o*. Since *rnd.o* is not a target in the *makefile*, *make* checks the current directory to see if a file named *rnd.o* exists. *make* creates this file by issuing the command *cc –O –c rnd.c*.

(5) make creates the target *fortune* itself. After macro expansion, *make* issues the command *cc –O –o fortune fortune.o rnd.o*. (*Note:* the command was specified since the implicit rules can only handle compiles containing one object file.)

(6) *make* returns to the *all* target entry to process the second dependency, *strfile*. Processing continues in a similar fashion to *make* the *strfile*, the *unstr*, and the *fortunes.dat* targets.

Finally, all dependencies are satisfied.

Case Study: A *makefile* Install

patience% **vi scene**
add new fortunes

patience% **make**
cp scene fortunes
./strfile –r fortunes
"fortunes" converted to "fortunes.dat"
There were 1192 strings
Longest string: 642 bytes
Shortest string: 11 bytes

Case Study: A *makefile* Install

Next the editor *vi* is used to add new *fortunes* to the file *scene*. *make* determines that the *fortunes.dat* dependency in the *all* target needs to be updated. A new *fortunes.dat* file is created.

Appendix D
Source Code

buggy.h

```
#define U_MSG "usage:\n\
\t b\t array boundary error\n\
\t c\t structure element clobber error\n\r\
\t d\t dangling pointer error\n\r\
\t f\t for null statement error\n\r\
\t l\t loop forever error\n\r\
\t m\t macro expansion error\n\r\
\t s\t stack overflow error\n\r"

#define TEST_STRING "Watson, come here..."
#define DEFAULT_COUNT 10
#define STRING_SIZE 3
#define ucase(c) isalpha(c)?(isupper(c)?c:toupper(c)):c
```

buggy.c

```
#include <stdio.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/times.h>

#include "buggy.h"

void
bounds(),clobber(),dangler(),for_null(),
loop_forever(),macro_gotcha(),stack_blow(),
no_op(),print_usage(),start_timer(),stop_timer();

char *dpointer();
char test_string[20];
int debug = 1;

main (argc, argv)
int argc;
char **argv;

{
    if (argc < 2)
    {
       print_usage();
       exit(1);
    }

   while (argc-- > 1)
   {
       switch (*argv[argc])
       {
           case 'b':  bounds();       break;
           case 'c':  clobber();      break;
           case 'd':  dangler();      break;
```

```
            case 'f':  for_null();      break;
            case 'l':  loop_forever(); break;
            case 'm':  macro_gotcha(); break;
            case 's':  stack_blow();   break;
            default :  print_usage();  break;
        }
    }
}

void
bounds()
{
    char instring[26];
    char outstring[26];
    int i;
    if (debug)
        printf ("in bounds\n\r");
    strcpy (instring, "abcdefghijklmnopqrstuvwxyz");
    for (i=1; i<26; i++)
        outstring[26-i] = instring[i];
    printf (outstring);
}

void
clobber()
{
    int i;
    char *ptr,
         *str = "Ubung macht den Meister";

    for (i = 0; i < 23; i++)
        *(ptr++) = str[i];
    if (debug)
        printf ("in clobber\n\r");
    printf (ptr);
}
```

```
void
dangler()
{
    char *char_ptr;
    if (debug)
        printf ("in dangler\n\r");
    char_ptr = dpointer();
#ifdef DEBUG
    no_op();
#endif
    printf ("%s", char_ptr);
}

void
for_null()
{
    int loop_count, i;
    if (debug)
        printf ("in for_null\n\r");
    for (i=0; i<20; i++);
    {
        loop_count++;
        printf ("%d", loop_count);
    }
}

void
loop_forever()
{
    int icount = 0, default_count = DEFAULT_COUNT;
    char instring[STRING_SIZE];
    long your_id;
    printf ("Enter number of loops: ");
    icount = atoi (gets (instring));
    start_timer();
    if (icount <= default_count)
```

```
        icount = default_count * 100;
    else
        icount = icount * 100;
    while (icount >= 0);
    {
        your_id = gethostid();
        icount--;
    }
    stop_timer();
}

void
macro_gotcha()
{
    int i;
    strcpy (test_string, TEST_STRING);
    if (debug)
        printf ("in macro_gotcha\n\r");
    while (i < strlen (test_string))
        putc (ucase (test_string[i++]), stdout);
}

void
stack_blow()
{
    char instring[80];
    int i=0;
    if (debug)
        printf ("in stack_blow\n\r");
    printf ("Enter an integer to be factored:");
    gets (instring);
    i = atoi (instring);
    printf ("%d", factor(i));
}

void
print_usage()
```

```
{
    printf(U_MSG);
}

/*************************************
*    dangler routines
*************************************/

char *
dpointer()
{
    char a[80];
    strcpy(a, "Now you see it.\n\r");
    return a;
}

void
no_op()
{
    char b[80];
    strcpy (b, "Now you don't.\n\r");
}

/*****************************************
*        stack_blow routine
*****************************************/

factor(i)
int i;
{
    int next = 0;
    next = i - 1;
    if (next = 0)
        return 1;
    else
        return (i* factor (next));
}
```

```
/*****************************************
*     timing routines
*****************************************/

static struct tms start_time;
static long since_start;

void
start_timer()
{
    since_start = times (&start_time);
}

void
stop_timer()
{
    struct tms stop_time;
    static long since_stop;
    since_stop = times (&stop_time);

    printf ("total ticks :%d\n\r\
    process ticks:%d\n\r\
    system ticks:%d\n\r",
        (stop_time.tms_utime - start_time.tms_utime)+
        (stop_time.tms_stime - start_time.tms_stime),
        (stop_time.tms_utime - start_time.tms_utime),
        (stop_time.tms_stime - start_time.tms_stime));
}
```

stations.c

```
main()
{
   static int wstation_count;
   wstation_count = 99;
   while (wstation_count > 1)
   {
      fstations_(&wstation_count);
      printf("\n\n");
      pstations(wstation_count);
      wstation_count--;
   }
   printf("And the backup is gone!\n\r");
}
```

fstations.f

```
      function fstations (icount)
      integer icount
      write (*, 100) icount
100   format (//, i3, ' shiny new workstations left!')
      write (*, 200) icount
200   format (i3, ' shiny new workstations!;')
      icount = icount - 1
      write (*, 300)
300   format ('you pack one up, and ship it out!')
      write (*, 400) icount
400   format ('There''s', i3, ' shiny new workstations left!')
      return
      end
```

pstations.p

```
procedure pstations(Count:integer);
begin
   writeln(Count:3, ' shiny new workstations left!');
   writeln(Count:3, ' shiny new workstations!');
   writeln('you pack one up, and ship it out.');
   Count:=Count-1;
   writeln('There''s', Count:3, ' shiny new workstations left!');
end;
```