Solutions To Exercises

Chapter 1

1.1 The operating system (OS) allocates the CPU, memory and registers to be used by programs in execution. It also offers a set of programs as additional services for performing routine functions like file copying, directory removal and so forth.

1.2 It makes a call to the operating system to do the job on its behalf.

1.3 No, when a program needs to perform an operation that doesn't need the services of the CPU (like doing a disk I/O operation), the process representing the program vacates the CPU while the operation is in progress.

1.4 Multiprogramming means that multiple programs can be concurrently present in memory. Multiuser operation implies that these multiple programs can be run by multiple users. Multitasking means that a single user can also run multiple programs.

1.5 UNIX design allows commands to be interconnected so one command takes input from another. This is possible only if the commands are noninteractive. Header information of one command has no meaning for another, the reason why most UNIX commands don't produce headers.

1.6 System calls are routines built into the kernel that handle all calls made to the OS. They are available as functions that can be invoked by C programs designed to run on a UNIX system.

1.7 False, they must use the system calls specified by POSIX.

1.8 *[Ctrl-d]*, **`logout`** and **`exit`**. The last one will always work.

1.9 `README` and `readme` exist as separate files; UNIX filenames are case-sensitive.

1.10 (i) **`tty`** shows the filename of your terminal which also shows up against your username in the **`who`** output. (ii) Clears the screen. (iii) **`id`** shows the username and the group the user belongs to. (iv) **`echo $$`** shows the PID of your shell which also appears in the **`ps`** output.

1.11 The shell. Most systems offer the Bourne, C, Korn and Bash shells.

1.12 The user-id is associated with all files and processes that you create. When you create a file, your user-id is its owner. When you run a program, your user-id is the owner of the process associated with the program. When you send a mail to someone, the mail header shows your user-id as the sender.

1.13 The AT&T and Berkeley schools. Two major standards bodies, POSIX and The Open Group, have developed the Single UNIX Specification that serves as a single reference for all developers.

1.14 The **`cd`** command returns you to the directory where you were placed on login irrespective of where you are currently.

1.15 Because the shell scans the command for special characters and recreates a simplified command structure that is understood by the kernel.

1.16 They are all represented by files.

Chapter 2

2.1 The command is not executed. The shell ignores all text following the `#`, so the `#` could serve as a comment character in shell scripts.

2.2 (i) UNIX commands are generally not interactive; Windows programs are. (ii) UNIX commands don't need to have any specific extensions in their filenames; Windows must have `.EXE` or `.COM`. (iii) UNIX is sensitive to case; Windows is not.

2.3 The current directory doesn't exist in `PATH`.

2.4 (i) By changing the value of `PATH` to include that directory. (ii) By using a pathname of the command.

2.5 In `/sbin` and `/usr/sbin`.

2.6 **`cd`** is an internal command of the shell.

2.7 The one in `/bin` is an external command all right, but it's the shell builtin that is normally executed.

2.8 An option is also an argument. No arguments, the >, < and the words following them are interpreted by the shell *before* the command is executed.

2.9 Because the command considers it as one of its options and often generates an error.

2.10 `tar -tvf /dev/fd0`

2.11 The arguments, `20` and `foo.tar`, should be reversed because the `-f` option needs a filename, and the `-b` option needs a numeric argument. The command will also work without the hyphen, though POSIX discourages such use.

2.12 In the first example, **`cat`** tries to open the file. In the second example, the Bash shell tries to do the same.

2.13 The shell compresses multiple spaces to a single space unless they are quoted.

2.14 The > is normally the secondary prompt and appears when an *[Enter]* is pressed without closing a quote.

2.15 **`tail`** will work either with `-f` or `-r` (but not both), and **`ls`** can handle multiple filenames.

2.16 `man` *name*, `man -s 5` *name*, and `man -s 2` *name*

2.17 Use **`man -k`**.

2.18 (i) Use the `-e` option. (ii) Use **`echo -n "Filename: "`**.

2.19 An escape sequence is a string preceded by a \ that assigns a special meaning to the string. `\c` places the cursor in the current line, `\n` prints a newline and `\t` prints a tab.

2.20 `printf "%o\n" 255` and `printf "%x\n" 255`

2.21 **`script`** runs another shell process.

2.22 **`mailx`** can be used in a shell script in a noninteractive manner. Also, the components of the message need not be known in advance but can be obtained from shell variables or from the output of another program.

2.23 Yes, a user can log in multiple times.

2.24 Use **`uname -n`**.

2.25 Can't do that.

2.26 `date +"%d/%m/%Y"`

2.27 Before accepting input, use **`stty -echo`**. To revert to the previous setting, use **`stty echo`**.

2.28 The kernel maintains a buffer that stores all keyboard input.

2.29 Use **`stty intr \^c`**. The setting is not permanent unless it is placed in a startup script.

Chapter 3

3.1 A directory contains the inode number and filename for every file in its fold. When a file is created or copied, an entry is added to the directory. When the file is deleted, its corresponding directory entry is removed. When a file is renamed, the existing directory entry is updated with the new name. The size of a directory is small because it is related to the number of files and not to their contents.

3.2 A device file is simply an interface to the actual device. When a command accesses a device file, the kernel identifies the actual device from the file's attributes, and then uses these attributes to operate the device.

3.3 (i) Won't work without the `-p` option. (ii) Will work. (iii) Will work only if `c` is empty. (iv) Can't remove `a` when `a/b` exists, so the command will remove only `a/b`. (v) Won't work since `/bin` is not writable by nonprivileged users.

3.4 (i) The directory `c_progs` is not empty. (ii) `c_progs` is an ordinary file. (iii) You don't have write permission in the current directory.

3.5 Can't create (i) and (ii) but can create (iii). Every directory has `.` and `..` as invisible directories and you can't create or remove them.

3.6 The directory `bar` contains hidden files which can only be displayed by **`ls -a bar`**.

3.7 (i) `bar1` becomes a subdirectory of `bar2` (ii) `bar1` is renamed to `bar2`.

3.8 **`cd ~charlie`** changes the current directory to charlie's home directory, but **`cd ~/charlie`** switches to the subdirectory `charlie` in the user's home directory. Both commands can work if there is a user-id named charlie and the user running the commands also has the subdirectory `charlie` in her home directory.

3.9 By using `$HOME/html` in the first place. This would have allowed charlie to change his home directory any time without changing any of his scripts.

3.10 When the current directory doesn't feature in `PATH`, or there's another program of the same name whose directory occurs earlier than the current directory in `PATH`.

3.11 Numbers, uppercase letters and lowercase letters.

3.12 (i) `bar` is an ordinary file. (ii) `bar` is a directory containing a single file named `bar`.

3.13 (i) Switches to the `/` directory. (ii) You can't create a directory in `/home` if you are not the superuser. (iii) You can't remove your parent directory unless you position yourself above it. (iv) Displays all filenames of the parent directory.

3.14 All commands switch to the home directory.

3.15 (i) `backup` is not writable. (ii) `hosts` is not readable. (iii) `hosts.bak` is not writable.

3.16 Make use of a temporary directory named `z` and issue these commands from the home directory:

```
cd a ; mv a ../z ; cd .. ; rmdir a ; mv z a
```

3.17 (i) Removes from the current directory all files except directories and hidden files. (ii) Same as (i) except that the task is performed interactively. (iii) Same as (i) except that all directory structures are also forcibly removed.

3.18 (i) Moves the `include` directory to the current directory. (ii) Recursively copies the directory `bar1` to `bar2`. (iii) Moves all files in the current directory (except the hidden files) to the `bin` directory of the parent.

3.19 (i) Yes (ii) No, must use **`cp -r foo bar`**.

3.20 The repeat factor is an integer prefixed to a command to repeat the command as many times as the value of the integer. For instance, **`10f`** scrolls by 10 pages. The command **`/include`** searches for `include`, and **`n`** repeats the search. The dot command repeats a navigation command like **`10f`**, but **`n`** repeats a search command.

3.21 There are two file types: block special and character special.

3.22 DOS files use CR-LF as the line terminating characters. UNIX files use only LF. **`dos2unix`** and **`unix2dos`** perform conversions from one format to another.

3.23 All lines are terminated by *[Ctrl-m]*.

3.24 The contents of the file are displayed on the terminal.

3.25 Use **`tar -cvf archive.tar foo.html bar.html ; gzip archive.tar`**. To reverse the process, use **`gunzip archive.tar.gz ; tar -xvf archive.tar`**.

3.26 **`zip`** (i) can compress multiple files into a single archive. (ii) can archive a complete directory structure. (iii) combines the functions of **`tar`** and **`gzip`** in a single command.

3.27 Assuming that a directory structure named `bar` has to be mailed, use (i) `tar -cvf dir.tar bar` (ii) `zip -r dir.zip bar`. The recipient has to use (i) `tar -xvf dir.tar` (ii) `unzip dir.zip`. **`zip`** is superior because it compresses the directory structure also unless the **`tar`** archive is also compressed with **`gzip`**.

3.28 A command is said to behave recursively when it descends a directory structure to examine all files found in the tree-walk. The following commands behave recursively: (i) `rm -r *` (ii) `cp -r bar1 bar2` (iii) `zip -r bar.zip bar` (iv) `tar cvf bar.tar bar`

Chapter 4

4.1 2048 bytes on a system where the disk block size is 1024 bytes.

4.2 By default, yes. But ownership patterns can be changed with separate commands.

4.3 Displays the listing of the (i) current directory (ii) all files in the parent directory.

4.4 We need to assume a default value for the permissions for using relative assignment. Assuming that the default permissions are `644`, use (i) `chmod u+x,go-r foo` (ii) `chmod 700 foo`.

4.5 Assuming that the directory has write permission, (i) the file can't be read, written or executed but can be removed. (ii) you can do anything with this file including removing it.

4.6 romeo is both a user-id and a group-id. User romeo (i) can edit this file only if he also belongs to the romeo group. (ii) can delete the file if he has write permission in the current directory. (iii) cannot change permissions since he is not the owner. (iv) cannot change ownership since he is not the superuser.

4.7 Using relative permissions (i) `u+x,g+w,o+wx` (ii) `u-w,g-x,o-r` (iii) `u-rw,g-x` (iv) `u-rw,g-rx,o-r`
Using absolute permissions (i) `777` (ii) `440` (iii) `044` (iv) `000`

4.8 The command removes the write permission for the current directory for user, so you can't create or remove files in it. `-w` is not the same as `a-w`, an undocumented feature of **`chmod`**.

4.9 `foo` exists at the destination but is write-protected. Remove or rename this file before copying.

4.10 Remove the read permission from the directory for others.

4.11 `bar` doesn't have execute permission.

4.12 File deletion is controlled by the directory permissions and is in no way determined by the file's permissions. The file can be deleted only if the directory has write and execute permission.

4.13 (i) No, because the owner's permissions override the group's. (ii) Yes, if the directory is writable by the owner.

4.14 (i) All files are editable and all directories writable by all categories of users, which implies the total absence of security. (ii) All files are editable and directories writable by user and group members. The user may want this arrangement only if group members need to edit the same file.

4.15 True, the UNIX file system generally comprises multiple file systems, each having its own root directory. These file systems combine to appear as a single file system at the time of booting.

4.16 The link count in the inode is incremented by one. An entry for the new filename is made in the directory with the same inode number as the filename linked to.

4.17 They are linked filenames.

4.18 (i) `bar` is linked to `foo`. (ii) Command fails. (iii) Link for `foo` is created in the `bar` directory with the same name (i.e. `foo`).

4.19 Use **`ls -i`** and see whether they have the same inode number. It's quite possible for two files in two separate file systems to have the same inode number, so a comparison with **`cmp`** must also return success.

4.20 (i) Hard links help prevent accidental file deletion. (ii) If a program expects to find a file at location A, and the file is actually at location B, then placing a link for the file at location A makes everything work without problems. Hard links can't be used to link directory names and can't link across file systems.

4.21 Make a symbolic link between the two directories.

4.22 Fast symbolic links are used on Linux systems where the pathname is stored in the inode itself and not in the file. Dangling symbolic links point to nonexistent files.

4.23 (i) From the directory. (ii) The numeric UID from the inode and its name representation from `/etc/passwd`. (iii) The numeric GID from the inode and its name representation from `/etc/group`.

4.24 Create a file `foo` and then use **`chown`** to transfer ownership to another user. Next use **`chgrp`** to surrender group ownership to another group of which you are not a member. If both commands succeed, then you are not using the BSD versions of these commands.

4.25 False, the owner can change only the permissions but not ownership. The owner itself can be changed only by the superuser.

4.26 The time of last modification, access and change of inode. **`ls -l foo`** displays the last modification time and **`ls -lu`** displays the last access time.

4.27 Find out its last access time with **`ls -lu foo`**.

4.28 (i) **`ls -l`** sorts listing in ASCII collating sequence, but **`ls -lt`** sorts by modification time (ii) **`ls -lu`** shows last access time but sorts by ASCII sequence, but **`ls -lut`** also sorts by access time.

4.29 (i) `find $HOME -name "*.[hH][tT][mM][lL]" -print` (ii) `find $HOME -inum 9076 -print`
(iii) `find $HOME -type d -perm 666` (iv) `find $HOME -mtime +1 -mtime -2 -print`
These commands can produce an incomplete list if **`find`** encounters a directory that doesn't have execute permission. **`find`** in (iii) can't make a recursive examination of a directory that has `666` as the permissions.
4.30 (i) `find / -mtime -1 type f -exec mv {} ../posix \;`
(ii) `find $HOME \( -name a.out -o -name core \) -ok rm {} \;`
(iii) `find /oracle -type f -name login.sql -exec cp {} . \;`
(iv) `find . \( -type f -exec chmod 644 {} \; -o -type d -exec chmod 755 {} \; \)`

Chapter 5

5.1 Input Mode, Command Mode and ex Mode. *[Esc]* makes a switch from the Input Mode to Command Mode. A : in the Command Mode switches **`vi`** to the ex Mode.
5.2 (i) `I/*`*[Esc]* (ii) `A*/`*[Esc]*
5.3 (i) `$xx` (ii) `02x`
5.4 You have modified the buffer but have not saved it. Use **`:q!`** to abort editing or **`:x`** to save and exit.
5.5 (i) Writes current line through line 10 to `foo`. (ii) Overwrites `foo` with last line of buffer. The commands are executed in the ex Mode. If `foo` exists, the first command won't work; you'll then have to use **`:.,10w! foo.`**
5.6 Use **`:!who`**. For extended sessions with the shell, use **`:sh`** to make a temporary exit to the shell. Use *[Ctrl-d]* or **`exit`** to return to the editing mode of **`vi`**.
5.7 (i) `/#include` (ii) `4dd` (iii) `1G` (iv) `P`
5.8 Assuming that the cursor is at the beginning of the line, (i) `wi-n` *[Esc]*`lr"` and `$.2b2x` (ii) `if`*[Esc]*`wastderr,` *[Esc]* (iii) `2w10yl$P`
5.9 (i) `yl` (ii) `y0` (iii) `10yw` (iv) `y/esac`
5.10 (i) `dG` (ii) `1GdG`
5.11 `100G:.,$w foo`
5.12 Search with **`/printf`**(*[Enter]*, enter `7s` followed by `fprintf(stderr,`. Use **`n`** to repeat the search and **`.`** to repeat the action on other lines.
5.13 Use **`/printf`** and then use **`N`**.
5.14 (i) `:1,10s/cnt/count/g` (ii) `:.s/cnt/count/g` (iii) `:1,$s/cnt/count/g`. For interactive replacement, add the `c` parameter at the end of the substitution command.
5.15 Use **`/{`***[Enter]* and **`d/}/+`**
5.16 The buffer may not be saved and the swap file won't be removed. Use **`vi -r foo`** to perform a recovery, and if the contents seem to be acceptable, save and exit the editor and then remove the swap file.
5.17 You should have copied the lines to a buffer with **`"a20yy`**.
5.18 Save to a different file with **`:w`** *filename*, and then rename the file after quitting **`vi`**.
5.19 First use **`:!chmod +x %`***[Enter]* and then **`:!%`***[Enter]*.
5.20 Use the `c` operator with **`/:c/Packet Switching/`***[Enter]*.
5.21 (i) `1,5s/^/ /c` (ii) `1,5s/^/ /`
5.22 Make these settings in `.exrc`: (i) `:set autowrite` (ii) `:set tabstop=3`
5.23 (i) `:1,10w passwd1` (ii) `:11,20w passwd2` (iii) `:21,$w passwd3`

5.24

Command	*Action*
`1G`	Moves cursor to line 1
`lx`	Moves cursor right and deletes space between `#` and `include`
`wi` *[Esc]*	Moves a word forward to the `<` in `<errno.h>` and inserts a space
`O`	Opens line above line 1
`#include <stdio.h>`*[Esc]*	Adds line `#include <stdio.h>` at top
`2j$`	Moves cursor down 2 lines and then to end of line (beginning with `void quit`)
`i, int exit_status`*[Esc]*	Inserts comma, a space and string `int exit_status`
`J`	Pulls up `{` below
`j`	Moves cursor to line containing first `printf`
`I/*` *[Esc]*	Adds comment prefix `/*` to line
`A */`*[Esc]*	Adds comment suffix `*/` to line
`:w`*[Enter]*	Saves buffer and continues editing
`j^`	Moves cursor down to beginning of first word of line (`p` in `printf`)
`if`*[Esc]*	Inserts `f` to change `printf` to `fprintf`
`w`	Moves cursor to `(` in `fprintf(`
`astderr,` *[Esc]*	Appends `stderr,` and a space
`2l`	Moves cursor to `e` in `error`
`~`	Changes `error` to `Error`
`7w`	Moves cursor to closing quote
`iquitting program\n`*[Esc]*	Inserts `quitting program\n`
`jdd`	Deletes following line containing `printf`
`2w`	Moves cursor to `1` in `exit(1)`
`sexit_status`*[Esc]*	Replaces `1` with `exit_status`
`:x`*[Enter]*	Saves and quits **`vi`**

Chapter 6

6.1 (i) Enter the text up to `!`. Next use **`C-q C-g`** to enter the control character, followed by `"`. (ii) `C-x C-s` (iii) `M-x shell`*[Enter]* `chmod +x foo` (iv) `C-x b`*[Enter]* (v) `C-x C-c`

6.2 **`emacs`** reads `.emacs` on startup. Either rename the file or use **`emacs -q`**.

6.3 Use **`M-x`** first and then use (i) `got`*[Tab]*`l`*[Tab]* (ii) the Up key.

6.4 Assuming that **`emacs`** is in the insert mode, use **`C-a /* C-e */`**.

6.5 If it is a single character, then it must be *[Esc]*. (i) *[Ctrl-q][Esc]* (ii) Create a region using **`C-`***[Spacebar]* and **`C-f`**, copy it with **`M-w`** and place it with **`C-y`**.

6.6 **`emacs`** (i) attempts to quit. (ii) saves and quits without prompting. (iii) suspends the session which is resumed with **`fg`** at the shell prompt.

6.7 First use **`C-x h`** to define the entire file as a region and then use **`C-x C-x`** to toggle.

6.8 Use **`M-x shell`** to open a shell and a buffer. After all commands have been executed, use **`C-x b`** to return to your work buffer.

6.9 (i) `C-x C-s` (ii) `C-x C-w foo2`

6.10 **`recover-file`** loads the autosaved version of the file, but **`revert-buffer`** loads the one that was saved *explicitly* by the user.

6.11 (i) `C-s #include`*[Enter]* (ii) `C-a M-4 C-k` (iii) `M-<` (iv) `C-y`

6.12 Only **C-d** is not a kill operation; it can't be restored from the kill ring.
6.13 First use **C-***[Spacebar]* and then (i) `C-f M-w` (ii) `C-a M-w` (iii) `M-10 M-f M-w` (iv) `C-s esac`*[Enter]* `M-w`.
6.14 First copy the lines using **C-***[Spacebar]* **M-5 C-e M-w**. Open the second file with **C-x C-f foo2** and then use **C-y** to paste the copied text. Use **C-x b** to toggle.
6.15 (i) `M-3 M-d` (ii) `C-u C-u M-d`
6.16 (i) `C-`*[Spacebar]* `M-> C-w` (ii) `C-x h C-w`
6.17 The **write-region** command saves a region to a file.
6.18 **M-3 C--** will work both for undoing and redoing.
6.19 Incremental search because it looks for the string as it is keyed in.
6.20 First use **C-s printf***[Enter]* and then use **C-r***[Enter]* repeatedly. To make the searches case-insensitive, place the entry `(setq-default case-fold-search t)` in `~/.emacs`.
6.21 The buffer may not be saved at that moment. Use the **recover-file** command to retrieve the autosaved version.
6.22 (i) `M-c` (ii) `M-5 M-u` (iii) `C-x h C-x C-u`
6.23 (i) `C-x 2` (ii) `C-x o` (iii) `C-x C-f foo2` (iv) `C-x C-c` and then `!`.
6.24 First delete lines 11 to 20 (with **M-10 C-k**), follow it with a dummy non-kill operation (like moving the cursor) and then delete the other lines. Restore the first group with **C-y** at both places. At the second location, undo the effect of **C-y** and use **M-y**.
6.25 (i) `C-h k M-y` (ii) `C-h f revert-buffer`
6.26 Make these settings in `~/.emacs`: (i) `(setq auto-save-interval 100)` (ii) `(setq tab-width 4)`

Chapter 7

7.1 `foo` is (i) overwritten. (ii) created before data is appended. (iii) truncated before **cat** is run. (iv) overwritten to contain only a newline character.
7.2 The shell interprets the wild cards to match filenames and then rebuilds the command line with all metacharacters removed. It then uses the services of the kernel to execute the command.
7.3 (i) `[fF]oo[125]` (ii) `quit.[cho]` (iii) `[Ww]atch.[Hh][Tt][Mm]*` (iv) `.*.swp`
7.4 **ls .*** displays all filenames in the current directory including the hidden files as well as all files in the parent directory excluding the hidden files. **ls *.** displays all filenames in the current directory ending with a dot.
7.5 (i) `rm .[!.]*` (ii) `rm \#*\#` (iii) `rm [0-9][0-9][0-9]*` (iv) `rm *.?` (i) won't work in the C shell.
7.6 (i) `[0-9]?*[!a-zA-Z]` (ii) `???*` (iii) `?*2004*?`
7.7 All filenames (i) comprising at least 5 characters where the first character is alphabetic. (ii) containing at least one numeral. (iii) not ending with a numeral. (iv) having a 2-character extension that is not `.sh`.
7.8 This is quite troublesome; **rm \-*** or **mv \-*** won't work. First create a directory `bar2` at the same hierarchical level as `bar`. Next, move all files except this file to `bar2` with **mv [!-]* ../bar2**. Now, move up with **cd ..**, remove `bar` with **rm -r bar** and rename `bar2` to `bar`.
7.9 (i) `rm -rf \*  My\ Documents` (ii) `rm -rf "*" "My Documents"`
7.10 Quotes protect most special characters from interpretation by the shell. Single quotes protect all special characters but double quotes enable the `$` and `` ` `` to be interpreted as the variable evaluation and command substitution characters. Single quotes protect double quotes and double quotes protect single quotes.
7.11 When there is a single file matching this pattern in the current directory. Use **rm chap0\[1-5\]** to remove the file.
7.12 Input redirection occurs before error redirection.
7.13 In the first case, **wc** opens `foo` and also prints the filename. In the second case, the shell opens `foo` and connects it to **wc**'s standard input, so **wc** can't print the filename.
7.14 `cat foo1 - foo2`

7.15 `newlist` features as a filename in the contents of `newlist`, which means redirection occurs before the command is executed.

7.16 `( echo "<html>" ; cat foo.html ; echo "</html>" ) > htmlfile`

7.17 File descriptors are small integers allocated by the kernel to open files. All operations on the file can then be performed by referring to the file descriptor. 0 and 1 are the default file descriptors of standard input and standard output, which means < and > are the same as `0<` and `1>`. The file descriptor for standard error is 2 and needs to be explicitly affixed to the > symbol.

7.18 At the end of the **`echo`** statement, add (i) `1>&2` (ii) `>/dev/tty`.

7.19 *prog1* must use standard output and *prog3* must use standard input. *prog2* must use both.

7.20 (i) `cal `date "+%m 20%y"`` (ii) `ls -l `cat foo``

7.21 The command saves the number of directories in the home directory tree in `list`, but will work only in double quotes or no quotes.

7.22 Only if a symlink named `scripts` is found and which points to a directory. **`pwd`** displays the pathname of the symlink, but **`/bin/pwd`** displays the pathname of the directory pointed to.

7.23 A filter is a command that writes to standard output and reads standard input when used without a filename as argument. `` `foo` `` will work if `foo` writes to standard output and the output is also a valid command line.

7.24 The - signifies standard output in **`tar`**, and **`gzip -c`** writes to standard output. To archive and compress the directory `bar`, use **`tar -cvf - bar | gzip -c > bar.tar.gz`**.

7.25 (i) `count=`cat *.c | wc -c`` (ii) `count=`wc -l < foo``

7.26 (i) `$x` evaluates to null and `=5` is executed as a command, which usually results in an error. (ii) The string `pwd`, `=`, and the output of the **`pwd`** command are concatenated and saved in the `directory` variable.

7.27 The inner command substitution sequence needs escaping:

```
count=`cat \`cat foo\` | wc -c`
```

Chapter 8

8.1 `$$` signifies the PID of the current shell, and `$!` represents the PID of the last background process. A shell script is run in a sub-shell, the reason why `$$` is different there.

8.2 Both are arranged in a hierarchical structure and maintain a parent-child relationship. A parent may have multiple children but a child can have only one parent. Both file and process have an owner and group owner.

8.3 No, part of the process image is shared.

8.4 The parent may (i) wait for the child to die. (ii) may perform other functions. The shell normally waits for the child to die, but when a command is followed by the `&` symbol, the shell doesn't immediately wait for the child's death.

8.5 Displays the total number of processes running in the system plus one (for the header.

8.6 Daemons are processes that run continuously in the background and have no controlling terminal. **`sendmail`** handles mail, **`httpd`** runs the Web service and **`lpsched`** handles print jobs. Use **`ps -e`** to view these processes.

8.7 (i) `lpsched` or `lpd` (ii) `sendmail` (iii) `sshd`

8.8 Had **`cd`** been an external command, it would have been executed in a child process, which means that the change wouldn't have been permanent after the child's death. It would then be impossible to change directories.

8.9 The **`init`** process with the PID 1. It is the parent of all daemon processes and login shells.

8.10 A process is created by first replicating itself and then overwriting its own image with that of a separate program. The **`fork`** system call replicates the current process, while an exec function is responsible for running a new program.

8.11 (i) All open file descriptors including the three standard streams (ii) real UID and GID (iii) effective UID and GID (iv) environment variables (v) current directory.

8.12 `foo` displays the value of `PATH` but not `x`. `PATH` is an environment variable, so it is visible in all processes. `x` is not an environment variable, so use **`export x`** in the command line before running `foo`.

8.13 A zombie is a dead child whose exit status has not been picked up from the process table by its parent. A zombie can't be killed except by rebooting the system.

8.14 (i) False, `SIGKILL` and `SIGSTOP` can't be ignored. (ii) False, a parent may decide not to wait. (iii) True (iv) False, a job is a group of processes working toward a common goal.

8.15 A process run with `&` is killed when the user logs out but not one that is run with **`nohup`** and `&`. However, some shells allow a process to run in the background with `&` and still not be killed on logging out.

8.16 A signal is the mechanism by which a process is made aware of an event. The interrupt key generates the `SIGINT` signal and *[Ctrl-z]* generates `SIGTSTP`. We should use signal names rather than numbers because the same signal name may represent two different signal numbers on two systems.

8.17 A job is a group of processes working toward a common goal. A pipeline is a job that comprises multiple processes. (i) *[Ctrl-z]* (ii) `bg` (iii) `fg`

8.18 (i) **`dial.sh`** runs every minute throughout the year. (ii) `00-60` is invalid (Valid values are 0-59); `22-24` is invalid (Valid values are 0-23); `30` is invalid as February has 28 or 29 days. (iii) The **`find`** command runs at 21:30 hours every day.

8.19 `00,30 08-18 * * 1,3,5 connect.sh`

8.20 The change of directory is not permanent because `foo` is executed in a child process.

8.21 **`exit`** kills the current shell and returns control to its parent. **`( exit )`** doesn't log you out because the command is executed in a child process.

8.22 Use **`ps -e | grep cron`** to see whether **`cron`** is running. First check whether `cron.allow` exists, and whether your user-id exists in that file. Next repeat the exercise for `cron.deny`.

8.23 She should place the names of the users to be denied in `at.deny` and `cron.deny`.

Chapter 9

Unless otherwise stated, all questions assume Bash as the working shell.

9.1 An interactive shell is used to run commands. A noninteractive shell runs shell scripts. History, job control and aliases, and variables like `PS1` have meaning only in an interactive shell.

9.2 `HOME` and `SHELL`.

9.3 Because programs are designed to be shell-independent. They expect the *name=value* format.

9.4 (i) Use **`ksh`**. (ii) Use **`chsh`** or modify `/etc/passwd`. `$SHELL` is not affected by (i) but is modified in (ii) when the user logs in again.

9.5 `MAIL` stores the location of the user's mailbox. `MAILCHECK` specifies the frequency of checking this mailbox. When the shell finds that the modification time of the mailbox has changed, it issues the message `You have mail`.

9.6 Set `CDPATH` to include the parent directory (`..`)—typically, `CDPATH=.:..`

9.7 It will display the event number before the `$`.

9.8 (i) `OLD_PS1="$PS1" ; PS1='[\h-\u \w]'` (ii) `PS1="$OLD_PS1"`

9.9 (i) `alias lh="ls -d .*"`

(ii) `alias lsl="find . -type l -exec ls -ld {} \;"`

(iii) Can't be done in Bash, but only in the C shell.

9.10 (i) Repeats event number 50. (ii) Displays command prior to the previous one. (iii) Repeats previous command. (iv) Replaces first occurrence of `doc` in the previous command with `bak`. In Korn, the equivalent commands are (i) `r 50` (ii) none (iii) `r` (iv) `r doc=bak`.

9.11 The second command is to be entered as (i) `cd $_` (ii) `^bak^doc` (Bash), `r bak=doc` (Korn) (iii) `cp $_ $_.bak`
9.12 You tried to execute the last argument of the previous command. This argument was a file that had no execute permission.
9.13 You need to use in-line editing. First use **`set -o vi`**, press *[Esc]*, next **`/^tar`** and then press **`n`** repeatedly.
9.14 Switches to (i) henry's home directory. (ii) the directory `henry` in the user's home directory. (iii) the last directory visited. (iv) Same as (iii).
9.15 The commands to be executed once, on login, are placed in the profile. The rc file contains settings that need to be executed every time a sub-shell is created.
9.16 (i) By logging out and logging in. (ii) Using the dot or **`source`** command to execute `.profile`.
9.17 You should log out and log in every time you change the file. If you use the **`.`** or **`source`** command to execute it, `PATH` would have the value of `$HOME/bin` added to it repeatedly.
9.18 Place a statement in the rc file to execute `~/.alias` using the dot or **`source`** command. Aliases are not available in shell scripts of both shells.
9.19 Create a file named `~/.last_login` in the `.profile` (with **`echo > ~/.last_login`**) and define this alias:

```
current="find . -type f -newer $HOME/.last_login -print"
```

9.20 Create an alias for **`cd`**:

```
alias cd 'cd \!* ; set prompt = "[$cwd] "'
```

Chapter 10

10.1 Use **`comm -13 foo1 foo2`**. The command won't work properly if the files are not sorted.
10.2 `a.out | cmp foo -`
10.3 (i) `head -n 10 foo | tail +5` (ii) `tail -n 2 foo | head -n 1`
10.4 `pr -t -n foo | sort -nr | cut -f2`
10.5 `tail +11 /etc/passwd | cut -d: -f1`
10.6 `ps -e | tr -s " " | sort -k 4`
10.7 `alias lvi="vi `ls -t | head -n 1`"`
10.8 `length=`head -n 1 shortlist | wc -c``
10.9 `year=`date | cut -d" " -f6 | cut -c3-` ; echo $year`
10.10 `ls -ld `echo $PATH | tr : " "` `
10.11 `sort -t: -n -k 4,4 -k 3,3 /etc/passwd`
10.12 Use **`date | tr ' ' '\n' > foo`** to have each field on a separate line. To revert to the original **`date`** output, use **`tr '\n' ' ' < foo`**.
10.13 Delete all characters except `?` and then make a character count with **`wc`**:

```
tr -dc '?' < foo | wc -c
```

10.14 `tr -d '\015' < typescript > typescript.new`
10.15 `sort -t: -nr -k 3 /etc/passwd | cut -d: -f1,3 | head -n 1`
10.16 First cut out the names from both files and create two sorted lists:

```
cut -d: -f1 foo1 | sort > foo1.sorted
cut -d: -f1 foo2 | sort > foo2.sorted
```

The following three commands display the three lists:

```
comm -23 foo1.sorted foo2.sorted
comm -13 foo1.sorted foo2.sorted
comm -12 foo1.sorted foo2.sorted
```

10.17 (i) `who | cut -d" " -f1 | sort | uniq -d`
(ii) `who | cut -d" " -f1 | sort -u | mailx root`
10.18 Both remove duplicate lines from a file, but **`uniq`** requires the file to be sorted.
10.19 Cut out all the fields of the **`date`** command output except the fourth and fifth (representing the time and time zone) and pipe it to **`uniq`**: `cut -d" " -f1-3,6 foo | uniq -c`
10.20 Assuming that the **`ps -e`** output has leading spaces, we need to pick up the fifth field with **`cut`**:

```
ps -e | tail +2 | tr -s " " | cut -d" " -f5 | sort | uniq -c
```

10.21 (i) Issue these commands from `bar1`:

```
find . -type d  | sort | tail +2 > foo
mv foo .. ; cd .. ; mkdir `cat foo`
```

(ii) Run the above **`find`** command in `bar2` and then use **`cmp`**, **`comm`** or **`diff`** to compare the output.
(iii) The sequence won't work properly when

- directory names contain embedded spaces.
- directories don't have execute permission.
- directories are symbolically linked.

Chapter 11

Some questions use the file `emp.lst` whose contents are shown in Section 11.1.
11.1 A wild card is a metacharacter that is interpreted by the *shell* to match filenames. A regular expression is a pattern comprising ordinary and metacharacters that is used by a *command* to match multiple strings. It has nothing to do with the shell.
11.2 (i) **`grep`** searches for `a` in files `b` and `c`. (ii) The command doesn't work because `<HTML>` is not quoted, and the shell interprets the < and > symbols as redirection characters. (iii) **`grep`** looks for zero or more occurrences of a `*`. It thus matches all lines. (iv) If `*` expands to multiple filenames, **`grep`** looks for the first filename in the remaining files. If `*` expands to a single filename, **`grep`** searches the standard input.
11.3 The `*` in `.*` signifies that `botswana` and `birds` can be separated by any number of characters. The `*` in `*.htm*` is a wild card and the expression matches all HTML files.
11.4. All lines in `foo`. When unquoted, **`grep`** searches for a single character in all filenames beginning with a dot as well as `foo`. It thus matches all lines in these files except those containing only newline. To understand why, first use **`echo .*`**.
11.5 **`grep`** searches `foo` for a tab. This is the preferred method of making **`grep`** see the tab character if the tab key doesn't produce this character at the shell prompt.
11.6 No, the second command also matches lines containing nothing.
11.7 Use an ERE: `grep -Ev "^#|^/\*|^//" foo`
11.8 `grep `date +%m/%d/[0-9][0-9]` emp.lst | cut -d: -f2`
11.9 `who | grep -v "`date +%b %d`" | cut -d" " -f1 | sort -u | mailx root`
11.10 Lists by modification time all files containing the string `fork`. Could have used this:

```
ls -t `grep -l fork *.c`
```

11.11 `ls -l | grep "^`ls -l foo | cut -d' ' -f1`"`
11.12 (i) `jeff[er][er][iy][ey]*s*` (ii) `hitch[ei]ng*` (iii) `[Hh][ei]a*rd` (iv) `di[xc]k*s*o*n*`
(v) `[Mm][ac]gh*ee` (vi) `wood(|cock|house)`
11.13 (i) `[0-9]*` signifies nothing or any number of digit characters. `[0-9][0-9]*` signifies at least one digit character. (ii) `^[^^]` matches at the beginning of the line a character which is not a caret. `^^^` matches 2 carets at the beginning of a line.
11.14 This is useful tool to use:

```
grep -i -E 'Subject:.*(urgent|immediate)|From:.*henry' /var/mail/romeo |
mailx romeo
```

11.15 The `-e` option may not be required: `ls -l | grep -e "^-.[^w]"`

11.16 Locates UID 100 in the third field of `/etc/passwd`. The pattern gets longer when you select a field with a higher field number.

11.17 `grep -E "[^fs]printf.|^printf" foo`

11.18. `grep -E "(bill|william) christ[iy]e?" foo`

11.19 (i) `find $HOME -name "*.c" -exec grep -l "int  *main *(" {} \;`

(ii) `vi `find $HOME -name "*.c" -exec grep -l "int *main *(" {} \;``

11.20 Use the **i** and **a** commands of **sed**:

```
sed -e '1i\
<HTML>
' -e '$a\
</HTML>
' foo > foo.html
```

11.21 Use a character class that has space and tab as members:

(i) `grep -v "^[	]*$" foo`

(ii) `sed -n '/^[	]*$/!p' foo`

11.22 (i) `grep "^\..*\.$" foo` (ii) `sed -n '/^\..*\.$/p' foo`

11.23 `grep "^.\{101,149\}$" foo` (ii) `sed -n "/^.\{101,149\}$/p" foo`

11.24 (i) `grep '\([a-zA-Z0-9_]\)\1\1' foo`

(ii) `sed -n '/\([a-zA-Z0-9_]\)\1\1/p' foo`

11.25 This needs both **grep** and **sed**. Pressing a `\` before *[Enter]* places every occurrence of `ENCRYPTION` on a separate line by itself:

```
sed 's/ENCRYPTION/&\
/g' foo | grep -c ENCRYPTION
```

11.26 (i) Select that line and those that don't begin with `#!`:

```
grep -E "^#!/bin/ksh|^[^#][^!]*" foo
```

(ii) `grep -v "^/\*.*\*/$" foo`

11.27 `sed -e 's/^  *//' -e 's/  *$//' foo`

11.28 (i) It is meant to replace `print` with `printf` and vice versa but won't work if both exist in the file.

(ii) The first **s** command converts the `compute` in `computer` to `calculate` as well, leaving nothing for the second **s** command to act on. Reverse the instructions:

```
sed -e 's/computer/host/g' -e 's/compute/calculate/g' foo
```

11.29 Form a group representing zero or more instances of the `/`:

```
sed 's/\(<\/*\)B>/\1STRONG>/g' foo
```

11.30 `sed '/^ *PATH=/s/\/usr\/local\/bin//' $HOME/.profile > $$`

`mv $$ $HOME/.profile`

11.31 Extract the extension of all files with **sed**:

```
ls -1 | sed -n 's/.*\.\(.*\)/\1/p' | sort | uniq -c
```

11.32 Use **sed** to remove the blank lines from the sorted output and then add one blank line after each line:

```
sort foo | sed -e '/^ *$/d' -e 'a\
   ... blank line ....
'
```

Chapter 12

Some questions use the file `empn.lst` whose contents are shown in Section 12.1.

12.1
```
awk -F: '$7 ~ /bash$|ksh$/ { print $1, $7 | "sort -k 2" }' /etc/passwd
```

12.2
```
awk -F: '{ if ($1 == "nobody")
               foundline = 1
           if (validuser == 1)
               lastuid = $3 > lastuid ? $3 : lastuid
           if (foundline == 1)
               validuser = 1 }
END { print "The next available UID is", ++lastuid }' /etc/passwd
```

12.3
```
find / -print | awk 'length > 100'
```

12.4 Sort the **find** output so awk can ignore lines with identical inode numbers:
```
find . -type f -ls | sort | awk 'BEGIN { prev = "" }
{ if ($1 != prev )
    tot+= $7
  prev = $1
} END { print tot }'
```

12.5 This two-line script should do the job:
```
pid=`ps | awk '$4 ~ /'$1'/ {print $1}'`
kill $pid
```

12.6
```
tar tvf archive.tar | awk '/\/$/ {print $NF}'
```

12.7
```
who | awk '{ kount[$1]++ }
      END { for ( user in kount )
                print user, kount[user] }'
```

12.8
```
ps -ef | awk '{ kount[$1]++ }
         END { for ( user in kount )
                   printf "%-10s %4d\n", user, kount[user] }'
```

12.9 The **awk** program should be placed in a shell script and should use double quotes and a generalized field number:
```
awk "{tot += \$$1} END {print tot}"
```

12.10 Assuming that the **id** output has this form:
```
uid=500(sumit) gid=100(users)
```

the following program should set `LOGNAME`:
```
LOGNAME=`id | awk '{ begpos = index($0, "(")
   endpos = index($0, ")")
   print substr($0, begpos + 1, endpos - begpos - 1)
}'` ; export LOGNAME
```

12.11 The stamp data is assumed to be in the file `want_list`:
```
awk '{
    # First extract the number from the last field
    price = substr($NF,2)

    tot += price    # Add the extracted price to the total

    # Start a loop to progressively build the string for the description
```

```
      field = 4         # From the fourth field
      description = ""
      while ( field++ < NF )
         description = description $field " "
      printf "%-12s %-7s %4d %-35s $%.2f\n", $1, $2, $3, description, price
  } END { print "TOTAL PRICE:", tot }' want_list
```

12.12

```
  awk '{ y = 127 - length($0)
         printf "%s", $0
         if ( y > 0 )
            for (i = 0; i < y; i++)
               printf "%s", " "
            printf "\n"
       }' foo
```

12.13

```
  awk '{ for (i=1; i<=NF; i++) {
             row[NR]+= $i
             col[i]+= $i
         }
         printf "%s", $0
         printf " %d\n", row[NR]
  } END { for (i=1; i<=NF; i++)
              printf "%d ", col[i]
          printf "\n"
        }' foo
```

12.14

```
  awk -F: '{ gid[$4] = gid[$4] $1 " " }
  END { for (n in gid) {
           printf "%s ", n
           print gid[n]
      }
  }' /etc/passwd
```

12.15

```
  awk 'BEGIN { FS = "'$3'" }
       { mod = NR % '$1'
         if (mod == 1)
            line = $0
         else {
            line = line FS $0
            if (mod == 0) {
               print line
               line = ""
            }
         }
       }' $2
```

12.16 The input must be sorted. You may like to modify this program to print the date also:

```
  sort -t: -k 4 empn.lst | awk -F: '{
      if ($4 != previous && NR != 1) {
          printf "\n\t\t     TOTAL PAYOUT :%8d\n",tot
          tot =0
      }
      if ( $4 != previous ) {
          printf "\nDEPT : %-12s\n\n", $4
          previous = $4
```

```
    }
    printf "%4s %-18s %-12s %6d\n", $1,$2,$3,$6
    tot+= $6
} END { printf "\n\t\t     TOTAL PAYOUT :%8d\n",tot }'
```

12.17 We assume that the output of **last** is stored in last.txt:

```
awk '{ last_field = $NF
       hour_min = substr(last_field, 2, 5)
       split(hour_min, arr, ":")
       minutes = arr[1] * 60 + arr[2]
       name[$1]+= minutes
     } END { for ( i in name) {
               hours = name[i] / 60
               mins = name[i] % 60
               printf "%s: %d hours %d minutes\n", i, hours, mins
             }
       }' last.txt
```

12.18 The following sequence run from bar1 saves the directory list in dir_list:

```
find . -type d -print | sort | tail +2 |
awk '{
       if (NR > 1) {
          string = previous "/"     # Add a / at the end of the pathname

          # Check whether previous pathname exists in the current line
          if (index($1, string) == 0 ) {
             print previous         # Prints leaves only
          }
       }
       previous = $1
} END { print previous }' > dir_list
```

tail ignores the directory . that is printed by **find**. Next, copy dir_list to bar2 and run **mkdir -p `cat dir_list`**. This replicates the directory structure of bar1 in bar2.

If there are too many directories to create, **mkdir** may not be able to create the structure in a single invocation. In that case, use the following sequence to create 100 directories at a time:

```
cat dir_list | xargs -n 100 mkdir -p
```

Chapter 13

13.1 Assuming that the files bar1, bar2 and bar3 exist, $# is initially 5. If the options are combined, $# becomes 4.

13.2 **bc** is superior to **expr** because it can also handle floating point numbers:

```
#!/bin/sh
if [ $# -ne 2 ] ; then
    echo "Enter two numbers"
else
    echo "Sum of $1 and $2 as performed by bc: \c"
    echo $1 + $2 | bc
    echo "Sum of $1 and $2 as performed by expr: \c"
    expr $1 + $2
fi
```

13.3 x has the value expr 5 + 10. If single quotes had been used, then x would have evaluated to expr $x + 10. You should be using backquotes instead to obtain the value 15.

13.4 (i) Spaces not permitted around = (1) (ii) test operators [] missing in **while**'s control command (2) (iii) quotes not closed with echo (6) (iv) $ missing in choice (8) (v) then missing (9) (vi) fi instead of

`endif` (13) (vii) `;;` required instead of `;` (13) (viii) `esac` instead of `endcase` (15)

13.5 You have to escape the `\`, `/` and `*` here:

```
#!/bin/sh
for file in *.c ; do
   sed -n '/^\/\*/,/*\/$/!p' $file > $$
   mv $$ $file
done
```

13.6 This script inserts a newline at the end of every occurrence of the pattern. Place a `\` before pressing *[Enter]* in the first line:

```
#!/bin/sh
sed 's/$1/&\                                    Pattern is the first argument
/g' $2 | grep -c "$1"                           Filename is the second argument
```

13.7 This code section should be placed in `/etc/profile`. It makes use of a file `/etc/user.deny` which contains a line for every user who is not allowed to log in multiple times:

```
DENYFILE=/etc/user.deny
if [ `who | cut -d" " -f1 | grep -c "^$LOGNAME$"` -gt 1 ] ; then
    if grep "^$LOGNAME" $DENYFILE >/dev/null ; then
        echo "You are not authorized to log in more than once"
        sleep 2                                 # Let user see the message
        exit 0
    fi
fi
```

13.8 Yes, it does; it stores the output of the **set** command itself in positional parameters.

13.9 Append the symbols `2>&1` to every statement in the script that has the potential of producing error messages. Redirect the script to a file on execution.

13.10

```
#!/bin/sh
case $# in
   0) filenames=`ls *.c` ;;
   *) filenames="$*" ;;
esac
for file in $filenames ; do
    echo "Displaying first 10 lines of $file\n"
    head -n 10 $file
    echo "\nDelete file $file? (y/n) \c"
    read reply
    [ "$reply" = "y" ] && rm $file
done
```

13.11 This script uses the **:** (null) command. Regular expressions have been used to take care of whitespace:

```
#!/bin/sh
for file in *.c ; do
    if grep "f*printf *(" $file >/dev/null ; then
        if grep "^ *# *include *<stdio\.h>" $file >/dev/null ; then
           :
        else
           ( echo "#include <stdio.h>" ; cat $file ) > $$
           mv $$ $file
        fi
    fi
done
```

13.12 This script runs an infinite loop and leaves behind a temporary file when it is finally interrupted:

```
#!/bin/sh
ls *.pdf *.PDF > filelist1 2>/dev/null
while true ; do
   sleep 60
   ls *.pdf *.PDF > filelist2 2>/dev/null
   if ! cmp filelist[12] 2>/dev/null ; then
       echo “The following files have been created in the last minute:”
       comm -13 filelist[12]
   else
       echo “No file has been created in the last minute”
   fi
   mv filelist2 filelist1
done
```

13.13

```
#!/bin/sh
[ $# -ne 1 ] && { echo “Usage: $0 uncompressed_filename” ; exit 1 ; }
[ -f $1 ] || { echo “File doesn’t exist” ; exit 1 ; }
[ -r $1 ] || { echo “File is not readable” ; exit 1 ; }
filename=$1
set -- `ls -ld $1`
usize=$5                 # The uncompressed file size

echo “File: $filename” 1>&2
for program in compress gzip bzip2 zip ; do
   if ! type $program >/dev/null 2>&1 ; then
       echo “Compression program $program doesn’t exist” 1>&2
       continue
   else
       ofile=”” ; option=””
       case $program in
       compress) extension=Z ; uprogram=uncompress ofile=””;;
           gzip) extension=gz ; uprogram=gunzip ofile=”” ;;
          bzip2) extension=bz2 ; uprogram=bunzip2 ofile=”” ;;
            zip) extension=zip ; uprogram=unzip ; option=”-o”
                 ofile=$filename.$extension ;;
       esac
       # Compress the file
       $program $ofile $filename >/dev/null
       set -- `ls -l $filename.$extension`
       size=$5                          # Get the compressed size
       echo “$program $usize $size”
       # Uncompress the file
       $uprogram $option $filename.$extension >/dev/null
   fi
done | awk ‘{ printf “%-9s %d %d %2.1f\n”,
              $1, $2, $3, ($2-$3)*100/$2 | “sort -n -k2” }’
```

13.14 This script features a clever use of the **sed** command:

```
#!/bin/sh
CONFIG_FILE=config.txt
echo “\
      MENU

1. Add
2. Modify
3. Delete
4. View
\n CHOICE: \c”
```

```
read choice
case $choice in
   1|2) echo "Enter setting: \c"
        read setting
        variable=`expr "$setting" : '\(.*\)=.*'`
        case $choice in
           1) if ! grep "^$variable=" $CONFIG_FILE >/dev/null ; then
                 echo "$setting" >> $CONFIG_FILE
              else
                 echo Variable exists
              fi ;;
           2) if grep "^$variable=" $CONFIG_FILE >/dev/null ; then
                 # Replace the entire entry
                 sed "/^$variable=/s#.*#$setting#" $CONFIG_FILE > $$
                 mv $$ $CONFIG_FILE
              else
                 echo "Variable doesn't exist"
              fi ;;
        esac ;;
   3|4) echo "Enter variable name: \c"
        read variable
        case $choice in
           3) # First use grep to check whether the variable exists
              # and then select the remaining lines if it does
              # Unlike grep, grep -v doesn't return a false value if
              # the pattern is not found -- unfortunate design?
              if grep "^$variable=" $CONFIG_FILE >/dev/null ; then
                 grep -v "^$variable=" $CONFIG_FILE > $$
                 mv $$ $CONFIG_FILE
                 echo "Setting deleted"
              else
                 echo "Variable doesn't exist"
              fi ;;
           4) if ! grep "^$variable=" $CONFIG_FILE ; then
                 echo "Variable doesn't exist"
              fi ;;
        esac ;;
esac
```

13.15 We need to use **find**'s -newer option:

```
#!/bin/sh
for file in bar2/* ; do
    file=`basename $file`
    if [ ! -f bar1/$file ] ; then
       cp bar2/$file bar1
    else
       # We don't make use of the set parameters. We are only interested
       # in knowing whether find could locate a newer file.
       # And all this because find doesn't return a false exit status
       # if the file is not found!

       set -- `find bar2 -name $file -newer bar1/$file -print`
       if [ $# -gt 0 ] ; then    # find selects the file
          cp bar2/$file bar1
       fi
    fi
done
```

13.16

```
#!/bin/sh
case $# in
     3) [ -d $1 ] || { echo "$1 doesn't exist" ; exit 1 ;}
        [ -d $2 ] || { echo "$2 doesn't exist" ; exit 1 ;}
        [ -d $3 ] && { echo "$3 exists" ; exit 1 ;} ;;
     *) echo "Usage: $0 dir1 dir2 dir3"
        exit ;;
esac

bar1=$1
bar2=$2
bar3=$3

# Create the new directory $bar3 and copy all files
# from $bar1 to this directory.

mkdir $bar3 ; cp $bar1/* $bar3

# Now copy the new and newer files from $bar2 to $bar3.
# The rest of the logic is similar to Ex. 13.15.
```

13.17

```
#!/bin/sh
set -- `find /bin /usr/bin -type f -name $1 -ls`

if [ $# -gt 0 ] ; then
   inodenum=$1
   links=$4
   if [ $links -gt 1 ] ; then
       find /bin /usr/bin -inum $inodenum -print
   else
       echo "File not having multiple links"
   fi
else
   echo "File not found"
fi
```

13.18 Save this script in the file **accept_telenum.sh**:

```
#!/bin/sh
[ $# -ne 1 ] && { echo "Usage: $0 number" ; exit 1 ; }
n="[1-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]"

case $1 in
    0*) echo "Number can't begin with 0" 1>&2
        exit 1 ;;
    $n) n1=`expr $1 : '\(...\).*'`
        n2=`expr $1 : '...\(...\).*'`
        n3=`expr $1 : '.*\(....\)'`
        tele_numb=$n1-$n2-$n3
        echo $tele_numb ;;
     *) echo "Not a 10-digit number" 1>&2
        exit 1 ;;
esac
```

13.19 This script ignores multiple logins by the same user. It also leaves behind temporary files:

```
#!/bin/sh
who | cut -d" " -f1 | sort -u > userlist1
```

```
while true ; do
   sleep 60
   who | cut -d" " -f1 | sort -u > userlist2
   for username in `cat userlist1` ; do
       if ! grep "^$username$" userlist2 >/dev/null ; then
           echo "$username has logged out"
       fi
   done
   for username in `cat userlist2` ; do
       if ! grep "^$username$" userlist1 >/dev/null ; then
           echo "$username has logged in"
       fi
   done
   mv userlist2 userlist1
done
```

13.20 Use **ls** with two sets of options and then extract the necessary words:

```
#!/bin/sh
if test $# -eq 0 ; then
    echo "Enter proper command line arguments"
else
    printf "%-11s %10s  %-20s %-12s   %-12s\n" \
           "Permissions" "Size" "Filename"  "Modtime" "Access time"
    for filename in $* ; do
        set -- `ls -lud $filename`
        mdt1="$6 $7 $8"
        set -- `ls -ld $filename`
        mdt2="$6 $7 $8"
        printf "%-11s %10s  %-20s %-12s   %-12s\n" $1 $5 $9 "$mdt2" "$mdt1"
    done
fi
```

13.21 The delimiter used by **sed** has been changed from / to ^:

```
#!/bin/sh
set -- `type ksh` 2>/dev/null
case $# in
    3) pathname=$3                    # type shows pathname as third argument
       for file in *.sh ; do
           if head -n 1 $file | grep "^#!$pathname" >/dev/null ; then
               echo "File $file contains correct she-bang line"
           else
               sed "1s^.*^#!$pathname^" $file > $$
               mv $$ $file
           fi
       done ;;
    *) echo "Korn shell not found on system" ;;
esac
```

13.22 Extract the inode number with **ls -i** and then use **find** to locate the links:

```
#!/bin/sh
case $# in
   1) if [ ! -f $1 ] ; then
         echo "File doesn't exist in current directory"
         exit 1
      else
         set -- `ls -id $1`
         inum=$1
         set -- `find $HOME -inum $inum -print`
         case $# in
```

```
            1) echo "No links found; file exists only in current directory"
               exit ;;
            *) ls -l $* | tee /dev/tty | mailx $LOGNAME ;;
         esac
      fi ;;
   *) echo "Enter a single filename"
      exit 2 ;;
esac
```

13.23

```
#!/bin/sh
# Script to periodically issue reminders to save a file while editing

file=$*
while true
do
   sleep 180
   tput cup 24 0 ; tput smso
   echo "\007TIME TO SAVE! \c"           # Use echo -e if using bash
   tput rmso
done &
vi $file
kill $!                                  # Kills shell that starts loop
pid=`ps | awk '/sleep/ { print $1 }'`    # sleep is still running though
kill $pid                                # so kill it
```

13.24 You need to supply the full name or number. The name must be enclosed within quotes:

```
#!/bin/sh
# Script requires the complete name or a 10-digit number

TELEDIR=$HOME/teledir.txt
TMP=foo
case $# in
   1|2) ;;
     *) echo "Usage: $0 [name number]"
        exit 1;
esac

case $# in
     1) case $1 in                  # Either name or number
         [0-9]*) accept_telenum.sh $1 > $TMP || exit
                 search_number=`cat $TMP`
                 if ! grep "$search_number$" $TELEDIR ; then
                     echo "Number doesn't exist"
                 fi ;;
       [A-Za-z]*) search_string=$1
                 if ! grep "^$search_string:" $TELEDIR ; then
                     echo "Name doesn't exist"
                 fi ;;
        esac ;;
    2) case $1$2 in                # Both name and number
         [A-za-z]*[0-9]*) accept_telenum.sh $2 > $TMP || exit
                          search_number=`cat $TMP`
                          search_pattern="$1: $search_number"
                          if ! grep "$search_pattern" $TELEDIR ; then
                              echo "Adding entry"
                              echo "$search_pattern" >> $TELEDIR
                          else
                              echo "Entry exists"
                          fi ;;
```

```
                        *) echo "Name must occur before number"
                           exit 1 ;;
        esac ;;
esac
rm $TMP 2>/dev/null
```

Chapter 14

14.1 Unlike the telephone system, TCP/IP breaks up data into packets which take different routes to reach the destination. Lost packets are also resent. There is no dedicated connection between sender and receiver.

14.2 An FQDN, representing a fully qualified domain name, is a host specification that uses the hostname and its domain in a dot-delimited string. This string uniquely identifies the host. Two hosts on the Internet may have the same hostname, but their FQDNs will always be different.

14.3 The resolver looks up `/etc/hosts` when handling simple hostnames. It generally looks up the name server when using FQDNs.

14.4 (i) Hierarchical structure of hostnames, (ii) distributed database, (iii) delegation of authority. The hosts file must be maintained in every host of a network while a single name server will do in DNS. DNS also specifies the mail host that receives mail for a domain.

14.5 The name server maintains all FQDN-IP address mappings for a domain. If a name server can't handle an FQDN it refers the matter to another name server.

14.6 No, because the change of directory took place in a child process (**`ftp`**), and changes in the child are not transmitted to the parent.

14.7 By encrypting the data with (i) A's private key (ii) B's public key.

14.8 Encrypt that data using a symmetric key and then encrypt the key with either the sender's private key or the recipient's public key.

14.9 So the next time you connect to the host, you are assured that it is the same host.

14.10 Run **`ssh-keygen -t rsa`** and enter the passphrase. Copy the file containing the public key to other machines.

14.11 The command **`eval `ssh-agent -s`**` invokes the SSH agent which sets two variables. **`ssh-add`** hands over the private key to the agent and this key remains in memory as long as the user is logged in.

14.12 Unlike **`ftp`**, **`scp`** (i) can copy a directory structure. (ii) is secure because all transmission is encrypted.

14.13 X reverses the normal client-server paradigm by assigning the terminal-handling job to the server program while the application itself runs as the client.

14.14 First, juliet has to use **`xhost +saturn`** to allow romeo to write to her display. `romeo` can then simply use **`netscape -display uranus:0.0`**. romeo doesn't need to run X but juliet does.

14.15 `DISPLAY` needs to be set only once so no client needs to be used with the `-display` option.

14.16 In a situation where users download their mail using a dialup line, the MTA in the ISP's mail server doesn't hand over the mail to the MDA but maintains it in a POP server. Users download this mail using a POP client.

14.17 URL expands to Uniform Resource Locator. It is *uniform* because the resource description is uniform irrespective of the protocol used. All protocols access a *resource* which may be a file or a directive to run a program. The *locator* specifies the location of the resource on the server. If the port number is left out in the URL, the default protocol is used.

14.18 A connection is not aware of the previous connection, so data in one connection is not available in another. That's why HTTP is known as a stateless protocol. In a Keep-Alive connection, the connection is not closed after a resource is fetched but only after a specified period of inactivity.

14.19 (i) HTML is a text-based language, hence HTML documents are small in size. (ii) An HTML document can be viewed by any browser on any machine running any operating system. (iii) The HTML specification is nonproprietary.

14.20 The domain name is case-insensitive, but whether `CATALOG.HTML` is case-insensitive depends on the operating system. On UNIX systems, the two URLs are not identical.

14.21 (i) 11 (ii) 1

14.22 The server usually passes this data to the CGI interface to call up an external program. This program processes the data and returns HTML output to the server for onward transmission to the client.

14.23 A helper application is an external on-disk program that is invoked by a browser to handle a fetched resource that it is incapable of handling on its own. The browser looks up `mailcap` to associate the content type with the helper application to be used. To handle files fetched by FTP as well as files on the local machine, the browser looks up the file's extension in `mime.types` to identify the content type. It then uses `mailcap` to identify the helper application for that content type.

14.24 MIME expands to Multipurpose Internet Mail Extensions. It is used to handle both mail attachments and Web resources. SMTP can handle only 7-bit characters, so it can't transport binary data without first encoding them. It can't also handle lines longer than 1000 characters. The MIME standard provides for encoding the data and provides two headers, Content-Type and Content-Transfer-Encoding, that accompanies every mail attachment. At the receiving end, MIME is used to identify both the method to be used in decoding the data and the application to handle it.

Chapter 15

15.1 Comments are shown against line numbers:

1: Should be in line 2.

2: Should be in line 1 and `!` to follow the `#`.

3: A `;` to be placed as statement terminator.

4: Should be `$bs`.

6: Opening curly brace missing.

7: Should be `\n`.

9: Should also have a `\n` before closing quote.

15.2 `perl -ne 'print $_ . "\n"' foo`

15.3 It's simpler to use **`split`** with a null string (`//`):

```
#!/usr/bin/perl
print("Enter a string: ") ;
$string = <STDIN> ;
chop($string) ;
@arr = split(//, $string) ;
$length = @arr ;
for ( $x = 0 ; $x < $length ; $x++ ) {
     print "$arr[$x]\n" ;
}
```

15.4

```
#!/usr/bin/perl
$number = $ARGV[0] ;
@arr = split(//, $number) ;
$multiplier = 1 ;
foreach $bit (reverse @arr) {     # Works from end of array first
    $tot = $tot + $bit * $multiplier ;
    $multiplier *= 2 ;
}
print "The decimal number of $number is $tot\n" ;
```

15.5 Find the numeric GID of the user in the first loop and then use it in the second loop to retrieve its name representation:

```
#!/usr/bin/perl
open(FILEIN, “/etc/passwd”) ;
open(FILEIN2, “/etc/group”) ;

$prevuid = 0;
while (<FILEIN>) {
   ($username, $x, $uid, $gid, $gcos) = split(/:/, $_);
   if ($uid > $prevuid) {
        $uidmax = $uid; $prevuid = $uid;
        $groupid = $gid;
        $loginname = $username;
        $realname = $gcos;
   }
}
while (<FILEIN2>) {
   ($groupname, $x, $gid) = split(/:/, $_);
   if ($gid == $groupid) {
      last;
   }
}
print “$uidmax $loginname $realname $groupid $groupname\n”;
```

15.6

```
#!/usr/bin/perl
$weekstring = “SunMonTueWedThuFriSat”;
for ($i=0 ; $i < 7; $i++) {
    $weekday[$i] = substr($weekstring, $i*3, 3);
    print uc($weekday[$i]) . “\n” ;
}
```

15.7

```
#!/usr/bin/perl
open(FILEIN, $ARGV[0]) || die(“Cannot open file”);
while (<FILEIN>) {
   if (/^\s*\w+=.*\s*$/) {  # If a variable is found
      s/^\s+//;             # then remove leading whitespace
      s/\s+$//;             # and also trailing whitespace
      $var = $_ .”\n”;
      $var[$i++] = $var;    # Save all variable settings in memory
   }
}
print sort @var;
```

15.8 The environment variables are available in the associative array, `%ENV`:

```
#!/usr/bin/perl
foreach $key (sort(keys %ENV)) {
    print “$key” . “=” . “$ENV{$key}\n” ;
}
```

15.9 We need two **`foreach`** constructs:

```
#!/usr/bin/perl
@entire_file = <> ;                  # Read entire file into an array
chop(@entire_file) ;
foreach $line(@entire_file) {
    @arr = split(/\W+/, $line) ;     # Split on a nonword character
    foreach $word (@arr) {
        $count{$word} += 1 ;
    }
}
```

```
foreach $word (sort (keys (%count))) {
    print("$word: $count{$word}\n") ;
}
```

15.10 This program reads the file to an array and then reads the array twice:

```
#!/usr/bin/perl
open(FILEIN, $ARGV[0]) || die ("Cannot open file") ;
@file = <FILEIN>;                  # Read file into array
$lines = $#file;                   # Number of lines in file
$maxlength = 0;

# Read the array
for ($i=0; $i <= $lines; $i++) {
    chop($file[$i]);
    $linelength = length($file[$i]);
    if ($linelength > $maxlength) {
       $maxlength = $linelength;   # Sets length of longest line
    }
}

# Now read the array again:
for ($i=0; $i <= $lines; $i++) {
    $linelength2 = length($file[$i]);
    if ($linelength2 < $maxlength) {
       $appendstring = " " x ($maxlength - $linelength2);
    } else {
       $appendstring = "";
    }
    print $file[$i] . $appendstring . "\n" ;
}
```

15.11 Use **`find / -mtime +365 -print | perl -ne 'chop ; unlink ;'`**. **`perl`** is invoked just once and deletes all selected files. **`find`** with `-exec rm` will call **`rm`** for every file found.

15.12 (i) `perl -p -i -e "s/^#!.*/\/usr\/local\/bin\/perl/" *.pl`

(ii) It's safe to test this program by saving the transformed output in a `.bak` file:

```
#!/usr/bin/perl
foreach $file (`ls *.pl`) {
    chomp($file);
    $outfile = $file . ".bak" ;
    open(FILEIN, "$file");
    open(FILEOUT, ">$outfile");
    while (<FILEIN>) {
        if ($. == 1) {
           if (/^#!\/usr\/local\/bin\/perl/) {
               ;
           } elsif (/^#!/) {
               s/.*perl/#!\/usr\/local\/bin\/perl/;
           } else {
               print FILEOUT "#!/usr/local/bin/perl\n";
           }
        }
        print FILEOUT;
    }
    close(FILEOUT);
    close(FILEIN);
}
```

15.13 `perl -p -i -e "tr/[a-z]/[A-Z]/" foo`

15.14 For convenience, we used the % as the delimiter for the **s** command:

```
#!/usr/bin/perl -n
s%(HREF=”).*/([^/]+”)%\1\2% ;
s%(SRC=”).*/([^/]+”)%\1\2% ;
print ;
```

15.15

```
#!/usr/bin/perl
open(FILEIN, $ARGV[0]) || die(“Cannot open file”);
open(FILEOUT, “>$ARGV[1]”) || die(“Cannot open file”);
while (<FILEIN>) {
    print FILEOUT $_ ;
}
```

15.16 **sed** can pick up a single segment, but this program removes all comment lines:

```
#!/usr/bin/perl
open(FILEIN, “foo.c”) || die(“Cannot open file”);
while (<FILEIN>) {
    if (/^\/\*/) {                # Opening /* found
       $pflag = 1;                # Sets a flag; doesn’t print
    }
    if ($pflag != 1) {
       print ;                    # Prints only non-comment line
    }
    if (/\*\/$/) {                # Ending */ found
       $pflag = 0;                # Resets the flag; doesn’t print
    }
}
```

15.17 This program uses the **splice** function:

```
#!/usr/bin/perl
open(FILEIN, “foo”) || die(“Cannot open file”);
while (<FILEIN>) {
    chomp;
    if (/^\d+$/) {
       $larr[$i] = $_ ;        # Store the numbers in an array
       $i++;
    } else {
       $rarr[$i] = $_ ;        # Store the country names in another array
       $i++;
    }
}

$k = $i / 2;                   # k represents the number of sets
splice(@rarr, 0, $k);          # Need to delete half of the country array
for ($k=0; $k <= $#larr ; $k++) {
   print $larr[$k] . “. “ . $rarr[$k] . “\n” ;
}
```

15.18

```
#!/usr/bin/perl
open(FILEIN, “/etc/passwd”);
open(FILEOUT, “>passwd”);
while (<FILEIN>) {
   split(/:/) ;
   $uid = $_[2] ;
   $shell = $_[6] ;
```

```
    if ($uid > 100 && $shell =~ /\/usr\/bin\/pdksh|\/bin\/ksh/) {
        $_[6] = "/bin/bash\n" ;
    }
    $line = join(":", @_) ;
    print FILEOUT $line;
}
```

15.19

```
#!/usr/bin/perl
open(FILEIN, "/etc/passwd");
@passwd_arr = <FILEIN> ;     # Read file into array
print("Enter a string to look up: ") ;
chop($input = <STDIN>) ;
@found_arr = grep (/$input/, @passwd_arr) ;  # Search at beginning

if ($#found_arr == -1 || $input eq "") {     # -1 means null array
   print("String does not exist\n") ;
   exit ;                                    # Go to beginning of loop
} else {
   print @found_arr;
}
```

15.20

```
#!/usr/bin/perl
$delim = $ARGV[0];
open (INFILE, "$ARGV[1]");
while (<INFILE>) {
    chomp;
    @arr = split(/$delim/);
    @bar = reverse(@arr);
    $line = join($delim, @bar);
    print "$line\n";
}
```

15.21 All existing files will be overwritten:

```
#!/usr/bin/perl
$file = $ARGV[0];
open (INFILE, "$file");

$i = 1;
while (<INFILE>) {
    $lines = $. % 10 ;
    if ($lines == 1) {
        $sfile = "$file.$i";
        $i++;
        open(FILEOUT, ">$sfile");
    }
    print FILEOUT $_;
    if ($lines == 0 ) {        # Written 10 lines already
        close(FILEOUT);        # So close the file
    }
}
```

15.22 Use both the **find** and **sort** commands in a **perl** script:

```
#!/usr/bin/perl
open(FILEOUT, "| sort -k2 -n");
foreach $file (`find . -type f -print`) {
   chop($file);
   $m_age = -M $file;
```

```
    print FILEOUT "$file $m_age\n";
}
```

15.23 The stamp data is assumed to be in the file `want_list`:

```
#!/usr/bin/perl
open(FILEIN, "want_list");
while (<FILEIN>) {
    chop;
    split;                                # Splits on @_ array
    $last_index = $#_;
    $last_field = $_[$last_index];

    # Extract the number from the last field
    $price = substr($last_field,1);
    $tot+= $price ;

    # Start a loop to progressively build the string for the description
    $field = 3;                  # From the fourth field
    $description = "";
    while ( $field <= $last_index - 1 ) {
        $description .= " " . $_[$field];
        $field++ ;
    }
    printf("%-12s %-7s %4d %-36s \$%.2f\n",
            $_[0], $_[1], $_[2], $description, $price);
}
print "TOTAL PRICE: $tot\n";
```

Chapter 16

16.1 An executable is a standalone program which contains the object code of all functions used by the program. Associated object files don't contain this code.

16.2 Wrong, both source and object files must be preserved for program maintenance to be possible, The **`make`** utility compares the last modification time of these two files to determine whether the source needs to be compiled or not.

16.3 The `-c` option creates only the object files, the `-o` option allows specification of the executable filename. The `-l` option is required to specify the name of the library file. The `-g` option enables the program to be debugged using a debugger tool.

16.4

```
# Rule 1
rec_deposit: rec_deposit.o compute.o arg_check.o quit.o
    cc -o rec_deposit rec_deposit.o compute.o arg_check.o quit.o -lm

# Rule 2
rec_deposit.o: rec_deposit.c rec_deposit.h prototype.h

# Rule 3
quit.o: quit.c quit.h

# Rule 4
arg_check.o: arg_check.c arg_check.h

# Rule 5
compute.o: compute.c compute.h
```

16.5 Use **`make -n`**.

16.6 **make** assumes that the program `foo.c` exists and runs the command **`cc -o foo foo.c`** if it finds that `foo` either doesn't exist or is older than `foo.c`.

16.7 When the dependency is not specified as below:

```
quit.o:
```

make assumes that the base source filename (`quit`) is the same as the object file. That is, **make** understands `quit.o` as the object file of `quit.c`. Also, when **make** is used with an argument (`clean`), it matches the argument with a target:

```
clean:
     rm *.o
```

and then runs the command (**`rm *.o`**) associated with the target.

16.8 (i) `ar -rv foobar.a foo[12].o` (ii) `ar -dv foobar.a foo2.o`

This archive can't be used with **`cc -l`** because its name doesn't have the `lib` prefix.

16.9 The object file, `a.o`, is available in the archive `foo.a`. **make** will run **`cc -o a a.c`** if `a.h` is modified and then invoke **ar** to replace the object file, `a.o`, in the archive.

16.10 SCCS stores the first version in full but RCS stores the latest version. Both store the differences between two consecutive versions. RCS is generally faster because later versions are more often required than earlier versions.

16.11 (i) `mv foo1.c foo.c ; admin -i foo.c s.foo.c`

(ii) `get -e s.quit.c ; mv foo2.c foo.c ; delta s.foo.c`

(iii) `get -e s.quit.c ; mv foo3.c foo.c ; delta s.foo.c`

16.12 The first 3 deltas are produced automatically by repeatedly invoking **`get -r s.foo.c`**, making some changes to the checked-out version and then running **`delta s.quit.c`** to check in the delta. To create the branch delta, use **`get -e -r1.2 s.quit.c`** and then check it in normally with **delta** after making changes.

16.13 **sact** obtains state information from a lock file. The lock file is created when an editable version of a file is checked out. The lock file is deleted when the delta is checked in.

16.14 Use **`unget s.quit.c`** to reverse the action of **get** because SCCS keeps track of all checked-out versions. If that is not done, the same version can't be checked out for editing.

Chapter 17

Use system calls wherever possible. However, you may use **printf**, **perror**, **strerror** and the directory handling library functions.

17.1

```
#include <stdio.h>
int main(void) {
    int i;
    extern int sys_nerr;              /* Total number of error messages */

    for (i =0; i < sys_nerr; i++)
        printf("%d: %s\n", i, strerror(i));
    printf("Number of errors available: %d\n", sys_nerr);
    exit(0);
}
```

17.2 An atomic operation comprises a group of actions that are either performed completely or not at all. (i) `fd = open("foo", O_WRONLY | O_TRUNC, 0644)` (ii) `fd = open("foo", O_WRONLY | O_CREAT, 0644)`

When **open** is used to create a file but only if it doesn't exist, **open** performs the check for existence of the file and its subsequent creation as an atomic operation. **creat** can't check for the existence of a file; the file is truncated if it exists.

17.3 This program assumes that the destination exists if it is a directory but the program won't overwrite an existing file. We'll use **lstat** instead of **stat**, and the `S_ISDIR` macro:

```
#define BUFSIZE 8192
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char **argv) {
    struct stat statbuf;
    int fd1, fd2, n;
    char buf[BUFSIZE]; char *fileout;

    if (argc != 3) {
       printf("Two arguments required\n") ;
       exit(1);
    }
    if ((fd1 = open(argv[1], O_RDONLY)) == -1) {
       printf("File not readable or doesn't exist\n") ;
       exit(2);
    }
    if (lstat(argv[2], &statbuf) == 0) {
       if (S_ISDIR(statbuf.st_mode)) { /* If destination is a directory */
          chdir(argv[2]);
          fileout = argv[1];           /* File copied to directory */
       }                               /* with same name */
    } else
       fileout = argv[2];

    if ((fd2=open(fileout, O_WRONLY | O_CREAT | O_TRUNC, 0644)) == -1) {
       printf("Error in opening destination file\n") ;
       exit(4);
    }
    while ((n = read(fd1, buf, BUFSIZE)) > 0)
       write(fd2, buf, n);
    exit(0);
}
```

17.4 This program has to be linked before it can be used:

```
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv) {
    int fd, n;
    char u;

    fd = open(argv[1], O_RDONLY) ;
    while ((n = read(fd, &u, 1)) > 0) {   /* Single-character buffer */
       if (strcmp(argv[0], "upper") == 0) {
         if (u >=97 && u <= 122)      /* Testing for lowercase letter */
             u -= 32;                 /* and changing it to uppercase */
       }
       else if (strcmp(argv[0], "lower") == 0) {
         if (u >=64 && u <= 99)       /* Testing for uppercase letter */
             u += 32;                 /* and changing it to lowercase */
       }
       write(STDOUT_FILENO, &u, 1);
    }
    close(fd);
    exit(0);
}
```

The **a.out** executable has to be linked to **upper** and **lower** using **ln**. The program can then be invoked by either **upper** or **lower**.

17.5 The kernel maintains a pool of buffers for I/O operations, and **read** and **write** access these buffers. Since system calls have significant overheads, the lesser the number of calls, the more efficient programs are. This efficiency is achieved by reading or writing in one **read** or **write** call as many characters as the size of the kernel buffer itself.

17.6 (i)

```
/* Program using a single-character buffer */
#include <fcntl.h>
#include <unistd.h>

int main(void) {
    int fd, n;
    char buf;

    if ((fd = open("/etc/passwd", O_RDONLY)) == -1) {
       perror("open");
       exit(1);
    }
    while ((n = read(fd, &buf, 1)) > 0) {
        write(STDOUT_FILENO, &buf, 1);
    }
    exit(0);
}
```

(ii)

```
/* Program using a multi-character buffer */
#include <fcntl.h>
#include <unistd.h>
#define BUFSIZE 2048

int main(void) {
    int fd, n;
    char buf[BUFSIZE];

    if ((fd = open("/etc/passwd", O_RDONLY)) == -1) {
       perror("open");
       exit(1);
    }
    while ((n = read(fd, buf, BUFSIZE)) > 0) {
        write(STDOUT_FILENO, buf, n);
    }
    exit(0);
}
```

17.7 Save the old value before printing the new one:

```
#include <stdio.h>
#include <fcntl.h>

int main(void) {
   mode_t old_mode;
   old_mode = umask(0);                          /* No mask */
   printf("Current umask value: %o\n", old_mode);
   umask(old_mode);                              /* Revert to previous mask */
   exit(0);
}
```

17.8 The code skeleton looks like this:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main(void) {
    int fd1, fd2;

    if (access(".lockfile", F_OK) == 0) {
        printf("Lockfile exists, quitting...\n");
        exit(1);
    } else                    /* Create the lockfile */
        fd1 = open(".lockfile", O_WRONLY | O_CREAT, 0644);

    if ((fd2 = open("foo", O_RDONLY)) == -1) {
        perror("open");
        unlink(".lockfile");
        exit(1);
    }
     /* You may now read this file or do
          something else */
    close(fd1);
    close(fd2);
    unlink(".lockfile");
    exit(0);
}
```

17.9 No splitting is obviously required if the file size doesn't exceed 10,000 bytes:

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>

int main(int argc, char **argv) {
    struct stat statbuf;
    char buf[10000];              /* For each segment of 10,000 bytes */
    char outfile[256];            /* For the output filename */
    char ext[100];                /* For the extension, .1, .2, etc */
    off_t size;                   /* To check file size */
    int n, fd1, fd2;
    int fileno = 1;
    char *infile = argv[1];

    /* First check whether file needs any splitting at all */
    lstat(infile, &statbuf);
    size = statbuf.st_size;
    if (size <= 10000) {
        printf("File size is small; not split\n");
        exit(0);
    }

    fd1 = open(infile, O_RDONLY);

    while (1) {
        /* First build the destination filename */
        strcpy(outfile, infile);
        strcat(outfile, ".");
        sprintf(ext, "%d\0", fileno);
        strcat(outfile, ext);             /* Done */
```

```
        /* Now open the destination and write 10,000 characters to it */

        fd2 = open(outfile, O_WRONLY | O_CREAT | O_TRUNC, 0644);

        while ((n = read(fd1, buf, 10000)) > 0) {
            write(fd2, buf, n);
            close(fd2);
            if (n != 10000) {       /* Reached the end */
                close(fd1);
                exit(0);
            } else {
                fileno++ ;
                break;        /* Switch to the next filename */
            }
        }
    }
}
```

17.10

```
#include <fcntl.h>
#include <unistd.h>
#define BUFSIZE 2048

int main(void) {
    int fd1, fd2, n;
    char buf[BUFSIZE];

    if ((fd1 = open("foo", O_RDONLY)) == -1) {
       perror("1st open");
       exit(1);
    }
    if (unlink("foo") == -1) {
       perror("unlink");
       exit(2);
    }
    while ((n = read(fd1, buf, BUFSIZE)) > 0)
        write(STDOUT_FILENO, buf, n);
    if ((fd2 = open("foo", O_RDONLY)) == -1) {
       perror("2nd open");
       exit(1);
    }
}
```

unlink removed the directory entry for the file, but the inode was not freed because the file was still open. The file could still be read using the descriptor `fd1`. But a second open on the same file failed because the file has no directory entry. The **exit** function (or program termination) closed the file and freed its inode.

17.11 The `S_ISDIR` and `S_ISREG` macros come in handy here:

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#define BUFSIZE 8192

int main(int argc, char **argv) {
    char *destination = argv[argc-1];
    int fd1, fd2, m, n;
    char buf[BUFSIZE];
    struct stat statbuf, statbuf2;
    char destination_pathname[256];
```

```
    if (access(destination, F_OK) == 0) {      /* If destination exists */
        lstat(destination, &statbuf);
        if (!S_ISDIR(statbuf.st_mode)) {       /* and is not a directory */
            printf("File is not a directory; creating a directory\n");
            unlink(destination);               /* remove the file */
            mkdir (destination, 0755);         /* and create the directory */
        }
    }
    else {                                     /* Destination doesn't exist */
        mkdir(destination, 0755);              /* so create it */
        printf("Directory created\n");
    }

    for (m=1; m < argc-1; m++) {              /* For each source file */
        lstat(argv[m], &statbuf2);
        if (!S_ISREG(statbuf2.st_mode)) {     /* that is not a regular file */
            printf("%s: Not an ordinary file\n", argv[m]);
            continue;                         /* ignore */
        }

        /* Create the pathname for the destination filename */
        strcpy(destination_pathname, destination);
        strcat(destination_pathname, "/");
        strcat(destination_pathname, argv[m]);

        /* Finally copy each file */
        fd1 = open(argv[m], O_RDONLY);
        fd2 = open(destination_pathname, O_WRONLY | O_CREAT | O_TRUNC, 0644);
        while ((n = read(fd1, buf, BUFSIZE)) > 0)
            write(fd2, buf, n);
    }
    exit(0);
}
```

17.12

```
#include <fcntl.h>
#include <sys/stat.h>

int main(void) {
    int fd;
    mode_t old_mode, file_perm;
    struct stat statbuf;

    old_mode = umask(0);
    if ((fd = open("foo", O_WRONLY | O_CREAT,
                   S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1) {
       perror("open");
       exit(1);
    }
    close(fd);
    system("ls -l foo");

    if (lstat("foo", &statbuf) == -1) {
       perror("lstat");
       exit(1);
    }
    chmod("foo", statbuf.st_mode | S_IWGRP);  /* Inode updated so */
    system("ls -l foo");
```

```
     if (lstat("foo", &statbuf) == -1) {       /* we need to stat again */
        perror("lstat");
        exit(1);
     }
     chmod("foo", (statbuf.st_mode & ~S_IROTH));
     system("ls -l foo");
     unlink("foo");
}
```

17.13 In any case, you can change the permissions of a file only if you are the owner:

```
#include <dirent.h>

int main(void) {
   DIR *dir;
   struct dirent *direntry; struct stat statbuf;

   /* Using library functions, so need to check for NULL values */
   if ((dir = opendir(".")) != NULL)
       while ((direntry = readdir(dir)) != NULL)
          if (lstat(direntry->d_name, &statbuf) == 0)
              if (getuid() == statbuf.st_uid)
                  chmod(direntry->d_name, statbuf.st_mode & ~S_IRWXO);
}
```

17.14

```
#include <sys/stat.h>
#include <fcntl.h>
#include <utime.h>

int main(int argc, char **argv) {
     struct stat statbuf;
     struct utimbuf timebuf;
     mode_t old_mask, perm;

     lstat(argv[1], &statbuf);

     /* Get permissions and time stamps of existing file */
     perm = statbuf.st_mode & ~S_IFMT ;
     timebuf.actime  = statbuf.st_atime;
     timebuf.modtime = statbuf.st_mtime;

     /* Now create the file with these attributes */
     old_mask = umask(0);
     open(argv[2], O_RDONLY | O_CREAT, perm); /* Permissions set */
     utime(argv[2], &timebuf);                /* Time stamps set */
     umask(old_mask);                         /* Restore previous mask */
}
```

17.15 The program, **`print_permissions.c`**, does the job specified in (i):

```
#include <sys/stat.h>
#include <unistd.h>

void print_permissions(char *fname, struct stat *addbuf) {
   if (stat(fname, addbuf) < 0) {
       perror("stat");
       exit(1);
   } else
       printf("File: %s  Permissions: %o\n", fname, addbuf->st_mode & ~S_IFMT);
}
```

The program, **check_permissions.c**, does the job specified in (ii):

```
#include <stdio.h>

void check_permissions(int perm, int flag, char *message) {
   if (perm & flag)
       printf("%s\n", message);
}
```

The main program, **check_all_perm.c** uses both functions:

```
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>

/* Function prototypes */
void print_permissions(char *, struct stat *) ;
void check_permissions(int, int, char *);

int main(int argc, char *argv[]) {
    int i, perm;
    char *filename = argv[1] ;
    struct stat statbuf;
    mode_t perm_flag[] = {S_IRUSR, S_IWUSR, S_IXUSR,
                          S_IRGRP, S_IWGRP, S_IXGRP,
                          S_IROTH, S_IWOTH, S_IXOTH,
                          S_ISUID, S_ISGID, S_ISVTX };
    char *mesg[] = { "User-readable", "User-writable", "User-executable",
                     "Group-readable", "Group-writable", "Group-executable",
                     "Others-readable", "Others-writable", "Others-executable",
                     "SUID bit set", "SGID bit set", "Sticky bit set" };

    print_permissions(filename, &statbuf);

    perm = statbuf.st_mode & ~S_IFMT;
    for (i = 0; i < 12; i ++)
        check_permissions(perm, perm_flag[i], mesg[i]);
    exit(0);
}
```

Finally compile this program with this command:

```
cc check_all_perm.c check_permissions.c print_permissions.c
```

Chapter 18

18.1 The stack stores the local variables and arguments of a function while the heap memory is used when a process needs to create memory dynamically. Since the kernel may be concurrently executing on behalf of several processes, it needs a separate stack for each process.

18.2 The first column shows the size of the text segment. The second column shows the size of the initialized data segment that holds constants and global variables. The third column shows the size of the uninitialized data segment.

18.3 The MMU contains a set of hardware registers that point to the translation maps of the currently running process. When the process changes, the MMU loads the registers with the translation maps of the next process. Because every process has its own set of translation maps, one process can't access another's address space.

18.4 The environ array stores all environment variables:

```
#include <stdio.h>

int main(void) {
    extern char **environ ;
    int i = 0;

    while ((char*)environ[i] != '\0') {
       printf("%s\n", environ[i]);
       i++;
    }
    exit(0);
}
```

18.5 The parent sleeps for 2 seconds to ensure that the child dies before it does:

```
#include <stdio.h>
#include <sys/types.h>
#define PATH_LENGTH 30

int main (void) {
   pid_t pid;
   int x = 100;
   char newdir[PATH_LENGTH + 1];    /* Additional space required for \0 */

   getcwd(newdir, PATH_LENGTH);    /*Get current directory before fork */
   printf ("BEFORE FORK -- Current directory: %s\n", newdir);

   pid = fork ();
   switch (pid) {
      case -1:
         perror("fork");                              /* fork error */
         exit(1);                                     /* Parent exits */

      case 0:   /* Child*/
         printf("CHILD -- Inherited value of x: %d\n", x);
         x = 200 ;                                   /* Change x in child */
         printf("CHILD -- Changed value of x: %d\n", x);
         printf("CHILD -- Inherited value of PATH: %s\n", getenv("PATH"));
         setenv("PATH", ".", 1);  /* Change PATH here; use putenv("PATH=.") */
                                  /* if setenv() not supported*/
         printf ("CHILD -- New value of PATH: %s\n", getenv("PATH"));
         if (chdir("/etc") != -1) {               /* "cd" to /etc */
             getcwd(newdir, PATH_LENGTH);         /* Do a "pwd" */
             printf("CHILD -- Current directory changed to: %s\n", newdir);
         }
         break;

      default:   /* Parent */
         sleep(2);                              /* Allow child to complete */
         getcwd(newdir, PATH_LENGTH);           /*Getting new directory*/
         printf("PARENT -- Value of x is unchanged: %d\n", x);
         printf("PARENT -- Value of PATH is unchanged: %s\n", getenv("PATH"));
         printf("PARENT -- Current directory is still: %s\n", newdir);
   }
   exit(0);
}
```

18.6 The statement `Before fork` occurs twice in the file. I/O buffers are copied during a fork and the standard library functions like **`printf`** are fully buffered when writing to a file. That's why the `printf` statement in the parent was not written to the terminal when the output was redirected.

18.7 First define a shell script named **`environment.sh`** like this:

```
#!/bin/sh
echo "The value of HOME is $HOME"
echo "The value of PATH is $PATH"
echo "The value of MAIL is $MAIL"
echo "The value of TERM is $TERM"
```

Next use **`execve`** in this program to run **`environment.sh`** with an environment that includes only `PATH` and `HOME`:

```
#include <stdio.h>
int main (int argc, char** argv) {
    char *cmdargs[] = { "environment", (char *) 0 };
    char *env[] = {"PATH=/bin:.", "HOME=/home/romeo", (char *) 0};
    execve("environment.sh", cmdargs, env);
    printf ("execve error\n");
}
```

The output will show null values of `MAIL` and `TERM`.

18.8. (i) Changes only with fork. (ii) Changes only with fork. (iii) Copied in fork and preserved in exec unless they are closed by exec. (iv) Copied only in fork.

18.9 True, the process is removed from the process table only after the parent has picked up the exit status. Until that happens, the process remains in the zombie state.

18.10 Because the parent dies before the child, **`init`** takes over the parentage of the child:

```
#include <stdio.h>

int main(void) {
   int pid;

   if ((pid = fork()) > 0)       /* Parent */
      exit(10);                  /* Parent exits without calling wait */
   else if (pid == 0) {          /* Child */
      sleep(2);                  /* Lets parent die in this time frame */
      printf("CHILD: Adopted by init now, PPID: %d\n", getppid());
      exit(0);
   }
}
```

18.11 Because there are some attributes of an open file that need to be replicated during a fork, and some that must not be. For a parent and child to be able to write the same file without conflict, the descriptor needs to be replicated but not the offset pointer. This is possible only if these parameters are held in two separate tables.

18.12 Each file descriptor is associated with a separate file table entry except when **`dup`**, **`dup2`** and **`fork`** are invoked.

18.13 The descriptor table now contains two entries that point to (i) separate file tables. (ii) same file table. A descriptor can be replicated by **`fork`**, **`dup`**, **`dup2`** and **`fcntl`**.

18.14 Since only eight bits are used to store the exit status, the maximum value is restricted to 255:

```
#include <stdio.h>
#include <sys/wait.h>

int main (int argc, char **argv) {
   int a, b, c, status;
```

```
    switch(fork()) {
       case 0:
           switch(fork()) {
              case 0:
                 a = atoi(argv[1]);    /* Convert string to integer */
                 b = atoi(argv[2]);
                 c = a + b ;
                 exit(c);              /* Parent to pick up the sum */
              default:
                 wait(&status);
                 printf("The sum of the two numbers is %d\n",
                                 WEXITSTATUS(status));
                 exit(WEXITSTATUS(status*2));
           }

      default:
           wait(&status);
           printf("The sum after doubling is %d\n", WEXITSTATUS(status));
           exit(0);
    }
}
```

18.15 **waitpid** (i) need not block until a child dies. (ii) can wait for a child with a specific PID to die. (iii) can also handle process groups. The call **waipid(-1, &status, 0)** emulates **wait(&status)**.

18.16 If the parent doesn't wait for the death of the child, the child moves to a zombie state and retains its slot in the process table. The child is otherwise dead. A process turns to an orphan when its parent dies, in which case **init** takes over the parentage.

18.17 This program doesn't run the shell's internal commands. It uses the **memset** library function to initialize memory used by the buffers:

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>                  /* For strtok */
#include <wait.h>

#define BUFSIZE 200                  /* Maximum size of command line */
#define ARGVSIZE 40                  /* Maximum number of arguments */
#define DELIM "\n\t\r "              /* White-space delimiters for strtok */

int main (int argc, char **argv) {
    int i, n ;
    char buf[BUFSIZE+1];             /* Stores the entered command line */
    char *clargs[ARGVSIZE];          /* Stores the argument strings */
    int returnval ;                  /* Used by wait */

    for (;;) {                       /* Loop forever */
        /* First initialize the two buffers */
        memset(clargs, 0, ARGVSIZE);  /* Initialize argument array */
        memset(buf, 0, BUFSIZE);      /* and command buffer */

        write(STDOUT_FILENO, "Shell> ", 7);  /* Display a prompt */
        read(STDIN_FILENO, buf, BUFSIZE);    /* Read user input into buf */
        if (!strcmp(buf, "exit\n"))
            exit(0);                      /* Terminate if user enters exit */

        /* Split the command line into an array of pointers */
        clargs[0] = strtok(buf, DELIM);   /* first word */
        n = 1;
```

```
        while ((clargs[n] = strtok(NULL, DELIM)) != NULL)
            n++;                            /* ... all words are extracted */
        clargs[n] = NULL;               /* Set last argument pointer to NULL */

        switch(fork()) {
           case 0:                      /* Run command in child */
                if ((execvp(clargs[0], &clargs[0])) < 0)
                    perror(clargs[0]);
          default:                      /* In the parent */
                wait(&returnval);
                printf("Exit status of command: %d\n", WEXITSTATUS(returnval));
        }
    }
}
```

18.18 The trick here is to run **`sh -c`** with **`execl`**. The program runs the shell's internal commands and also handles metacharacters like wild cards, redirection and piping:

```
#include <stdio.h>
#include <unistd.h>
#include <wait.h>
#define BUFSIZE 200

int main (int argc, char **argv) {
    int i;
    char buf[BUFSIZE];
    int returnval ;

    for (;;) {
        memset(buf, 0, BUFSIZE);                /* Initialize buffer */
        write(STDOUT_FILENO, "Command> ", 9);
        read(STDIN_FILENO, buf, BUFSIZE);
        if (!strcmp(buf, "exit\n"))
            exit(0);

        switch(fork()) {
           case 0:
                if ((execl("/bin/sh", "sh", "-c", buf, NULL)) < 0) {
                    perror("exec");
                    exit(1);
                }
          default:
                wait(&returnval);
                fprintf(stderr, "Exit status of command: %d\n",
                                WEXITSTATUS(returnval));
        }
    }
}
```

18.19 A process can either ignore a signal, perform the default action or catch it to execute a signal handling function. The `SIGSTOP` and `SIGKILL` signals can't be either ignored or caught and will always perform the default action.

18.20 **`vi`** attempts to read from standard input. When a background process tries to do that, the kernel issues a `SIGTTIN` signal whose default action is to suspend the process. Unless you change the default disposition, you can't input text from the keyboard to a background process.

Chapter 19

19.1 **su** runs a shell with ownership set to root, and hence has to be terminated with **exit**. **su - romeo** runs a shell with romeo as its owner and whose environment is created as if romeo has logged in.

19.2 (i) Set the system date. (ii) Address all users with **wall**. (iii) Kill any process owned by any user. (iv) Change any user's password. (v) Set the hostname.

19.3 The command **passwd -x 28** *username* forces a user to change a password every four weeks.

19.4 Create a user with the **useradd** command and then manually change the UID in `/etc/passwd` to 0.

19.5 Use **useradd -u 212 -g dialout -d /home/john -s /bin/bash -m john**. To change the shell to Korn, use **usermod -s /bin/ksh john**.

19.6 The fourth field in `/etc/group` contains only the usernames who also belong to supplementary groups. romeo's primary GID is stored in `/etc/passwd` and this relationship is not maintained in `/etc/group`.

19.7 A restricted shell can't (i) use **cd**. (ii) reset `PATH`. (iii) reset `SHELL`. (iv) use pathnames containing a `/`. (v) use the > and >> symbols.

19.8 The **passwd** command is owned by root and has the SUID bit set, which means that its process will have the effective UID set to root and not of the user invoking the program. This privilege allows the process to update `/etc/shadow`.

19.9 (i) The sticky bit must be set for the directory. (ii) The users must belong to a common group and the directory must be owned by the group. (iii) There must be write permission for that group on that directory.

19.10 **init** initiates all system daemons by running the startup scripts. It also fork-execs a **getty** at every active terminal, and **getty** execs the **login** program when a user attempts to log in. On successful login, **login** execs a shell which now has **init** as its parent.

19.11 The default run level is set in the line containing `initdefault` in `/etc/inittab`.

19.12 When **init** enters a run level, it first executes the scripts beginning with `K` to kill processes that shouldn't be running at that level. **init** then executes the start scripts beginning with `S` to start processes that should be running at that level. On System V, these scripts are placed in `/etc/rc`*n*`.d`. On Linux systems, they are in `/etc/rc.d/rc`*n*`.d`.

19.13 Look for either `lpd` or `lpsched` in the **ps** output:

```
if ps -e | grep lpsched >/dev/null ; then
    echo "lpsched is running"
elif ps -aux | grep lpd >/dev/null ; then
    echo "lpd is running"
else
    echo "Printer daemon is not running"
fi
```

19.14 Starting from root, **find** locates (i) all files having the SUID bit set. (ii) all directories with the sticky bit set and also displays their listing. (iii) all ordinary files with size more than 2048 blocks that have not been modified in 365 days.

19.15 Multiple file systems make system administration work easier. Corruption in one file system doesn't affect another. Also, individual file systems can be backed up separately.

19.16 Mounting is the attachment of one file system to a specific directory of another file system. Unmounting is not possible when the user executing **umount** is placed in a directory of the file system.

19.17 The *ufs* and *ext2* file systems have multiple copies of the superblock distributed throughout the disk. If the main superblock is corrupt, the system can be directed to use another.

19.18 **sync** periodically updates the disk with the memory contents of the superblock and inodes. It schedules a writing operation but the write doesn't take place immediately. When **fsck** makes changes to the file system on disk, the disk copy becomes more recent than the memory copy. **sync** should not be used then.

19.19 The *ufs* file system has a variable block size, supports 255-character filenames, disk quotas and symbolic links. *proc* is a pseudo-file system that enables us to know process attributes, where each process is represented as a separate file in the system.

19.20 This needs the **dd** command:

```
echo "Insert source diskette in drive A and press the [Enter] key: \c"
dd if=/dev/rfd0 of=$$ bs=1474560
echo "Insert target diskette in drive A and press the [Enter] key: \c"
dd if=$$ of=/dev/rfd0135ds18 bs=1474560 ; rm $$
```

19.21 (i) Not possible. (ii) Not possible. (iii) `tar -rvf /dev/fd0 *`

19.22 First create a file immediately on login with **`touch .last_inc_backup_time`**. At the end of the day, run this command:

```
tar -cvf /dev/fd0 `find $HOME -newer .last_inc_backup_time -print`
```