1429
UNOS

Bell Laboratories

subject: Division 56 UNIX™ Course -
Charging Cases 49460-325, -326
Filing Case 40288-700

date: November 20, 1979

from: K. E. Wendland
5613-791120.01EN

ABSTRACT

Recently, Division 56 held an "in-house" UNIX™ course. The class notes reproduced here cover topics from "How to obtain a UNIX user ID" and login/logout procedures, to a thorough discussion of the context editor, the file and directory structure of UNIX, on-line/off-line printing, background processes, "shell" properties, and many useful UNIX commands.

These notes are meant to help the new user to make effective use of the UNIX/TS facilities available; they are written in a tutorial format, with many examples. They are also thoroughly indexed, which makes them a good reference for the experienced user.

IH-5613-KEW-bgp

K. E. Wendland

Abstract Only to
All Supervision Division 56
All Supervision Division 36
All Department Heads Centers 373, 376, 377

Complete Memorandum to
M. Blum
K. I. Bogard
S. L. Clarey
W. A. Depp
T. A. Dolotta
P. Johnson
P. L. Kitterman
D. A. Lautenbach
R. A. Meacham, Jr.
L. A. Nelsen
J. M. Scanlon
J. H. Shoemake
C. Tartanella
K. A. Ulven

CONTENTS

1. INTRODUCTORY COMMENTS ABOUT UNIX..... 1
1.1 What is UNIX ??..... 1
1.2 Obtaining a UNIX User ID..... 1
1.3 Login Procedures..... 1
1.4 Logoff Procedures..... 2
1.5 Typing Errors..... 2
Erase Character (#):..... 2
Kill Line Character (@):..... 3
1.6 Readahead Capabilities..... 3
1.7 Stopping Terminal Output..... 3
1.8 Command Prompt..... 3
1.9 File Description..... 4
1.10 References..... 4

2. CONTEXT EDITOR..... 5
2.1 Initial Entry..... 5
EDITOR command (ed):..... 5
APPEND command (a):..... 5
2.2 Writing onto a File..... 5
WRITE command (w):..... 5
2.3 Leaving the Editor..... 6
QUIT command (q):..... 6
2.4 Editing Existing Files..... 6
EDIT command (e):..... 6
WRITE command (w) revisited:..... 6
FILENAME command (f):..... 7
APPEND command (a) revisited:..... 7
Repeated EDIT (e) commands:..... 7
READ command (r):..... 8
2.5 Printing Text..... 8
PRINT command (p):..... 8
End of buffer symbol ($):..... 9
CEASE PRINTING command:..... 9
Current line symbol (.):..... 9
PRINT LINE NUMBER symbol (=):..... 10
Relative line number addressing:..... 10
Special meaning of , and ; as line numbers:..... 12
LIST command (l):..... 12
2.6 Deleting and Adding Lines..... 12
WRITE command (w) revisited:..... 12
READ command (r) revisited:..... 12
APPEND command (a) revisited:..... 13
INSERT command (i):..... 13
DELETE command (d):..... 13
CHANGE command (c):..... 14
MOVE command (m):..... 14
COPY command (t):..... 14
2.7 Text Searches and Special Characters..... 15
FORWARD SEARCH command (/.../):..... 15
REVERSE SEARCHES command (?...?):..... 16

Context addressing:..... 16
SEMICOLON separator (;):..... 16
Metacharacters in regular expressions:..... 17
BACKSLASH metacharacter (\):..... 17
PERIOD metacharacter (.):..... 18
DOLLAR SIGN metacharacter ($):..... 18
CIRCUMFLEX metacharacter (^):..... 19
ASTERISK metacharacter (*):..... 19
BACKSLASH/BRACE metacharacters:..... 19
SQUARE BRACKETS metacharacter ([...]):..... 20
2.8 Modifying Existing Test..... 21
SUBSTITUTE command (s):..... 21
Metacharacters in regular expressions revisited:..... 21
Multiline substitutions:..... 22
GLOBAL suffix command (g):..... 22
Asterisk metacharacter anomalies:..... 22
Breaking up a line:..... 23
Removing strings from a line:..... 23
Removing non-printable characters:..... 23
Metacharacters in replacement test:..... 24
AMPERSAND metacharacter (&):..... 24
Repeated regular expressions:..... 24
Repeated replacement text:..... 25
UNDC command (u):..... 25
2.9 Global Changes..... 25
GLOBAL command (g):..... 25
EXCLUDE command (v):..... 26
Multiline global commands:..... 26
UNDC command (u) revisited:..... 27
INTERACTIVE GLOBAL command (G):..... 27
INTERACTIVE EXCLUDE command (V):..... 27
2.10 Temporary Escape from the Editor..... 27
ESCAPE command (!):..... 27
2.11 Miscellaneous Editor Commands..... 28
EDITOR PROMPT command (P):..... 28
HELP commands (h and H):..... 28
The EDIT and QUIT commands (E and Q) revisited:..... 28
JOIN command (j):..... 28
MARK command (k):..... 28
Delimiter replacement in s-command:..... 29

3. BASIC FILE COMMANDS..... 30
3.1 Listing Files and the File Tree Structure..... 30
LIST command (ls):..... 30
Directories and the File Tree Structure:..... 31
LIST command (ls) revisited:..... 31
Pathnames:..... 32
LIST command (ls) again revisited:..... 33
3.2 File Protection..... 34
CHANGE MODE command (chmod):..... 34
Octal mode:..... 35

Symbolic mode:..... 35
Changing directory permissions:..... 36
3.3 Removing, Linking, Copying or Changing File Names..... 37
REMOVE command (rm):..... 37
REMOVE DIRECTORY CONTENTS command (rm -r):..... 37
DELETE INTERACTIVELY command (rm -ir):..... 38
MOVE or RENAME command (mv):..... 38
COPY command (cp):..... 39
LINK command (ln):..... 40
3.4 Printing File Contents..... 40
CONCATENATE command (cat):..... 40
PRINT command (pr):..... 41
OFFLINE PRINT command (opr):..... 41
3.5 Special Characters associated with Filenames..... 42
QUESTION MARK metacharacter (?):..... 42
SQUARE BRACKETS metacharacter ([...]):..... 42
ASTERISK metacharacter (*):..... 42
3.6 Directory Manipulations..... 43
CHANGE DIRECTORY command (cd):..... 43
Special meaning of . and ..:..... 44
PRINT WORKING DIRECTORY command (pwd):..... 44
MAKE DIRECTORY command (mkdir):..... 44
REMOVE DIRECTORY command (rmdir):..... 45
3.7 Redirection of Inputs and Outputs..... 45
OUTPUT REDIRECTION symbol (>):..... 45
APPEND REDIRECTION symbol (>>):..... 45
INPUT REDIRECTION symbol (<):..... 46
Shell description:..... 46
PIPELINE symbol (|):..... 46
MULTIPLE COMMAND symbol (;):..... 47
3.8 Background Processes..... 47
BACKGROUND COMMAND symbol (&):..... 48
PROCESS STATUS command (ps):..... 48
KILL command:..... 48
Immunity to Hang-Ups:..... 48
Time-Delayed Commands:..... 49
3.9 Other Miscellaneous Commands..... 50
News reports:..... 50
Inter-user MAIL command:..... 51
Sending Files to Another UNIX System:..... 52
DATE command:..... 53
MANUAL command (man):..... 53
SET TERMINAL OPTIONS command (stty):..... 53
TAIL command:..... 54
SPELL and TYPO commands:..... 54
GET REGULAR EXPRESSION command (grep):..... 54
WORD COUNT command (wc):..... 55
PASSWORD command (passwd):..... 55
DIFFERENCE command (diff):..... 55
Special file .profile:..... 55
Other commands of interest:..... 56

4. HOMEWORK..... 58
4.1 LEARN Program..... 58
4.2 Final Exam..... 58
4.3 Examination Solutions..... 66

Bell Laboratories

subject: Division 56 UNIX™ Course-
Charging Cases 49460-325, -326
Filing Case 40288-700

date: November 20, 1979

from: K. E. Wendland
IH 5613
5C-412 x2068
5613-791120,01EN

ENGINEER'S NOTES

1. INTRODUCTORY COMMENTS ABOUT UNIX

1.1 What is UNIX ??

UNIX* is an operating system, with a hierarchical file structure, a powerful context editor, high level programming languages, and other useful features. To the user it is a general-purpose, multi-user, time-sharing, interactive computer system.

1.2 Obtaining a UNIX User ID

If you do not have a valid UNIX user ID, go to room 6M-402 (the current location of the UNIX administrator) and obtain a "COMPUTER SERVICES REQUEST" form. Complete the "User Data", "Charging Information" and "IH" sections, have your local administrator (probably your supervisor) complete the "Authorizer Data" section, and return to room 6M-402. Do not fill in the password section, as UNIX initially assumes your payroll account number is your password; the first time you enter the system, you will be prompted for a new personalized password. Also, go to room 6E-531 and obtain all available documentation on the PWB/UNIX time-sharing system.

1.3 Login Procedures

- Connect terminal modem to a green Dimension data phone and dial the local number for your UNIX system. Phone numbers are changed frequently, and there are many general purpose and project UNIX machines at Indian Hill --- check with your supervisor to determine the correct numbers to use. The terminal should be powered on, set to lower case (if possible), set to full duplex, and set to 30 characters per second (or high speed). Note: for access from outside BTL, you may dial a special Wheaton number for your UNIX machine on any telephone.

---

* UNIX is a trademark of Bell Laboratories

- Once you are logically connected to UNIX, an "on line" or "carrier" indicator light will be lit, and a printout will appear as:

  login:

  followed by a "beep". Type in your "user ID" (or "login name") followed by a carriage return (CR).

- The next request printed out is:

  Password:

  and UNIX expects your "personal password" followed by a CR, to be entered; printing of this password will be inhibited.

  Note: if you are calling to the outside Wheaton number, the computer will additionally request:

  External Security:

  to which you must respond with the current "external security code word" (printing inhibited) followed by a CR; this code word changes monthly and is available from your supervisor.

- After printing one or more lines of informative comments, a prompt

  $

  will be printed. UNIX is now ready to accept your commands.

1.4 Logoff Procedures

Once you are through with your UNIX session, you can simply break the phone connection to disengage the computer. Alternately, you may type "Control-D" (typing D while depressing the CTRL Key), which will result in a new LOGIN request; at this point you may login again or simply hang-up.

1.5 Typing Errors

Erase Character (#): To correct typing errors caught before a "carriage return", you can use

#

to erase the last character typed; repetitive use of "#" can erase any number of characters back to the beginning of the line, but not beyond. An example is:

```
my#anxx##
```

which will be interpreted by UNIX as "man" typed correctly. A common error is to "backspace" (simultaneously depressing the "CTRL" and "H" keys) over an unwanted "space"; use the "#" erase character instead, as both "space" and "backspace" are legal characters in UNIX. Also, it should be noted that "#" will work during the login interval.

Kill Line Character (@): The "at-sign"

```
@
```

will erase the entire line printed thus far, return you to the beginning of a new line, and allow you to retype that line from "scratch". Again, the "@" will work during the login interval.

1.6 Readahead Capabilities

Once the initial command prompt is received, UNIX allows full readahead, i.e., you may type at any speed regardless of what is printing on the terminal. It should be noted this may cause a strange intermixing of input/output characters to be printed, but UNIX will eventually interpret your typed commands correctly.

1.7 Stopping Terminal Output

At times, it may be convenient to stop the "printing" of terminal output temporarily (especially when you are trying to keep information from rolling off a CRT type terminal, before it can be read). This may be accomplished by typing "CONTROL-S" (simultaneously depressing the "CTRL" and "S" keys). To resume printing, simply type another "CONTROL-S" (or any other character).

To terminate (permanently) a command in progress, simply hit the "BREAK" or "DEL(ETE)" key.

1.8 Command Prompt

Your response to a "$" command prompt is to issue UNIX commands. These commands are numerous, and many will be discussed in later sections. For now, respond to your first "$" with

```
date
```

followed by a CR. At this stage, it should be obvious that all commands and responses, when complete, should be followed by a "carriage return" --- the CR will be implicit in the remainder of these notes. The result of the "date" command is a print-out of the current date and time.

1.9 File Description

A file is simply a collection of characters (data, text, programs, executable code, etc. ---- a maximum of several million characters) stored in UNIX. The simplest way to form a file is via the CONTEXT EDITOR, which will be considered next.

1.10 References

The following references might be of use (until revised, they are somewhat outdated by the introduction of the new UNIX/TS system):

"A Tutorial Introduction to the UNIX Text Editor" by Kernighan

"Advanced Editing in UNIX" by Kernighan

"UNIX for Beginners" by Kernighan

The following "bible" contains all available UNIX commands and their option descriptions, and is the most valuable reference source available. Use it only as a reference, as you will confuse yourself, if you attempt to read it as a tutorial.

"UNIX/TS User's Manual" by Dolotta, et.al.

All of the above documents are available from the UNIX library.

Also of interest is a program which can be used as a source of supplemental "homework" problems. To access this program, simply type "learn" in response to the "$" command prompt. You will be transferred to a self-explanatory, interactive course on various UNIX topics. More details of this special program are given in Chapter 4.

2. CONTEXT EDITOR

2.1 Initial Entry

EDITOR command (ed): The UNIX CONTEXT EDITOR allows you to get information stored into a file. Upon first entry type:

```
ed                [don't forget CR]
```

in response to a "$" prompt, and you are now in the editor.

APPEND command (a): To enter text, the APPEND command must be given, i.e., type:

```
a                 [again, don't forget CR]
```

followed by as many lines of text as you desire. Remember "#" and "@" can be used to correct typographical errors. When you have typed in the desired text, type a line with only a single "period" on it, i.e.,

```
.
```

The "period" must be in the first space followed immediately by a CR --- any variation will not allow you to leave the "append" mode and all additional lines typed will be added as garbage to your text, until you properly terminate the appending session as described. The sequence of commands:

```
a
I love telephones
.
```

will result in the line "I love telephones" being stored in the "editor buffer" (a temporary storage area). The initial "a" and final "." will not appear in the buffer, as they are editor commands, not text.

2.2 Writing onto a File

WRITE command (w): Once text is accumulated in the editor buffer, it is likely we would like to store it in an external (disk) file for later use. We can use the editor WRITE command as follows:

```
w file_name   [A "blank" after "w" is necessary]
```

A "file_name" is composed of up to 14 alphanumeric and/or special characters; a "blank", "?", "\", "/", "*", "[", or non-printable characters may/should not be used. The editor always responds with a count of characters written onto the file, if all goes well; the count includes "blank" and "carriage return" (or "new line") characters. It should be noted that writing onto a

file simply copies the contents of the editor buffer to the named file, leaving the contents of the buffer undisturbed. If writing onto an existing file, the original contents are destroyed by overwriting.

2.3 Leaving the Editor

QUIT command (q): To leave the editor, simply type the QUIT command

```
q
```

which will return you to the UNIX command level, indicated by a "$" prompt. At this point the contents of the editor buffer vanishes, which is why you should write it out onto a file before quitting. If you have modified the contents of the editor buffer in any way, and did not use the write command before quitting, the editor will respond with a "?" and give you another chance. (In fact, any unrecognizable or illogical editor command will result in a "?" response and an invalidation of the typed line.)

2.4 Editing Existing Files

EDIT command (e): When editing existing files, rules for the "ed", "a" and "w" commands can be enhanced. For the purpose of exposition, assume a file "junk" contains a single line: "I love telephones". To transfer the contents of "junk" into the editor buffer, any of the following commands may be used:

```
ed junk
```

or

```
ed
e junk
```

or

```
e junk
```

The editor will respond with a character count of the text entered into its buffer (in this case, 18). Note that "ed" is referred to as the EDITOR command, and "e" is called simply EDIT.

WRITE command (w) revisited: The editor remembers the name of the file it is working on. Therefore, the command:

```
w                    [followed directly by a CR]
```

in this case, is equivalent to:

```
w junk
```

FILENAME command (f): The editor remembers the file associated with the last "e" or "ed" command — if you forget, use the FILENAME command:

```
f
```

which will yield a response "junk" (in this case). If you want to change the name associated with the editor buffer, say to "garbage", you may type:

```
f garbage
```

to which UNIX will "echo" the new filename. The contents of the "editor buffer" remain unchanged, but the "filename" the editor remembers is now altered.

APPEND command (a) revisited: The APPEND command, used immediately after an existing file is brought into the editor, will add text to the end of the current buffer. For example:

```
ed junk
18                  [character count of file read into buffer]
a
--- especially Bell telephones
.
w
49                  [character count of buffer written]
q
```

will result in a two line file "junk" now containing:

```
I love telephones
--- especially Bell telephones
```

Repeated EDIT (e) commands: If at any time you have finished working with an editor file, you may issue a new "e" command without "quitting" the editor. Consider:

```
e junk      [transfers 18 character file "junk" into
18                                          editor buffer]
{ editing session }
w           [writes 40 character editor buffer contents
40                                        into file "junk"]
e nuts      [transfers 23 character file "nuts" into
23                                          editor buffer]
{ editing session }
w           [writes 37 character editor buffer contents
37                                        into file "nuts"]
q
```

Note, when the command "e nuts" is typed, the old contents of the editor buffer (from file "junk") is destroyed and replaced by the contents of file "nuts"; if the contents of an existing editor

buffer has been modified in any way, and you do not write it onto a file before invoking a new "e" command, UNIX will respond with "?", giving you another chance. The editor always remembers the filename associated with the latest "e" or "ed" command invoked.

READ command (r): The final command of interest here is the READ command. This command reads the contents of a specified file, and adds it to the end of the existing contents of the editor buffer. Consider:

```
e junk     [transfers 18 character file "junk" into editor
18                                                    buffer]
r nuts     [adds 23 character file "nuts" to the end of the
23                          existing contents of editor buffer]
w          [writes combined 41 character editor buffer onto
41                                                 file "junk"]
q
```

Note that a "r" command does not change the file name currently associated with the editor; an "f" command, issued anywhere in the above program, would yield a response "junk", the name of the last file entered via an "e" or "ed" command.

2.5 Printing Text

PRINT command (p): The PRINT command "p" will print the contents of the editor buffer (or specific parts of it) on the terminal. We specify the line numbers where printing is to begin and to end, separated by a "comma", immediately followed by "p". Assume we have a file "stocks" containing the following lines

```
AT & T              [line #1]
General Motors      [line #2]
Georgia Pacific     [line #3]
Ford                [line #4]
```

Consider the following commands:

```
e stocks
41
2,4p                          [print lines 2 to 4]
General Motors                [line #2 -- starting line]
Georgia Pacific               [line #3]
Ford                          [line #4 -- ending line]
q
```

Note that 0 (zero) cannot be used as a starting line number, that the ending line number must be greater than or equal to the starting line number, and that the ending line number must be less than or equal to the last line in the buffer; a breach of these rules will cause a "?" to be printed on the terminal, indicating an irrational command.

The editor command

```
4,4p
```

will print only line #4 of the text and has an equivalent abbreviated form:

```
4p
```

End of buffer symbol ($): It should also be noted that in many typing sessions, you will lose track of the size of your buffer, i.e., the number of the last line. To alleviate this problem, a special symbol "$" can be used to designate the last line in the buffer. For our file "stocks"

```
$p
```

will print line #4, the last line in the buffer, and

```
1,$p
```

will print the entire buffer contents.

As a further form of abbreviation, when printing a single line, the terminating "p" may also be deleted, i.e., the following editor commands are equivalent for the file "stocks": "4,4p", "4p", "4", "$,$p", "$p", "$".

CEASE PRINTING command: It should be noted that many times a long print sequence is started, and for some reason you wish to stop it; depressing the "DEL(ETE)" or "BREAK" key will cause printing to cease and return control to editor.

Current line symbol (.): The editor also provides the concept of "current line"; the "most recent" line that we have done anything with is symbolically denoted by "." (referred to as "dot"). If we issue a print command

```
.p
```

or, in an abbreviated form,

```
p
```

the current line will be printed. Consider a more complicated sequence of print commands, based on our file "stocks":

```
e stocks
41
1,2p                    [causes line #1 and #2 to be printed]
AT & T
General Motors          [sets "." to line #2 at this point]
p                       [prints current line, i.e., line #2]
General Motors
.,$p                    [prints current line #2 to end of buffer]
General Motors
Georgia Pacific
Ford
q                       [upon quitting the editor, "." = "$" = "4"]
```

As shown above, the "." can be used as a line number in any print command.

PRINT LINE NUMBER symbol (=): In complex editing situations, you may lose track of the current line number. The command

```
.=
```

will result in a printout of the value of "dot". Similarly, "$=" will yield the value of the last line in the buffer, but this command will not change the value of "." to "$")

Relative line number addressing: The editor also allows printing of lines relative to the current ("dot") line. Consider the following commands:

```
.+1p                    [prints "next line"]
.-1p                    [prints "previous line"]
.+3p                    [prints "third line after current line"]
```

In all of the above cases of single line printing, the "." and "p" may be optionally deleted without effect.

Multiple line printing may also incorporate this relative format, such as:

```
.-1,.+1p
```

will print the "previous, current and next lines". For multiline printing, the "." again is unnecessary in either argument, but the suffix "p" must be present. Also, if the "." is retained, the "+" may be deleted in forward relative addresses, i.e. ".+3p", "+3p", ".3p", ".+3", "+3" and ".3" are all equivalent.

The "$" (end of buffer) may be used instead of the "." in similar expressions, such as:

```
$-1p                [prints the "next to last" line in buffer]
$-3,$p              [prints the "last four" lines in buffer]
```

Also, there are abbreviations of relative printing commands, such as:

```
-
```

which is equivalent to ".-1p", "--" which is equivalent to ".-2p", "---" which is equivalent to ".-3p", and so forth. Similarly, the command:

```
+
```

is equivalent to ".+1p", "++" is equivalent to ".+2p", "+++" is equivalent to ".+3p", etc. Finally, a simple carriage return will print the next line (equivalent to ".+1p").

A warning in all relative printing statements is that: (1) the value of "." can change, after a command has been executed, (2) lines referred to must be valid lines in the buffer, i.e., in the range "1" to "$", inclusive, and (3) in multiple line printing statements, the beginning line number must always be less than or equal to the ending line number. To ensure your understanding, assume a file "number" exists, with contents:

```
one
two
three
four
five
six
```

and consider the following editor dialogue:

```
e number
28
6p              [prints line #6, "dot" set to "6"]
six
-3              [prints line .-3 = line #3, "dot" set to "3"]
three
++              [prints line .+2 = line #5, "dot" set to "5"]
five
p               [prints current line, "dot" unchanged]
five
(carriage return)   [prints next line, "dot" set to "6"]
six
+               [attempts to print line #7
?               [invalid command, "dot" unchanged]
.-3,-p          [prints lines .-3 = "#3" to .-1 = "#5"]
three
four
five
q               ["dot" set to "5" when quitting editor]
```

Special meaning of , and ; as line numbers: As a final note, the line range "1,$" may be abbreviated by the single character "comma" (,). The range ".,$" can be replaced by the single "semicolon" (;) character. These range abbreviations for line numbers may be used wherever applicable, not only in PFINT commands.

LIST command (l): There is another printing command called LIST ("l" ---- that's lower case "L", not the number "1") which follows all the rules of the "p" command. For normal text, the "l" and "p" commands will yield identical results, but the "list" command has the capability of indicating unprintable characters. Examples are: "tab" lists as "→", "backspace" lists as "←", and other control characters will print as a "backslash" followed by a string of digits, such as "\07"; if a line is over 72 characters in length, the "list" command will print it on multiple lines with each line, except the last, terminated with a "backslash" (\) to indicate continuation.

If any invisible characters (other than "backslash" at the end of a line, "tab" or "backspace") appear in a listing, the odds are that your finger slipped in typing, as you almost never want them. Some of these special characters can cause havoc in a large program, and the "p" command will be of no use in trying to locate them.

2.6 Deleting and Adding Lines

WRITE command (w) revisited: With the introduction of various techniques of line numbering, it may be wise to return to the WRITE command momentarily. When using the "w" command, the entire editor buffer contents need not be written onto a file; you may choose specific lines, if you wish, such as:

```
$-9,$w
```

will write the last 10 lines of the buffer onto the last file indicated by the last "ed", "e" or "f" command, and

```
-1w good
```

will write the line previous to the current "dot" line into file "good".

As a final note, the value of "dot" will not be affected by any WRITE command.

READ command (r) revisited: The READ command:

```
4r junk
```

will read the contents of the specified file (in this case, "junk") and add it to the editor buffer after the indicated line

number (in this case, line #4). If the line number is omitted, "$", or the last line in the "editor buffer", is the default (as was illustrated in a previous section). Also, "0" is a valid line number, if you wish to insert the file text at the beginning of the editor buffer. After the READ command has completed execution, "dot" is set to the last line read into the editor buffer.

APPEND command (a) revisited: Now let's revisit the APPEND command. Consider:

```
4a                          [adds text after line #4]
{ text to be added }
.                           [quits "append" mode]
```

which adds text after line #4. The integer "4" was illustrative; actually, any line number in the range "1" to "$", "." or valid relative addressing (as described for the "p" command) may be used. In addition "0a" is allowed to add text to the beginning of the editor buffer. It should be noted that at the end of the appending session, "dot" will be set to the last line of text added, and all line numbers of original text following the appended lines will be changed. A final comment is that "a" alone defaults to ".a". This is useful when first entering the editor via "e file_name", which sets the "dot" to the last line read from the file; an "a" (without line number) will now append text to the end of the editor buffer.

INSERT command (i): A similar command is INSERT ("i"), which adds text before the indicated line number, such as:

```
1i                          [inserts text before line #1]
{ text to be inserted }
.                           [quits insert mode]
```

which will add lines before line #1 or at the beginning of the editor buffer. Valid line numbers are "1" through "$", "." or valid relative addresses. As a default, "i" is equivalent to ".i". Again, at the end of a session, "dot" is set to the last line inserted.

DELETE command (d): Another useful command is DELETE ("d"), which may be used for single or multiple line deletions, such as:

```
4,$d        [deletes line #4 through the end of the buffer]
+1d         [deletes "next line" (after "dot" line)]
d           [deletes "current line" -- equivalent to ".d"]
```

In all cases of deletions, "dot" is set to the line after the last line deleted, unless you deleted lines from the end of the buffer, in which case "." is set to "$". The range of allowed line numbers is again "1" to "$", "." or valid relative addresses.

CHANGE command (c): A related command is CHANGE ("c"), which can be used in single or multiple line format; this command deletes designated line numbers and replaces them with desired text, such as:

```
4,9c                          [deletes lines #4 through #9]
{ replacement test }
.                             [quits change mode]
```

It should be noted that the number of replacement lines need not equal the number of lines deleted, and you may issue the CHANGE command for a single line, such as the command "7c", which will delete line #7 and replace them with text which follows. After completion of a CHANGE session, "dot" is set to the last replacement line. The valid range of line numbers is again "1" to "$", "." and relative addresses. As a default, "c" is equivalent to ".c".

MOVE command (m): You can move lines within the buffer via the MOVE ("m") command, of general form:

```
A,BmC
```

where "A" and "B" represent the beginning and ending line numbers, respectively, of lines to be moved ("B" must be greater or equal to "A", and, if only one line is to be moved, a single line number replaces the couplet "A,B"), and these lines will be placed after line "C". The line numbers "A", "B" and "C" may fall in the usual range "1" to "$", "." or relative addresses; in addition, "C" may have value "0" to move text to the beginning of the editor buffer, and "C" may not fall within the inclusive range dictated by "A" and "B". The "dot" is set to the last moved line, and the original lines of text are moved, i.e., they no longer exist in their original location.

COPY command (t): A related command COPY ("t") provides the same effects as the MOVE command, except the original lines remain intact. Also, if we view the general form "A,BtC", "C" may have values "0" to "$", "." or relative addresses, and "C" may fall within the inclusive range dictated by "A" and "B".

To check your understanding of these commands, consider manipulating the previous "number" file on Page 11, as follows:

```
e number
28
2a              [append after line #2]
help
.               [quits append mode - "dot" set to "3"]
3i              [insert before line #3 - equivalent to ".i"]
I'm
.               [quits insert mode - "dot" set to "3"]
4c              [delete line #4 - replace with new text]
lost
!!!
.               [quits change mode - "dot" set to "5"]
5,.+1d          [deletes lines #5 and #6 - "dot" set to "5"]
1,2m5           [moves lines #1 and #2 after line #5 --- "." = "5"]
7t$-1           [copies line #7 after line "$-1" = "6"]
w               [writes buffer onto file "number"]
35
q               [quits editor]
```

Check your interpretation of the commands with my current contents of file "number":

```
I'm
lost
four
one
two
five
six
six
```

A final warning about the "a", "i" and "c" commands ----- the terminating "." is easily forgotten, and will cause garbage to be erroneously added to the buffer, until you realize your error. At this point "bite the bullet" and make use of the "d" command (carefully).

2.7 Text Searches and Special Characters

FORWARD SEARCH command (/.../): Many times you wish to search for a line(s) containing a particular character string -- this is called "Context Searching". An editor command:

```
/string_of_characters_you_want_to_find/
```

will locate the next occurrence of the string of characters between the "slashes" (called a "regular expression"). The search starts with the next line after the search command is given (i.e., line .+1), continues to the end of the file (i.e., line $), "wraps around" to line #1 and continues to the current "." line. The first line containing the match that is found is set to "dot", and the line is automatically printed for verification; if many lines in the buffer contain the "regular expression", only

the first one encountered will be found by the search command. If no match is found, the editor will prompt with "?". The "regular expression" may be composed of any printable or non-printable characters, with some characters ("metacharacters") having special meaning, as delineated below.

REVERSE SEARCHES command (?...?): We may also search for lines in a reverse order, i.e., start our search at ".-1", continue backwards to "1", "wrap around" to "$" and continue backwards to "." . This reverse search is instituted by a "regular expression" enclosed by "question marks", as:

```
?string_of_characters_for_reverse_search?
```

Context addressing: In addition to finding lines containing specific character strings, the search command itself may be substituted for a line number (called "context addressing"), when using the READ, APPEND, DELETE, PRINT, CHANGE, INSERT, MOVE, COPY or WRITE commands. The following commands are all valid:

```
/help/d
/old/,/old/+3p
?five?i
```

SEMICOLON separator (;): Suppose you have a file of the following form:

```
....        ["...." lines do not contain "a" or "b"]
....
ab
....
....
bc
....
....
```

Starting at line #1, we might expect the command

```
/a/,/b/p
```

to print all lines between "ab" and "bc" inclusive. Actually, only the line "ab" is printed. This is due to the fact that searches for "a" and "b" both start at the same point, and "/a/" and "/b/" both find the same line, "ab". Worse, if a line before "ab" contained a "b", the entire print command would be in error. The problem is that the "comma" separator for line numbers doesn't set "dot" as each address is processed ---- "dot" is reset only after a command is actually executed.

In the editor, the "semicolon" can be used just as a "comma" separator of line numbers, but the use of ";" forces "dot" to be reset to the first line number, after it is evaluated. The command:

```
/a/;/b/p
```

prints the range of lines from "ab" to "bc" inclusive. The line containing "a" is found first, "dot" is changed to that line number, and then the search command for "b" is instituted, starting at the next line.

Another interesting use for the "semicolon" is a command like:

```
23;/a/
```

which would generate a search for the string "a" beginning at line #24. The initial "23" causes "dot" to be set to that line, and the search then proceeds with the next line. You can also use a command like

```
0;/a/
```

to start a search at the beginning of a file (this is one of the few places where "0" can be used as a legal line number).

Metacharacters in regular expressions: Some characters, called "metacharacters", have special meaning when included in "regular expressions; they are the "period (.)", "left bracket ([)", "asterisk (*)", "backslash/brace" (\{)" combination, "circumflex (^)", "dollar sign ($)", "pound sign (#)", "at-sign (@)" and "backslash (\)".

BACKSLASH metacharacter (\): If you desire to eliminate the special meaning of any character, the "backslash" (called the "escape character") can be used. Simply preceding the "metacharacter" by the "backslash" will accomplish this. The "pound sign", which is a "global metacharacter", has the property of "erasing" the previous character typed; the following line:

```
IX#\#2
```

is equivalent to "I#2", as the first "#" erases the previous character, but the second grouping "\#" removes the special meaning of the "pound sign" and introduces the "#" character into the buffer. Similarly, for the "at-sign", the sequence "\@" will return you to a new line, making you think you've erased the line; in reality, the text will contain the "@" symbol and you are actually typing on the same line. If you wish to remove the special meaning of the "backslash" character itself, the sequence "\\" will accomplish this. If we wish to search for an unlikely sequence of text "#\", the proper search command is:

```
/\#\\/
```

The "backslash" found at the end of a line has special meaning also; it removes the meaning of the "newline" ("carriage return") character, and you are in effect typing continuously on the same

line. Finally, in the search mode, we would have difficulty trying to match a "slash", such as:

```
/ab/c/
```

would not match "ab/c"; the "slash" between "b" and "c" would indicate a complete search command followed by "c/" (garbage), and a "?" would be printed indicating an error. A proper search could be accomplished by:

```
/ab\/c/
```

since the combination ""\/" is equivalent to searching for the "/" character, once its special meaning is removed.

PERIOD metacharacter (.): The "period metacharacter" is a regular expression standing for any single character. Thus the search

```
/a.b/
```

finds any line where "a" and "b" are separated by any single character, and would match any of the following strings:

```
a-b
abb
a+b
a.b
a b
```

If you wish to locate the text string "a.b" only, you must use the "backslash" to remove the special meaning of the "." metacharacter, i.e.,

```
/a\.b/
```

The "." is very useful in matching "metacharacters" and is very useful in eliminating non-printable characters, as will be seen in the next section.

DOLLAR SIGN metacharacter ($): The "dollar sign", when used in a regular expression represents the "end of the line" (as opposed to "end of buffer" when used in line numbering). Revisiting our original file "number" on Page 11, we could locate the line "two" with the search command:

```
/o$/
```

since only an "o" at the end of a line is sought. The search /o/ could match "one", "two" or "four". The "$" must be used as the last character in a "regular expression" to retain its special meaning.

CIRCUMFLEX metacharacter (^): A related metacharacter, "^" (the "circumflex" or "hat" or "caret" symbol), will restrict your searches to characters appearing at the "beginning of a line". The search command:

```
/^o/
```

would match "one" in the file "number", but not "two" or "four". The "^" must be used as the first character in a "regular expression" to retain its special meaning. Other uses for the special meaning of "$" and "^" will be discussed later. Also remember the special meaning of "$" and "^" may be "turned off" by preceding them with a "backslash".

ASTERISK metacharacter (*): Next, a character in a regular expression followed by a "*" ("asterisk" or "star") stands for any number of consecutive occurrences of that character. Be aware that as "any number" means zero or more occurrences; a misinterpretation here can lead to many problems. Assume we are looking for a line containing a character string "abbbbbbbbbbc"; certainly a search command

```
/abbbbbbbbbbc/
```

would accomplish our aims, but the command

```
/ab*c/
```

is far simpler. If the search above encountered a line containing the string "ac" first, a match would have occurred; the string contains an "a" followed by a "c", separated by no "b's", which is a legitimate match for the couplet "b*" in the regular expression. Again, the special meaning of "*" may be "turned off" by preceding it with a "backslash".

BACKSLASH/BRACE metacharacters: Any character in a "regular expression" followed by "\{m\}", "\{m,\}", or "\{m,n\}", where "m" and "n" are non-negative integers less than 256, has special meaning. The grouping "\{m\}" indicates a match of exactly "m" occurrences of the character preceding this special sequence in the regular expression; "\{m,\}" matches at least "m" occurrences of that character; "\{m,n\}" matches any number of occurrences of that character in the range "m" through "n" inclusive.

Some examples are:

```
/aB\{2\}/     [Search for "a" followed by exactly two "B's"]
/40\{3,\}/    [Search for "4" followed by three or more "0's"]
/X\{5,9\}/    [Search for five to nine "X's"]
```

These "metacharacters" are cumbersome to type and use, and are avoided by experienced editors, except in special circumstances.

SQUARE BRACKETS metacharacter ([...]): The last "metacharacter" of interest is the "left square bracket" or "[", which is always coupled with a closing "right bracket". The construction

```
/[12ab]/
```

will cause a search for any single character ("1", "2", "a" or "b") enclosed in "brackets". Similarly,

```
/a[0123456789]*/
```

searches for a line containing a string, "a" followed by zero or more digits. The command

```
/a[0-9]*/
```

accomplishes the same thing; the "minus" may be used to indicate a range of lexically consecutive ASCII characters, such as: [0-9] is equivalent to [0123456789], [a-f] is equivalent to [abcdef], and [W-Z] is equivalent to [WXYZ].

If the "circumflex" is used as the first character of the string within "square brackets", then any single character, except "new line" ("carriage return") and the remaining characters in the string, is matched; the "^" at any position other than the first has no special meaning. In fact, the characters "*", "[", "\" and "." have no special meaning when enclosed by "square brackets".

Confused --- consider the search:

```
/^[^\^]/
```

which seeks a line which does not begin with a "\" or "^".

A final comment is that the "-" loses its special meaning if it is first in a string (after an initial "^", if any) enclosed in "square brackets", or if it is the last character in that string; also, the "right bracket]" will not terminate a string if it is the first character (after an initial "^", if any). Remember, after proper interpretation, the string within "square brackets" represents a single character; verify that

```
/[ ][d-w]/
```

will match "]", "[", or any lower case letter in the range "d" through "w"; it matches only a single character, as there is only one set of valid "square brackets".

2.8 Modifying Existing Test

SUBSTITUTE command (s): Probably one of the most important commands is SUBSTITUTE. This command ("s") is used to change strings of characters within a line or group of lines. The basic form is:

```
s/regular_expression/replacement_text/
```

which will replace the "regular expression" denoted (possibly containing "metacharacters", as previously described) by the "replacement text" in the current "dot" line. If we have a current line "I love telephones" and issue the command:

```
s/e t/e Bell t/p
```

the line is changed to "I love Bell telephones", and the appended "p" will cause the corrected line to be printed for your inspection. Note that "p" (or "l" for "list") may be appended to many commands to cause printing of the current line after the command has been executed, such as ".dp" will delete the current line and print the new "dot" line.

Metacharacters in regular expressions revisited: Using "regular expression metacharacters" in SUBSTITUTE commands is almost essential to good editing style. Let's consider various common examples using these special characters. To simplify matters, in all the examples below, assume we are always making substitutions in the line:

```
I love telephones
```

the command

```
s/^/Boy, do /p
```

will print the corrected line "Boy, do I love telephones"; the "circumflex" alone indicated the replacement is to be placed at the beginning of the line.

Similarly

```
s/$/!!!/
```

will add "!!!" to the end of the line, due to the special meaning of "$" in the regular expression. The command:

```
s/t.*/girls/p
```

will print "I love girls" which is due to the regular expression "t.*" (character "t" followed by zero or more of any character - this type of expression is very useful when replacing the remainder of a line after a certain point).

The command:

```
s/[oe]/x/p
```

will print "I lxve telephones"; the regular expression "[oe]" searches the line for either an "o" or "e", and the SUBSTITUTE command will act only upon the first match found.

Multiline substitutions: A more general form is:

```
1,$s/regular_expression/replacement_text/p
```

which causes a possible substitution in all lines in the buffer; the substitution is made only if the "regular expression" is matched in any given line. Of course, the range "1,$" is illustrative and may be replaced by any valid range of line numbers, including those containing "dot" and relative addresses. You may also choose a single line number, or if none is indicated, a default to "dot" is made. Note, the "p" suffix, in this case, will cause UNIX to print only the last line in which a substitution is made.

GLOBAL suffix command (g): We may also change all occurrences of a regular expression in a line by appending the SUBSTITUTE command with the letter "g", (representing "globally, across the line") for example:

```
s/[oe]/x/gp
```

will change "I love telephones" to "I lxvx txlxphxnxs". Remember, without the "g" suffix, only the first match in the line will be affected by the SUBSTITUTE command. Also note that the additional appending "p" will cause printing to occur, and only the order "gp" is allowed ("pg" is illegal).

Asterisk metacharacter anomalies: Using the "g" suffix may cause problems when using the "asterisk" metacharacter in a regular expression. Consider the line "I lov!! t!l!!!phon!!s", which would require all single or consecutive occurrences of "!" to be replaced by a single "e". The proper substitution is:

```
s/!!*/e/g
```

which would cause a conversion to "I love telephones". The regular expression "!*" would yield quite different results, i.e., "eIe eleoeve etelepeheoenese", as UNIX notes between every pair of "non-!" characters, there are zero or more "!'s" and, therefore, an "e" is substituted.

For the same line "I lov!! t!l!!!phon!!s", the substitute command:

```
s/!*/e/
```

will yield unexpected results. The line will be converted to "eI lov!! t!l!!!phon!!s"; the editor determines that no "!"s" exist between the beginning of the line and the first character "I", and the substitution is made. The wording in the UNIX/TS User's Manual is vague (at best): "If there is any choice, the longest leftmost string that permits a match is chosen"; this should be interpreted as "the leftmost match is chosen, regardless of how long the consecutive string is". An illustrative example is that the lines:

```
h?lp???
h??lp???
h???lp???
h????lp???
```

would all be converted to "help??? by the command:

```
s/??*/e/
```

Only the first (or leftmost) occurrence of "?", followed by zero or more "?"s" will be affected, not the longest consecutive grouping of "?"s" in the line.

Breaking up a line: There may be cases when you wish to "break a line in two". For the line "I love telephones", the command:

```
s/e t/e\(CARRIAGE RETURN)
t/
```

will create two separate lines "I love" and "telephones". The "backslash" before the "CARRIAGE RETURN" will cause the "new line character" to be transmitted to UNIX, as replacement text, although you are physically placed on a second line to complete the the SUBSTITUTE command.

Removing strings from a line: It is extremely useful to eliminate certain characters within a line; the command

```
4,9s/^ *//p
```

will eliminate any number of "blanks" occurring at the beginning of lines #4 through #9; the "//" replacement (with no internal spaces) means "replace with nothing".

Removing non-printable characters: Another tricky point of interest is the case when a line listed ("l" not "p" command) indicates an erroneous non-printable character, such as:

```
I love tel\07ephones
```

The "\07" represents the "bell" character - the question is how to remove it. The substitution

```
s/\\07//
```

will not work. However,

```
s/l.e/le/
```

will work, as the "." represents any single character (even non-printable ones) between the "l" and "e".

Metacharacters in replacement test: Other than the characters "#" and "@", which have special meaning anywhere in UNIX programming (unless "escaped" by a preceding "backslash"), characters within the replacement text with special meaning are the "backslash (\)", the "slash (/)", the "percent sign (%)", and the "ampersand (&)".

Consider our line "I love telephones", and the command:

```
s/lo/l/o/p
```

We wanted to change the line to "I l/ove telephones" (don't ask me why!), but instead the editor will print "?". This is due to the fact that UNIX assumed the replacement string ended with the third "slash" -- to place a "/" literally in the replacement text, it must be "escaped", such as:

```
s/lo/l\/o/p
```

A similar statement holds for the "backslash".

AMPERSAND metacharacter (&): The "ampersand" in a replacement text (not in a regular expression) represents the entire "regular expression" just matched, i.e.,

```
s/love/& \& &/p
```

will cause the line "I love & love telephones" to be printed. The first and third "&" in the replacement text cause the regular expression "love" to be substituted while the second "ampersand" is "escaped" (by "\&" notation) and is a literal replacement.

Repeated regular expressions: Another time-saver is illustrated by the following command,

```
/love/s//like/p
```

this command will search for a line containing the string "love", set "dot" to this line number, and obey the SUBSTITUTE command. The "s" command will change "love" to "like" in that line, because "two consecutive slashes (//)", representing the "regular

expression", is shorthand for "the last regular expression used".

Note: the use of this shorthand notation may also be used in repeated searches. Assume the search "/love/" is instituted and the wrong line is found -- the text wanted is another line containing the string "love". Simply type "//" for another forward search or "??" for a reverse search, as the editor remembers the most recent pattern, or "regular expression", searched for, (in this case, "love"). This procedure may be repeated as many times as necessary.

Repeated replacement text: If "%" is the only character in the replacement, the replacement text in the most recent substitute command is used as the replacement text in the current substitute command. The "%" loses its special meaning when the replacement text is more than one character long, or when it is escaped by a "backslash" (i.e., the couplet "\%").

UNDO command (u): An extremely useful command is the UNDO command, or simply:

```
u
```

which negates the effect of the last SUBSTITUTE command issued, if you really "botch" things up. You cannot use the "u" repeatedly to eliminate two or more previous "substitutions" --- only the last line changed by the most recent "s" command, can be "changed back to its original form".

2.9 Global Changes

GLOBAL command (g): The GLOBAL command "g" is used to perform one or more editing commands on lines containing a specified pattern (regular expression). The command:

```
g/love/p
```

will print all lines in the buffer containing the string "love", while

```
4,20g/love/p
```

will print only the matched lines in the indicated range (line #4 through #20, in this case).

Let consider a more sophisticated global command, such as

```
6,11g/love/s//like/gp
```

The first step is to mark every line, in the range line #6 to #11, that matches the indicated pattern "love". Then for every such line, the command following is executed with "." initially set to that line. In this case all occurrences of "love" in that

line are replaced by "like", and the line is printed. The string "love" is the regular expression of the "s" command (represented by the "//"), since the last regular expression used was "love" in the global search pattern; all occurrences in a line are changed because of the "gp" suffix on the "s" command, as explained previously.

EXCLUDE command (v): We have a diametrically opposed command EXCLUDE "v" which acts upon all lines not containing the pattern, for example:

```
v/love/p
```

prints out all lines not containing the string "love".

Multiline global commands: What about multi-line or multiple commands under control of a global command? Consider

```
g/love/s/love/like/\
s/telephones/radar/
```

this command will search the entire buffer for lines containing the string "love". For each such line found, the first occurrence of "love" is changed to "like", and the first occurrence of "telephones" is changed to "radar". The "backslash" at the end of the first command line tells the editor that another command will follow; this may be repeated for as many lines as desired, but the final line in the global command must not contain the terminating "\". The entire buffer is searched globally because, in the absence of any line number preceding "g", the range "1,$" is the default. Note that "//" was not used instead of "/love/" in the first substitute command, because, if more than one line contained "love", the last "regular expression" remembered, after the first line's substitution is complete, is "telephones" ---- be careful; this is due to the fact that the "g/love/" portion of the command is executed first to mark all lines containing the string "love", and then the SUBSTITUTE commands are sequentially executed for each matched line, never to return to the regular expression "love" associated with the "g" command.

You may also use multi-line commands such as "a", "i" and "c", as:

```
v/love/a\
I really hate telephones!\
.
```

This command seeks all lines in the buffer not containing the string "love", and appends each one with the line "I really hate telephones!". Remember the terminating "\" must appear on all lines, except the last, in the global command. As a final note, you may not nest another "g" or "v" command within a current global command.

UNDO command (u) revisited: One final note might be made with regards to the UNDO command. Typing a "u" line after a global substitute command will only reverse the effect of the last substitution (not all substitutions) made via the global command --- a common error!

INTERACTIVE GLOBAL command (G): An example of the INTERACTIVE GLOBAL command is:

```
5,20G/love/
```

This command will search the indicated line range ("5,20" in our example), and mark every line that contains a match to the "regular expression" (in this case, "/love/"); should the line range be omitted, the entire editor buffer is searched for matches. Then, sequentially, every matched line is printed, "." is changed to that line, and a single command line may be entered and executed (note that "a", "i", "c" or any global command will not be accepted by "G"). After the execution of that command, the process is repeated for the next marked line. The "carriage return" ("new line") acts as a "null" command; an "&" alone will cause the re-execution of the most recent command executed during the current invocation of the "G" command. The "G" command can be terminated by issuing an "interrupt" signal (DEL or BREAK key); also, any error in an interactive command will cause UNIX to print a "?" and exit "G".

INTERACTIVE EXCLUDE command (V): This command is identical to the INTERACTIVE GLOBAL command, except that all lines not matching the designated "regular expression" are marked.

2.10 Temporary Escape from the Editor

ESCAPE command (!): There are occasions when you may wish to issue a UNIX level command, but you want the editor buffer to remain intact for further work. If you type:

```
!any_UNIX_command
```

your current editing state will be suspended, and the UNIX command indicated will be executed. When this UNIX command is completed, the editor will prompt with another "!", at which time you may continue editing (note: "dot" is unchanged by this process). An example is

```
ed
{ editing session }
!date
{ UNIX prints the date }
!
{ editing continued }
q
```

2.11 Miscellaneous Editor Commands

There are other useful editor commands, which should be mentioned.

EDITOR PROMPT command (P): If we issue the command:

```
P    [that's capital "P"]
```

the editor will prompt with an "*" for all subsequent commands. This command alternately turns this prompt mode on and off; it is initially off.

HELP commands (h and H): Occasionally, you will obtain "?" diagnostic outputs, by making an error in a command issued or attempting to quit the current editor buffer before writing its contents unto a file. If you issue the command:

```
h
```

a short error message will be printed, explaining the most recent "?" diagnostic. For more "help", you may issue the following command upon entry to the editor:

```
H
```

and all "?" diagnostics will be explained as they are issued.

The EDIT and QUIT commands (E and Q) revisited: You may use the commands:

```
E
```

and

```
Q
```

which are analogous to the EDIT ("e") and QUIT ("q") commands previously described, except that the editor does not check to see if any changes have been made in the editor buffer after the last "w" command.

JOIN command (j): The JOIN command "joins" two contiguous lines by removing the "new line" character between them, i.e.,

```
4,5j
```

will combine line #4 and #5 into a single line #4. If no line numbers are indicated, then the range (.,.+1) is assumed.

MARK command (k): The MARK ("k") command is useful to label special lines with a lower case letter address. A command

```
5ka
```

will identify the letter "a" with current line #5; if no line number is given, "dot" is the default, and any other lower case letter may be used in place of "a". The address form "'a" is thereafter an alternate valid address for that line.

Delimiter replacement in s-command: One technique which is useful when the "replacement text" includes many "slashes" is to use a command similar to:

```
s!love!///!
```

which will replace "love" by "///". We have removed the special meaning of the "slash" by using "!" as a "delimiter" in the SUBSTITUTE command. The "!" could have been replaced by any character (other than a "space" or "new line") which does not appear in the "regular expression" or "replacement text", and is not a global metacharacter.

3. BASIC FILE COMMANDS

3.1 Listing Files and the File Tree Structure

LIST command (ls): Once you have created files, you may wish to list their names (not contents) using the LIST ("ls") command. Assume we have the files "junk", "number" and "stocks" in our "login directory"; for now, assume your "login directory" is simply a collection of files you've created via the "context editor". The command:

```
$ls                 [you type "ls" in response to "$" prompt]
junk                [UNIX
number                   lists
stocks                        filenames]
```

will cause filenames to be printed alphabetically (names beginning with numerals or strange characters will be listed first). It should be noted that all commands are issued in response to the "$" prompt from UNIX, and this fact will be implicit in most future examples.

If a filename begins with a "period" (called a "dot file"), the "ls" command, as shown, will not print it ----- to list all filenames, an "ls -a" command is required, noting the space between "s" and "-" is necessary. The list produced will be headed by "." and "..", which have special meanings, as explained in Section 3.6.

With an optional argument "-t", a temporal sort is provided, i.e., files are listed in the order in which they were last changed or created, most recent first.

```
$ls -t              [space between "s" and "-" necessary]
junk                [assumes "junk" was worked on last]
stocks
number
```

Another option "-l" (that's lower case "L", not number "1") will give a long listing (a lot of useful information), such as:

```
$ls -l          [again, "space" between "s" and "-"]
total 3
-rw-rw-rw-    1  owner_of_file  31  Sep22 12:56  junk
-rwxrwx---    1  owner_of_file  28  Aug30  1:26  number
-r-xrwx-w-    1  owner_of_file  41  Jan1   3:12  stocks
```

The first character printed is a "-" for an ordinary file, or "d" for a directory (the concept of "directory" will be explained shortly). The next three characters indicates "what the owner can do with the file"; the owner may "read"(r) the contents of the file, modify or "write"(w) onto it, "execute"(x) it (if the file is executable), or a "-" indicates the corresponding

permission is denied (these "file permissions" will be discussed in greater detail shortly). The next three characters refer to permissions of the owner's "group" (which may be administratively formed), while the last three indicate permissions of all "others". The next number will be explained shortly and need not concern you now. The entry "owner_of_file" symbolically represents the owner of the named file. The number following (i.e., 31, 28 and 41) indicates the number of characters in the corresponding file. Next comes the date and time of the last change to the file, and finally, the filename is listed. The "total 3" line before the "long listing" indicates the number of 512 character "blocks" used to store the listed files.

Also, any of the options listed may be combined (in any order), such as:

```
ls -lta
```

will cause a long listing of all files via a temporal sort.

Finally, if you get tired of a listing, hitting the "DEL(ETE)" or "BREAK" key will cause printing to cease and return you to the UNIX command level.

Directories and the File Tree Structure: Each user has his/her own "home directory", identified by his/her login name (user ID), which contains only files that belong to him/her. When you create a new file, unless you take special action, it resides in your "home (or login) directory", and is unrelated to any other file of the same name that might exist in someone else's directory.

A directory, in general, represents a collection of files (and possibly other subdirectories), and explicitly it contains the names of the files/directories in it and control information necessary to access those files; an ordinary file is simply a collection of data or characters stored on UNIX.

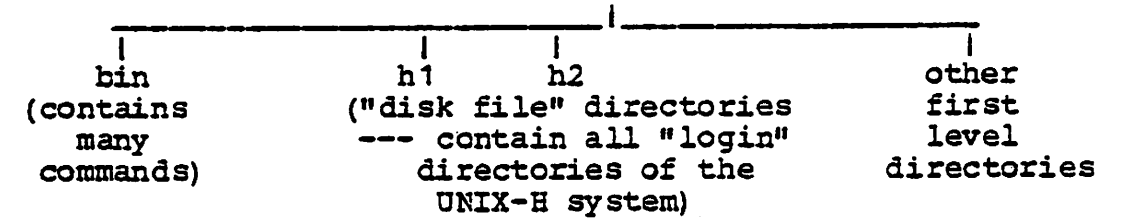
The set of all files UNIX knows about are organized into a large "tree" structure. A "master UNIX directory" is the "root" of this "tree", and after several levels of "branches" (subdirectories), we arrive at the "leaves of the tree" (the ordinary files). The "root" directory, using a UNIX convention, is designated by "/". The following section will hopefully clarify the concept of the "tree structure".

LIST command (ls) revisited: Now, try the command:

```
ls /
```

which will list various files/directories in the "root" directory, including "bin" and "h1" (assuming you are working on the UNIX-H system). If the argument of an "ls" command is a

"directory" name, the names of all files/subdirectories within that designated directory will be listed, according to the options "-l", "-t", "-a" which may be invoked. We may look at a "crude" diagram to illustrate this portion of the "tree", i.e.,



Next try the command

```
ls /bin
```

which will list the files and directories of a first branch directory "bin". You should recognize some of the command files listed, like "ed" and "ls", as this directory contains the most commonly used commands. Other commands are located in the directories "/usr/bin" and "/usr/lbin".

Next try listing the contents of "branch h1" via the command:

```
ls /h1
```

and among the directories listed, you should note your own "login name" (user ID). Directory "h1" is the "parent directory" for some people in Division 56; if your "login" is on a different file system, substitute its name for "h1". It should be pointed out here that the "h" identifies the physical computer (a PDP-11 system), while the digit "1" identifies a particular disk file on that system.

Then proceed up the "tree" and try:

```
ls /h1/your_login_name
```

which will list your personal files and subdirectories and is entirely equivalent to the command:

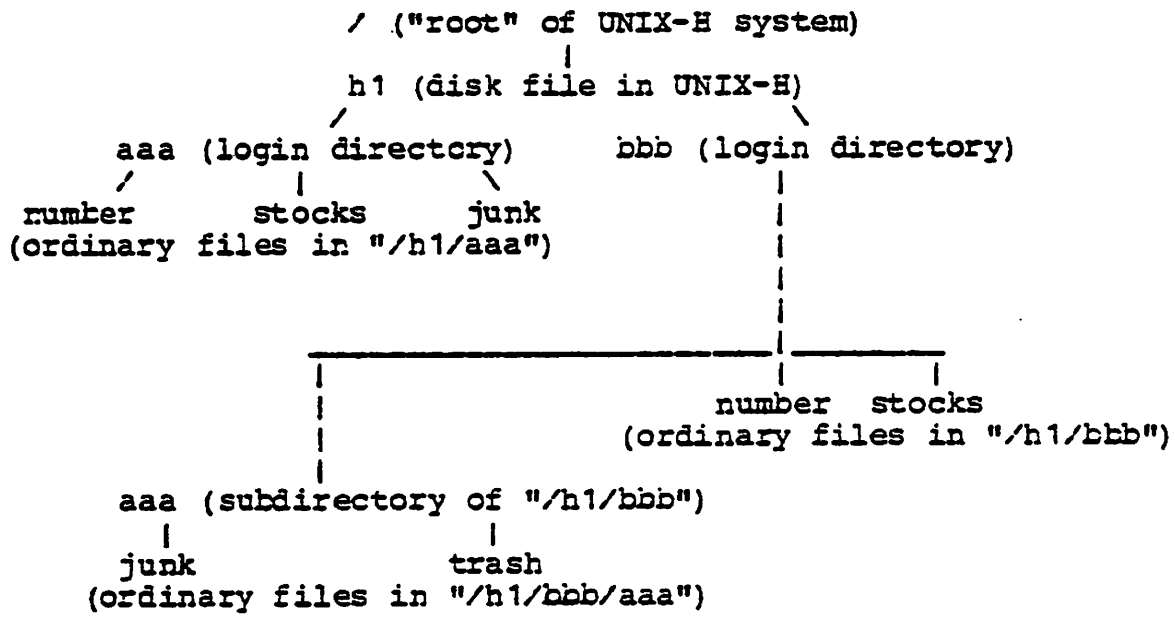
```
ls
```

which lists the contents of your "current" directory, as a default, as no directory name argument is given (presumably, you are working in your "login" directory).

Pathnames: It is a universal rule in UNIX that anywhere you can use an ordinary filename, you can use an entire "pathname". A "pathname" obviously implies the full name of the path you have to follow through the "tree of directories", starting from the

"root" ending with the desired file or directory name. When using the full pathname, separate each directory or file with a "slash"; the initial "/" is necessary to start the search in the "root" directory --- otherwise a "branch" from your "current" directory is implied (up to now, the "current" and "login" directories were assumed to be the same, but they need not be).

LIST command (ls) again revisited: For the purposes of illustration, consider the following "tree structure":



Note that many names are repeated (on purpose), such as "aaa", which may be a "login" directory in the "h1" disk file system, or a "subdirectory" under "login" directory "bbb" ---- these are distinct directories that can be uniquely identified by their pathnames. In the following examples, always assume our "current" (working) directory is login "bbb" (full pathname "/h1/bbb").

Now, we may define the general form of the "ls" command as:

```
ls options directory_name
```

The "options" are "-a", "-t" and "-l" (or some combination of these), as previously described. The "directory_name" may be a full or partial pathname. If a partial pathname (a pathname which does not start at the "root", designated by the leading "/") is indicated, the search for "directory_name" will be a "branch" from the current working directory); if "directory_name" is omitted, the contents of the "current" working directory will be listed, as previously shown. In any case, the command will produce a list of files and directories contained within "directory_name". The command:

```
ls aaa
```

assumes a "branch" from our current directory "/h1/bbb", and is entirely equivalent to:

```
ls /h1/bbb/aaa
```

and will list the filenames "junk" and "trash", belonging to subdirectory "aaa" in login directory "bbb". However, a full pathname command:

```
ls /h1/aaa
```

will list "junk", "number" and "stocks", the ordinary files contained within login directory "aaa". If you are confused at this point, please reread this section, as it is imperative that you understand the concepts of "tree", "branch" and "pathname" to properly execute commands.

If, however, the command contains the "-d" option as:

```
ls -ld aaa
```

which is equivalent to the full pathname command (in our illustrative example):

```
ls -ld /h1/bbb/aaa
```

then the status of directory "/h1/bbb/aaa" itself will be given (in this case, a "long listing"), and no information about its contents (ordinary files "junk" and "trash") will be printed.

We may also operate on selected ordinary files by listing their filenames at the end of the "ls" command (separate filenames by "spaces"), as:

```
ls -l stocks aaa/junk
```

will give long listing information about the ordinary files, with full pathnames "/h1/bbb/stocks" and "/h1/bbb/aaa/junk", as "branching" from the current working directory is again implied.

If you request information about a non-existent file or directory, UNIX will cryptically notify you that it does not exist.

3.2 File Protection

CHANGE MODE command (chmod): For files, you may wish to deny file "read" or "execute" access, or more probably "write" permission, to other users. This can be accomplished by the CHANGE MODE ("chmod") command, the general format of which, is:

```
chmod mode file1 file2 file3 . . . . .
```

where each file named has its mode changed (permissions to read, write and execute) according to the "mode" argument which may be "octal" or "symbolic" in nature. Note that filenames listed are assumed to be branches from the "current" working directory, unless you use full pathnames.

Read, write and execute permissions are allocated to three classes of users: "owner" of file, "group members " (as administratively assigned), and "others" (all other UNIX users). The use of the "ls -l" command, described previously, will indicate the current permissions associated with a file.

Octal mode: The "octal form" of mode designation is a string of four integers: a "0" followed by three octal digits (range 0 - 7). Each of the last three digits assigns permissions as follows:

0: can not read, write or execute
1: execute only
2: write only
3: execute and write only
4: read only
5: execute and read only
6: read and write only
7: read, write and execute

Note that "reading a file" means you can access the contents of the file; the "cat" or "ed" commands require this "read" capability. "Write" permission allows you to modify the contents of a file, which is required in an editor write, destination of a copy command, etc. "Execute" permission refers to files containing executable code, such as a compiled "C-program" or a "Shell Procedure" containing executable UNIX commands.

In the four digit number, the second digit refers to "owner" (so-called, "user") permissions, the third digit refers to "group" permissions, and the fourth refers to the permissions of all "others". Therefore

```
chmod 0740 junk
```

indicates file "junk" can be read, written onto and executed by the "user/owner", can be read by "group" members, and can not be accessed at all by "others". Note: the leading "0" in the "mode" may be deleted with no effect --- see the UNIX/TS User's manual for the meaning of a "non-zero" first digit.

Symbolic mode: The "mode field" may also be entered symbolically, where the first character may be chosen from the list:

u user/owner
g members of owner's group
o all others
a everybody

The second character may any of the following:

= to list a new set of permissions
- to delete permissions
+ to add to existing permissions

The remaining "mode symbols" may be chosen from the list:

r read
w write
x execute

Examples are:

chmod a+w junk

which adds the "write" permission to all users of file "junk" (in addition to existing permissions), and:

chmod u=rw junk

will assign "write" and "read" permissions to the "user/owner" and will leave permissions for "group" and "others" unaltered.

When a file is created using the "ed" command, the "owner" is assigned "read" and "write" permission, while "group" and "other" users have "read" permission only, by default. Any time thereafter, the "owner" of the file only may use the "chmod" command. It might be pointed out that a large file could have "write" permission denied its owner -- a protection against careless overwriting or removal.

Changing directory permissions: A final comment is that "chmod" may also be applied to directories in the form shown above, except "read", "write" and "execute" must be reinterpreted. "Read" permission implies that you have the right to list the names of files contained in the directory. "Write" permission allows you to create, destroy or modify files within the directory. "Execute" permission is equivalent to the right to "search" the directory, i.e., being able to enter it. Directory permissions are not as straight-forward as file permissions. If a directory has "write", but not "execute" (search), permission, you cannot remove a file because you can't gain access to the directory. If you wish to modify files within a directory, make sure a "7" directory mode is assigned to you; if you wish to read a file, a "5" directory mode is required. The usual default given when you create a directory is octal mode "0755", which is usually the permission status most user's desire.

3.3 Removing, Linking, Copying or Changing File Names

REMOVE command (rm): To REMOVE an existing file (or files), we may use the "rm" command, such as:

```
rm filenames
```

Leave a space between "rm" and each filename. Be careful here, as files listed are removed -- permanently!!! "Filenames" are assumed to be "ordinary files" in your current working directory, unless full pathnames are indicated.

If you attempt to remove an "unwritable" file, UNIX will notify you by printing the "permission mode" associated with that file, and expects a response from you. Such a message will be printed only if the file you've attempted to remove does not possess "write" permission. If you respond to UNIX with a line beginning with a "y", the file will be deleted, and otherwise the file remains intact. Only the "directory" containing the file must have "write" permission, for that file to be removed --- be careful. An alternative is to use the command:

```
rm -f filenames
```

where the "-f" option forces the removal of a file (in a "writable directory"), regardless of the permissions associated with that file, and will inhibit the "mode warning".

REMOVE DIRECTORY CONTENTS command (rm -r): The "rm" command with a "-r" option, i.e.,

```
rm -r directory_name
```

will remove all files in the specified directory, including all subdirectories and their contents. Finally, "directory_name" itself will be removed, if it is not your "current" working directory. It is important to note that "directory_name" itself and any subdirectory you wish to affect must have a "mode 7" permission associated with it. If any ordinary file to be removed is "unwritable", you will receive a "mode warning", but it can still be removed as described above.

Fortunately, to remove all files in your "current" directory, it must be explicitly named via a full pathname (or its abbreviation ".", explained in Section 3.6), and the command

```
rm -r
```

will do nothing; probably, you would never want to issue this command on purpose anyway.

DELETE INTERACTIVELY command (rm -ir): Another useful command is DELETE INTERACTIVELY ("rm" command with a "-ir" option), of the form:

```
rm -ir directory_name
```

In response, UNIX will sequentially list all files/subdirectory names in "directory_name", expecting a response from you.

If the response is prefixed by the word "directory", such as:

```
directory some_directory_name:
```

UNIX wishes to enter that (sub)directory for the removal process. A "y" response allows UNIX to enter, while a response beginning with any character other than "y" (preferably a "carriage return") will cause that directory to be skipped in the removal process. If you attempt to enter a (sub)directory without possessing a "mode 7" permission, you will not succeed in removing any files and the "interactive deletion" process will abort prematurely.

A listed file (or file partial pathname), without a prefix, indicates UNIX is awaiting your instructions concerning the removal of the named file. Again, a "y" response will cause removal of the named file, while a response beginning with any character other than "y" will leave the file intact. If the file in question is "non-writable", a "mode warning" will not be issued.

A (sub)directory may be subject to the same removal process, if all files/subdirectories within it have been removed previously. This "interactive deletion" process will sequentially penetrate all levels of subdirectories.

Hitting the "DEL(ETE)" or "BREAK" keys will cause you to exit the REMOVE command.

If you wish to "interactively delete" files in your "current" working directory, you must explicitly indicate its full pathname (or its abbreviation "."). The command:

```
rm -ir
```

will do nothing.

MOVE or RENAME command (mv): The MOVE ("mv") command should be called "RENAME THE FILE". The command:

```
mv file1 file2
```

changes the name of file "file1" to "file2". As usual, unless you specify full pathnames, the files listed are assumed to be "branches" of the "current" working directory. It should be

noted here that if you attempt to "move to" a file that already exists, the original contents of that file will be overwritten (lost forever!!), provided the file is "writable"; in our sample command, "file2" is in danger. If that file is "unwritable", UNIX will print its "permission mode" (similar to the REMOVE command) and await a response from you; if the first character you type is "y", the "move" will be executed, while any other response will neglect the MOVE command.

A command:

```
mv subdirectory_1 subdirectory_2
```

will change the name of "subdirectory_1" to "subdirectory_2", noting both subdirectories must appear in the same "parent" directory.

COPY command (cp): A similar command COPY ("cp") makes a duplicate copy of a file, leaving the original intact, i.e.,

```
cp file1 file2
```

will form a copy of "file1" under filename "file2". Beware that if you COPY a file to another one that already exists, the original contents are overwritten, if the the "target" file (in our example, "file2") is writable.

In the last example, we have copied ordinary files within the current working directory. It is possible to enhance this capability, by using complete pathnames, such as:

```
cp /h2/txx/one /h1/kew/first
```

In this case, file "one" in directory "/h2/txx" will be copied into file "first" in directory "/h1/kew". The only restrictions are that the files and directories involved must have proper permissions, and directories must be in the same UNIX system --- disk file systems "h1" and "h2" reside on the "UNIX-H system"; note, the "rm" and "mv" are similarly restricted to the UNIX system containing the "current" working directory. If you wish to copy files to another UNIX system, see the description of the "usend" command in a later section of these notes.

Another variation can be demonstrated by the command:

```
cp file_name directory_name
```

which copies "file_name" in the "current" working directory to a target file with pathname "directory_name/file_name". The argument "file_name" may be replaced by a sequence of names separated by "spaces". You may not copy "directories" using the "cp" command.

LINK command (ln): The same ordinary file may appear in several directories under possibly different names. This feature is called "linking". Once a "link" is established, all users linked to that file have equal status. If one owner modifies the file contents, all users linked to that file see the same modifications; it is, in fact, the same file. To create a "link" to an existing file, invoke the command:

```
ln existing_file target
```

where "existing_file" (an ordinary file) may be a simple filename, if it is located in your current working directory, or a full path name, if not. The "existing file" will now appear in the "target" directory with name "existing_file" (or the last component of "existing_file", if a full path name was designated). If you wish to rename the file in the "target" directory, then "target" should be a full path name. The commands:

```
ln /h1/kew/junk /h1/knn
ln /h1/kew/junk /h1/too/garbage
```

will produce a file with three "links";"/h1/kew/junk", /h1/knn/junk", and /h1/too/garbage" are the same file with access from three directories.

Note that you may not link files in different disk file systems, such as:

```
ln /h1/kew/junk /h2/bbb
```

is an illegal command; "h1" and "h2" are different disk files on the same UNIX-H system. You may not link directories. Also, if one owner of a linked file invokes the "rm" command, only that owner's link to the file is removed (the file is destroyed if it has only one link associated with it). The number of "links" associated with a file is indicated by the number before the file owner's name listed via the "ls -l" command.

3.4 Printing File Contents

CONCATENATE command (cat): We can print the contents of files via a direct UNIX command without entering the editor. The CONCATENATE ("cat") command will print the contents of one or more files onto the terminal. Consider the command

```
cat file1 file2
```

which will "concatenate" (join together) the files "file1" and "file2", in the order listed (there may be more than two files listed), and print them on the terminal. Remember, to leave a "blank space" between "cat" and each file name. As always, files not listed with full pathnames are assumed to be "branches" of the "current" working directory. Also, do not "cat" directories,

as garbage will be printed.

PRINT command (pr): The PRINT ("pr") command has the following general form:

```
pr file_names
```

and acts as the "cat" command except it prints the contents of the named files in a formatted manner. It provides a heading on each page, including date, time, page number and filename. Many options, such as specified page headers, variable width and length pages, multi-column outputs, double-spaced outputs, etc., are available; the interested user should consult the UNIX/TS User's Manual.

It should also be noted that printing due to an issued "cat" or "pr" command may be terminated by hitting the "DEL(ETE)" or "BREAK" key.

OFFLINE PRINT command (opr): There is also an OFFLINE PRINT ("opr") command which produces a printout on a lineprinter in the computer center. The general format of this command is:

```
opr options filenames
```

where "filenames" is the list of files (separated by "spaces") you wish to print. Useful "options", again separated by "spaces", are:

-b # will send the output to bin number "#", rather than the default bin read from your password file

-f code will yield an output printed on the following forms, instead of the default standard 11x14 (blue and white striped) computer paper, according to the following codes:

code	form description
unlined	unlined 11x14 paper (high quality print)
8by11	unlined 8x11 paper (high quality print)
xerox	photo reduced Xerox print
x8by11	full 8x11 Xerox print
x2on1	two physical pages photo-reduced on a single 8x11 sheet (Xerox process)

-c# will produce "#" (maximum 99) copies of the output, rather than the default single copy

See the UNIX/TS User's Manual for further details.

3.5 Special Characters associated with Filenames

When using commands associated with filenames, such as "ls", "cat", "pr", "mv", "cp", "rm" and more to come, there is a new group of "metacharacters" to help (and maybe confuse) you. They are the "asterisk(*)", "question mark(?)" and "square brackets([ ])" -- and of course the "#" (erase) and "@" (kill line) characters still work. Their characteristics are somewhat or drastically different than their "editor" counterparts -- be wary.

For illustrative purposes, assume our "current" working directory contains the following files:

```
.profile
junk
junky
bad_junk
jinx
```

QUESTION MARK metacharacter (?): The "?" may be used to match any single character (except the ".", if it is the first character of a filename). The command

```
rm ?profile
```

will not match file ".profile" or any other file in our directory, and UNIX will so indicate. However

```
rm j?n?
```

would successfully REMOVE files "jinx" and "junk".

SQUARE BRACKETS metacharacter ([...]): A string of characters enclosed in "square brackets" will match any single character in the string. You may indicate a lexical range with a "minus sign", such as "4-9" is expanded to "456789"; no other correspondence with the analogous editor "regular expression" structure should be inferred. The command

```
cat j[a-z]n[kx]
```

would concatenate and print files "jinx" and "junk".

ASTERISK metacharacter (*): The last metacharacter is the "asterisk", which may be translated to mean "anything at all". A leading "*" will match any or no prefix (except the ".", if it is the first character in a filename). Thus

```
pr *junk
```

would cause a formatted printing of files "bad_junk" and "junk". A trailing "*" will match any or no suffix, i.e.,

```
pr junk*
```

would print "junk" and "junky". A string enclosed by two "*"'s will act upon all files containing the string, i.e.,

```
pr *junk*
```

would print "junk", "junky", and "bad_junk". Again

```
pr *ro*
```

would not yield a match with file ".profile", as the first "*" does not match the "." in a filename, when it is the first character. Finally, "*" alone matches any file except those beginning with a "."; hence, a very dangerous command in UNIX is

```
rm *
```

which could wipe out all files except ".profile", in our illustrative directory.

Note that "metacharacters" can be used in the last directory or file in a pathname; all the following pathnames would match "/h2/abc/help" (and possibly other files in directory /h2/abc):

```
/h2/abc/h*
/h2/abc/[ho]?lp
/h2/abc/*e*
```

A pathname such as "/h*/a??/help", while not illegal, is confusing; it is best to use "metacharacters" only in the last file or directory in the path.

3.6 Directory Manipulations

CHANGE DIRECTORY command (cd): If you want to work on someone else's files rather than your own, you can change directories using the CHANGE DIRECTORY command "cd", for example

```
cd /h2/knn
```

will change the directory you are currently in to login name "knn" in disk file "h2". Hence, your "current" (or "working") directory need not be your "login" directory. Once you have changed to a foreign directory, you can only manipulate files to the extent the owner of that directory allows permissions. Also, you may only change directories within the same machine; for example, if your "login" is on UNIX-H, you have "cd" access to file systems "h1" and "h2" only. As usual, if you do not specify a full pathname, the "cd" command will seek a "branch" from the "current" working directory.

Special meaning of . and .. : You may go up one level in the tree structure by typing

```
cd ..
```

where ".." is shorthand for "parent of whatever directory you are currently in". If you are in your subdirectory "nose", i.e., "/h1/your_login_name/nose", the above command will leave you in the "parent of nose", i.e., "/h1/your_login_name". For completeness, "." is a shorthand name for the directory you are in, as mentioned previously. In fact, the command "ls -a" (which prints the names of all files/subdirectories contained within your working directory, including those beginning with ".") will always list as its first two names, your current working directory "." and your parent directory "..".

Finally, the command "cd" with no argument will automatically return you to your "home" or "login" directory.

PRINT WORKING DIRECTORY command (pwd): By this time, you're probably so damn confused, that you don't know what directory you're in. UNIX saves you with the PRINT WORKING DIRECTOFY command, i.e.,

```
pwd
```

will print the full pathname of whatever directory you're currently working in.

MAKE DIRECTORY command (mkdir): It is often convenient to arrange one's files, so that all files related to one special area of work are grouped together. We can accomplish this with the MAKE DIRECTORY command "mkdir".

Assume we wish to write a "book of three chapters", we could make a new directory:

```
mkdir book
```

which forms a directory of pathname "/h1/your_login_name/book" (again, I am assuming throughout that you are working in your "login" directory, which is located on the "h1" file system). Now you can get into this directory by the command:

```
cd book
```

Note that a "full pathname" is not necessary here, as a "branch" starting from your "current directory" is implied. Now you can form files "chap_1", "chap_2", "chap_3" via the editor, and you can locate any chapter, say "2", by using the full pathname "/h1/your_login_name/book/chap_2" . You may extend this directory creation procedure to any level desired.

REMOVE DIRECTORY command (rmdir): You may remove an empty directory using the REMOVE DIRECTORY command "rmdir" (possibly by referring to the directory by a "full pathname"); if it is not empty, you must delete all files in the directory first. For the "book" directory above, the following sequence will eliminate the directory:

```
cd book          [changes from "login" to "book" directory]
rm *             [removes all chapters from "book" directory]
cd               [return to "login" directory]
rmdir book       [removes empty directory "book"]
```

Obviously, the "rm -r book" command, as previously described, can be used to accomplish the same feat.

3.7 Redirection of Inputs and Outputs

Most of the commands we have seen so far produce output on the terminal. Some commands, like the editor, also take their input from the terminal. In UNIX, it is possible to replace the terminal with a file for either (or both) input and output.

OUTPUT REDIRECTION symbol (>): If we issue the command:

```
ls
```

a list of filenames is printed on the "standard output" (the terminal) by default. This output may be redirected to a file, say "names" in this example, with the following command:

```
ls >names
```

This command creates file "names" if it does not already exist, or will overwrite it, if it does; the output of the "ls" command is delivered to file "names" only -- no output is printed on the terminal. The symbol ">" is used throughout UNIX to mean "put the output on the following file, rather than on the terminal".

Another example is to combine several files into one file, i.e.,

```
cat file1 file2 file3 >single
```

will concatenate files "file1", "file2" and "file3", and, instead of printing them on the terminal, will write them onto the file "single".

APPEND REDIRECTION symbol (>>): There is an associated symbol ">>" which means "append the output to the end of the following file, rather than print on the terminal". The command:

```
cat file1 >>file2
```

will add the contents of "file1" to the end of "file2".

INPUT REDIRECTION symbol (<): Another universal symbol "<" which means "take the input from the indicated file, rather than the terminal". Assume we have a file "do_it" with the following contents:

```
$d
1m$
$-9,$w
q.
```

Now, we initiate the command:

```
ed file1 <dc_it
```

which brings "file1" into the editor for a working session and accepts editor commands from file "do_it" (i.e., deletes last line of "file1", moves the first line to the end of the editor buffer, write onto "file1" the last 10 lines of the editor buffer, and quits the editor) rather than from the terminal. It is possible to combine input and output redirection in a single command.

Shell description: Before continuing, the term "shell" will appear from time to time, but will not be explained in detail in these notes. The mysterious "shell" is simply a program which interprets what you type as commands and arguments; it acts as the interface (a transparent interface) between the user and the raw computing power of the machine. At a beginner's level, "shell procedures" will not have to be written. Any interested user has access to many tutorials, courses and other documentation associated with the "shell"; try to restrict your reading to items associated with the new "UNIX/TS Bourne Shell", which everyone is now using.

PIPELINE symbol (|): To understand the concept of a "Shell Pipeline", represented by the "|" (vertical bar) symbol, consider the following problem: we have three files "a", "b" and "c", which we would like to combine and print in a paginated format using the "pr" command. We could accomplish this with the following commands:

```
cat a b c >tempfile     [combines files]
pr tempfile             [prints combined file in paginated form]
rm tempfile             [remove temporary file]
```

but a single command line containing a "pipe" would suffice, ie,

```
cat a b c | pr  ["space" is not necessary before and after "|"]
```

The "pipe" takes the output from the first command (in this case, "cat a b c") and, instead of outputting it on the terminal (the "standard output"), forces it to be the "standard input" to the second command (in this case, "pr"). Any command, which would

normally write onto the terminal, can drive a "pipe"; any command, which reads from the terminal, can read from a "pipe" instead.

Another example of interest is:

```
date | cat - a b >c
```

The new "twist" here is the "-" argument in the "cat" command; this designation represents "standard input" from the terminal (a series of text lines terminated by a line containing only CONTROL-D, the "end-of-file character). In this case, "cat" accepts a redirection from the "pipe output". The above command will concatenate the "date/time" line (from the "date" command) with files "a" and "b", and write the entire text into the redirected output file "c". There may be more than one "pipe" in a command line.

MULTIPLE COMMAND symbol (;): Another command symbol is the "semi-colon"(;), which allows multiple commands to be executed before control is returned to the terminal. The multiple-command:

```
cat a b ; date
```

causes combined files "a" and "b" to be printed on the terminal, followed by a printout of the "date/time", and then control is returned to you.

Commands enclosed by "parentheses" and separated by "semi-colons" are are treated as a single command, such as:

```
(cat a b ; date)|pr
```

will combine files "a" and "b" (and does not print them on the terminal because of the "pipe" designation), executes the "date" command (and again does not print it), passes combined files "a" and "b" followed by the "date/time" line through the "pipe" to the "pr" command, which print them in a paginated format. Note that the "pr" command without any "filename" argument accepts input from the "standard input", or in this case, a redirection from the "pipe output"; the same holds true for the offline printing command "opr" and the concatenate command "cat".

3.8 Background Processes

This section will cover the techniques necessary to execute commands in the background, while the terminal becomes immediately available for other uses. This is most useful in a compiling, or executing, large "C" or text-formatting programs as a background process (which could take a reasonable amount of real time), while continuing to do other work.

BACKGROUND COMMAND symbol (&): To run a process (or execute a command) in "background", simply place an "ampersand" (&) at the end of the command line. UNIX will respond with a "process identification number", followed immediately by a "$" prompt, indicating you may continue with other work. This "process ID" uniquely identifies the process generated by the command line terminated by "&"; an exception is that many processes may be spawned by multiple commands in a "pipeline", in which case, the number of the last process in the "pipeline" is reported. As a note, you may not "hang-up" on UNIX while background processes are still active, or these processes will terminate prematurely.

With another use for the "&" character just defined, it should be clear at this point that "metacharacters" and other special symbols will have different meanings when used in the "editor", or at the UNIX command level in file/directory manipulations. This causes a great deal of confusion to the casual user or beginner; the "table of contents" references these special characters in various usages and may prove useful.

PROCESS STATUS command (ps): By issuing the command:

```
ps
```

you may obtain information about all active processes, including those operating in "background". UNIX will provide a list of "process ID's", the cumulative execution time for the process (actual amount of computer usage, not the passage of real time), and the command (with its corresponding arguments) that is being executed. There are optional arguments available for the "ps" command, and the interested reader is referred to the UNIX/TS User's Manual for details.

KILL command: At some point, you may wish to terminate specific processes (without "hanging-up" and terminating all background work). The command:

```
kill -9 process_IDs
```

will "kill" or terminate the processes identified by a list of ID numbers separated by "spaces" (the argument "process_IDs"). The "-9" argument is optional, but, if included, guarantees a "sure kill".

Immunity to Hang-Ups: To make any command, normal or in the background, immune to "hang-ups" (including any disruption between your terminal and the UNIX machine), simply use the form:

```
nohup command
```

For "pipeline" or multiple commands, the "nohup" prefix must be inserted before each command in the line. If the output of the command is not redirected to a file or sent to an offline

printer, it will be sent to a file "nohup.out", which will be created by UNIX in your current directory, if it does not already exist.

Time-Delayed Commands: A final way to run background jobs is via the "at/batch" command. You may specify:

```
at time date
command list
(Control-D)
```

The job list following the "at" command is terminated by a line containing only "Control-D" (simultaneously depressing the CONTROL and D keys). These jobs will be executed at the "time" and "date" indicated. The "time" may be specified by one/two digits, which is interpreted as hours, or by three/four digits, which is interpreted as hours and minutes; both designations assume a 24 hour clock, unless the suffix "am" or "pm" is used. The "date" may be the month followed by the day number, or a day of the week ("at" recognizes the month/day of week spelled out completely or abbreviated to three characters); also, two special days "today" and "tomorrow" are recognized. If the "date" is absent, today is assumed, if the hour specified is greater than the current hour; tomorrow is assumed, if it is less. Other variants for "time/date" exist; see the "at" page in the UNIX/TS User's Manual for details.

You may also specify times incrementally via the form:

```
at now + increment
```

where the "increment" is a number suffixed by "minutes", "hours", "days", "weeks", "months" or "years" (the singular forms are also accepted).

You may also issue a command of similar format, i.e.,

```
batch
command list
(Control-D)
```

which submits a "batch job". Such jobs are run whenever the load on the system falls to an acceptable level.

In response to either an "at" or "batch" command, UNIX will respond with a "job number". You can get a list of "job numbers" of all jobs not yet executed, by typing:

```
at
```

without any arguments. The list will include your login name, the "job number", an identifier a for "at" jobs or b for "batch" jobs, and the time each job is scheduled for execution. If you

wish to terminate one or more of your jobs before execution, use the command:

```
atz job_numbers
```

where "job_numbers" is a series of numbers separated by "spaces".

One final IMPORTANT note is that any "at" or "batch" job cannot send output to your terminal; all commands issued in this mode must have outputs redirected to a file (or an off-line printer).

3.9 Other Miscellaneous Commands

There are numerous other commands not yet discussed, which can be found in Section I of "UNIX/TS User's Manual", some of the more interesting or useful ones which will be discussed here.

News reports: You might want to consult "current news reports" concerning UNIX by typing the command:

```
ls /usr/news
```

which will print a rather long list of "news articles", some of which, may be of interest to you. To view a particular article, simply type:

```
cat /usr/news/name_of_article
```

Filename metacharacters may be used freely in the "name-of-article". An abbreviated command (a single "article_name" must be spelled out completely, without metacharacters) is:

```
news article_name
```

If you decide you're really not interested in the article after all, hitting the "DEL" or "BREAK" key will cause printing to cease and return you to the command mode.

Note, when you first login, UNIX will notify you as to all current "news reports" that you have not yet seen; to view these current items, simply type:

```
news
```

in response to a "$" command prompt. A marker ".news_time" keeps track of those articles not yet seen by you; do not destroy this file or UNIX will request that you read all news articles again (in this case, you may issue a command:

```
news >temp_file
```

which will deload the "news" queue and store these articles in a file for viewing later or destruction).

Inter-user MAIL command: Within UNIX, there exists the capability of sending and receiving "mail". To send mail to another user on the same UNIX system (for example, disk files "h1" and "h2" belong to system "UNIX-H"), simply type the series of commands:

```
mail login_name      [indicates person to receive mail]
{text of message}
.                    [line with "." only terminates message]
```

Note that a file's contents may also be mailed, via input redirection, i.e.,

```
mail login_name <file_name
```

will "mail" the contents of the named file to person identified by "login_name"

If you have mail waiting, UNIX will notify you upon login with the line:

```
you have mail
```

The mail is stored in a special file "/usr/mail/login_name", and you can retrieve it by typing:

```
mail
```

without any argument. Your mail is printed in reverse chronological order of receipt and is "postmarked" (the name of sender and the time of dispatch). After each message, UNIX issues a "?" query. A response of "d" will delete the current message and go on to the next; a response "p" will cause the message to be printed again; a "s filename" will cause the current message to be appended to "filename" in your current directory (if "filename" is omitted, "mbox" is the default and UNIX will create it , if it doesn't already exist); a "w filename" will store the message, without a "postmark" in the named file (file "mbox" is the default); a "q" will leave messages intact in the file "/usr/mail/login-name" and exit the "mail" command.

If you wish to send "mail" to another UNIX system, you will have to modify the command format, as follows:

```
mail system_name!login_name
{text of message}
.
```

where a complete list of the destination system names ("system_name") can be found on the "usend" page of the UNIX/TS User's Manual, and the "!" between the system and login names is mandatory.

You may also be interested in interactive communication between two users, who are logged in simultaneously; see the "write" command in the UNIX/TS User's Manual.

Sending Files to Another UNIX System: A vehicle exists to send ordinary files to a user on another UNIX system. The command "usend" has the following general form:

```
usend -d dest -m -u user filenames
```

With the above command you will send the files (represented by the last argument "filenames", which is a list of files separated by "spaces" --- note that filename "metacharacters" will work in this listing) in your current working directory to a destination system (argument "dest") and a destination user (argument "user"). For a complete list of destination systems available, consult the "usend" page in the UNIX/TS User's Manual or issue the command:

```
cat /usr/lib/udest
```

or

```
cat /usr/asp/udest
```

for an up-to-date table. The argument "user" is simply an appropriate "login" name on the destination system. The "-m" argument is optional, but, if present, will report to the destination user via "mail" when the file transfer is completed.

By default, the transferred files are delivered to a subdirectory "rje" (stands for "remote job entry") under the destination user's login directory. Subdirectory "rje" must have been previously created in "mode 777" for everything to work; the following commands (issued in the destination user's directory) will accomplish this:

```
mkdir rje
chmod 777 rje
```

The names of the destination files are the same as those sent, and will reside in the "rje" subdirectory upon completion of the transfer.

As an example assume I am in working directory "/h1/kew", and issue a command:

```
usend -d ihuxa -u eps file1 file2
```

which will send files "/h1/kew/file1" and "/h1/kew/file2" to the user "eps" on general purpose UNIX system A (via the destination code "ihuxa"). You will receive a response from UNIX, a "usend job number", indicating the file transfer will take place. Upon

completion, the transferred files will appear as "/a2/eps/rje/file1" and "/a2/eps/rje/file2" (assuming login "eps" is on the "a2" file system).

There are other options/variations available; see the UNIX/TS User's Manual "usend" command. Also, you may be interested in the "con" command, which sets up a communications link between two systems and also allows file transfer; also, the command "cpio" may be used to transfer entire directories.

DATE command: The DATE command, discussed previously, is quite simple, as typing

```
date
```

in response to a command prompt "$", will cause UNIX to print the current date and time on the "standard output" (the terminal), unless redirected. Options exist --- see the UNIX/TS User's Manual.

MANUAL command (man): The MANUAL ("man") command will print the latest copy of the page(s) in the "UNIX/TS User's Manual" of any command of interest, such as:

```
man at
```

would give full instruction of how the "at" command is utilized. An obvious drawback, is that you must know the exact command abbreviation for "man" to work. The command is extremely useful in obtaining "current updates" (or a description of new commands) of command capabilities, before they are formally published.

SET TERMINAL OPTIONS command (stty): An important command SET TERMINAL OPTIONS ("stty") sets certain input/output options, some of which are very important to the "C-language programming" or "text formatting". The general form of the command is:

```
stty option1 option2 option3 .......
```

The option "-tabs" will replace the "tab character" by an appropriate number of spaces when printing on the terminal. On terminals available, tabs are set across the page at an "8 space" interval. If this option is set, every time a "tab character" is encountered in printing, the carriage will move to the next "tab stop"; "tabs" are introduced into text via the "CONTROL-I" key (simultaneously hitting the "CONTROL" and "I" keys). Without this option, when reading a file containing "tab" characters, they will be ignored by the terminal.

The option "erase X" will cause "X" ("X" is illustrative and can be any single character, such as "CONTROL-H", representing "back-space") to be the "erase character", rather than "#". The option "kill X" similarly makes "X" the "kill character" rather than "@"

(many people prefer "CONTROL-X"). Finally, the option "ek" resets the "erase" and "kill" characters back to the default "#" and "@", respectively. The option "-lfkc" will inhibit the "new line" produced (as a default) after the "kill" character has been invoked; the option "lfkc" will return the default mode. See the UNIX/TS User's Manual for other less used options.

TAIL command: The command TAIL will deliver the last part of a file to "standard output" (the terminal), unless redirected; this is useful in checking where you ended in the last editing session or for splitting files. An example is

```
tail -6 junk >>garbage
```

which will remove the last six lines in the file "junk" and add them to the end of the file "garbage" (a redirected output --- hence, no printing on the terminal will occur). The "-6" option may be replaced by a "-" followed by any integer (up to the total number of lines in the file, or the limit of the temporary buffer area used by the "tail" command), and if this option is missing entirely, the default is "10 lines". Again, see the UNIX/TS manual for further options.

SPELL and TYPO commands: Two interesting commands are SPELL and TYPO, both of which tend to find typographical/spelling errors in text. The command

```
spell junk
```

will find likely spelling errors in the file "junk"; if your file is long, you can take a nap before results will be printed.

Similarly,

```
typo junk
```

will list probable typographical errors and unusual words, in decreasing order of peculiarity along with an "index of peculiarity" (a index of 10 or more is considered very suspicious). TYPO will usually provide a longer list than the SPELL command.

GET REGULAR EXPRESSION command (grep): Once you've found a spelling error (or for some other reason), you may wish to use the GET REGULAR EXPRESSION ("grep") command. This command searches a file for a particular character string, and has the general form:

```
grep -n regular_expression filenames
```

where any "regular_expression" to be located is specified (the special meaning of "metacharacters" are valid); if the regular expression contains "blank spaces", you should enclose it in "double quotes", or alternatively, a "blank" or any "metacharacter" enclosed in "single quotes" will lose its special meaning.

The argument "filenames" obviously represents any files you wish tc search (separated by "spaces", with filename metacharacters completely valid). The response from UNIX will be the filename and the line number (if the "-n" option is included), followed by the printing of the line containing the character string to be matched.

The "grep" command can also accept "pipe inputs", such as:

```
ls /usr/news | grep mail
```

The "ls" command passes the names of all "news articles" in directory "/usr/news/" into the "pipe". The "grep" command accepts the "pipe output" and selects only articles containing the string "mail", printing their names on the terminal.

WCRD COUNT command (wc): Another useful command is WOFD COUNT ("wc"). In gereral

```
wc file1 file2 file3 ----
```

will count the number of lines and words (a word is a maximal string of printing characters delimited by "spaces", "tabs" or "new line") and characters in each file listed, and print these results on the terminal, unless redirected.

PASSWORD command (passwd): The PASSWORD ("passwd") command allows you to change the "password" associated with your "login name", via

```
passwd
```

for which UNIX will request a new password and confirmation (since "echoing" or printing is turned off during the change process). UNIX may not allow you to change your "password", if it has been changed recently, nor will it accept "extremely simple passwords".

DIFFERENCE command (diff): The DIFFERENCE ("diff") command:

```
diff file1 file2
```

compares the two files listed and indicates the lines that must be changed to bring them into agreement. The outputs produced by this command are quite complex, and interested readers are referred to the UNIX/TS User's Manual for details.

Special file .profile: There is special file of interest: ".profile". After logging in, UNIX (the "shell" in particular) searches your "login directory". If it finds a file called ".profile", the commands therein are executed first, before reading commands from the terminal. You were assigned a standard ".profile" when you applied for your "user ID". Open ("cat") it to

view its contents; there may be many commands within it that you are unfamiliar with --- these are "Shell Procedures" which need not concern you at this introductory level. What is important is that you can edit this file to add "personal" commands, that are executed before you receive your first "$" prompt. I find the command "stty -tabs" a useful addition, especially when working on "C programs"; some people like the date/time printed, via the "date" command; I also modify my "erase" and "kill" characters via a "stty" command; you may have "mail" or "news" printed at login time; or may change your "command prompt" by adding the line:

```
PS1="YES MASTER!"
```

to humble the "damn computer" ("YES MASTER!" is only an illustrative replacement for the default "$" prompt ---- use your imagination).

When you issue a command, UNIX (actually, the "shell") must locate the directory in which the command is found. The default is to search your "current" directory first, then "/bin", then "/usr/bin", then "/usr/lbin", and finally, the subdirectory "bin" in your login directory, until the specified command is found. This default search scenario is dictated by the line in ".profile":

```
PATH=$PATH:$HOME/bin
```

You may alter the directory search by changing the "PATH=" line, which has the following general format:

```
PATH=path1:path2:path3:path4:.....
```

i.e., a sequence of full directory pathnames separated by "colons"; your "current" directory is indicated by the "null string". For example

```
PATH=:/bin:/usr/bin:/usr/lbin:/h1/kew/bin
```

is equivalent to the default search for my login directory, since the first pathname is the "null string", representing the "current" directory. Another example is:

```
PATH=/bin::/usr/bin:/usr/games
```

which searches for commands in directory "/bin" first, then in the "current" directory (again represented by a "null string"), then in "/usr/bin", and finally in the "/usr/games" directory (no games now --- you're supposed to be working).

Other commands of interest: There are numerous other commands which may be of interest to the individual user, such as:

- an emulated desk calculator (see "dc")
- an Assembler, or Fortran or Basic compilers (see "as", "fc", and "bs")
- fancy text formatters (see "mm", "nroff", "tbl", and also various papers in "Documents for PWB/UNIX Time-Sharing System")
- games (see Section VI of the UNIX/TS User's Manual --- "/usr/games/startrek1" isn't listed, but it's on the computer for your enjoyment, after 4:45, of course)
- the "C" programming language and compiler, and an input/output package for UNIX (see "cc" in the UNIX/TS User's Manual, and the book "The C Programming Language" by Kernighan and Ritchie, published by Prentice-Hall)

----- ENJOY YOURSELF -----

4. HOMEWORK

4.1 LEARN Program

The LEARN program is a good primary source of practice problems to gain familiarity with the editor and the UNIX file structure.

To enter the LEARN program, simply type "learn" in response to a "$" command prompt. You will be given explicit instructions on how to proceed, including a list of offered topics. You will be interested in the topics: "files", "editor" and "morefiles" (in that order). Each "course" (or topic) will be presented in a "tutorial manner", i.e., each lesson is prefaced with the background principles necessary to answer a specific question (this ideal is usually realized). The "teaching script" can follow three paths: a "fast" track for those who make no mistakes, and two levels of "slow" tracks (i.e., extra lessons) for those who are not perfect. If you're stuck on a question for a long time, you have the option (by refusing to try the problem again) of passing it up and going on, probably down a "slower track".

To exit the LEARN program, simply type "bye" in response to a "$" command prompt. You may reenter a subject at the point where you left off, by remembering the subject name and the number of the last lesson you've successfully completed.

4.2 Final Exam

After completing those sections of the LEARN program noted above, you might try this FINAL EXAMINATION (solutions are given in the next section). Do all the "examination" problems in a subdirectory "EXAM", which may be created via a command in your "login directory":

```
mkdir EXAM
```

and can be entered via the command:

```
cd EXAM
```

You can return to your "login" directory at any time by issuing the command:

```
cd
```

It should be noted that the following problems are quite "artificial", in that you may be forced to execute commands in a strange or impractical way, or analyze commands with little or no practical use. The intent of the problems is to make you aware of all of the commands at your disposal, and then you may choose a subset, and develop a style, suited to your needs.

Problem 1

Form a file named "begin" (in subdirectory "EXAM"), with contents:

```
one
two
three
```

Problem 2

Enter the editor anew (type the "ed" command with no filename associated with it). Load the following lines into the editor buffer:

```
four
five
six
seven
```

Use the "f" command to associate the filename "number" with the buffer. Next, "read" the contents of file "begin" at the beginning of the buffer. Check to see what filename is now associated with the buffer, and "write" the seven line contents of the buffer onto file "number", and quit the editor.

Problem 3

Bring the file "number" into the editor buffer, print its contents, and try the following commands, in the order shown. Try to figure out the editor responses or actions attributable to each command before actually working with the terminal.

```
.=                      [Command #1]
8p                             [#2]
.=                             [#3]
4,5p                           [#4]
+1,$p                          [#5]
p                              [#6]
----                           [#7]
.2                             [#8]
(carriage return)              [#9]
$-1,3p                        [#10]
.=                            [#11]
$1                            [#12]
1,$-2w                        [#13]
Q                             [#14]
```

Problem 4

Bring file "number" back into the editor buffer and perform the following actions:

(1) Copy line #1 at the end of the buffer
(2) Delete lines #1 and #2 of the buffer
(3) Move lines #1 and #2 of the current buffer after line #3 in the current buffer
(4) Copy lines #1 to #4 of the current buffer after line #3 in the current buffer
(5) Change line #4 in the current buffer to two lines:
    eight
    nine
(6) Insert, at the beginning of the buffer, the line:
    six
(7) Append the end of the buffer with line:
    seven
(8) Write lines #1 to #9 only onto file "number"
(9) Quit the editor

Now try to figure out the current contents of the buffer, and then print it out for verification.

Problem 5

Create a file "welcome" containing the following lines:

```
Hello!
I am UNIX.
This is to
inform you
that I am
your
powerful master
and
that this
terminal
will electrocute
you, should you
make an
error
- - - I demand
perfection!!!
- - - I demand
respect!!!!
```

Now bring "welcome" back into the editor buffer and perform the following actions:

(1) Change the word "UNIX" in line #2 to lower case letters (this humbles the damn computer)

(2) Change the word "electrocute" to "notify" (use a global command to make the substitution, noting that "electrocute" is the only word of text, not at the beginning of a line, that begins with the letter "e")
(3) Change the "powerful master" line to two lines "humble" and "servant" (note, the line you seek is the only line ending in "er" --- use a "context address" to locate it)
(4) Change all appearances of "demand" to "would like"
(5) Remove any number of "!" at the end of any line
(6) To make UNIX friendlier, change the first line to read "Hello Hello Hello Hello Hello Hello" using a single substitute command
(7) Print the contents of the modified buffer (see how humble UNIX has become), and destroy the buffer, i.e., leave the editor without writing the buffer contents onto file "welcome"

Problem 6

Create a file "wisdom" containing:

```
If nodeders
nodet
nodedings the
way summers
wrote sums,
then the first
woodpecker that
came along would
write.
```

Now bring the file into the editor buffer, and investigate the actions of the following commands (determine the action of each command before using the terminal):

```
(1) 1s/node/buil/
(2) +,$s//%/
(3) g/w[a-dr][c-j]../s//destroy civilization/
(4) g/sum/s//program/
```

Print the entire buffer and overwrite the file "wisdom".

Problem 7

This problem is a "real (expletive deleted)". Form a file "weird" containing the following text (BE CAREFUL and ENTER TEXT EXACTLY AS SHOWN):

```
To tire bhd nurd
it takes to 4 a task: tile
hb nord you bhink \\* should xxxake,
multiply by 4, and find the hair
change the tibble of measure
to the next 4 tibble. Thus we
locate 4 days for a hour
tark.
```

Now, for each command listed (in the order shown), determine the substitutions that will be made (without using UNIX), and then use the computer to physically execute the commands; after <u>each</u> command is completed, print out the entire buffer and compare the results with your "prophecies". The commands are:

```
(1)  1,-5g/d/s//e/\
     s/b/t/
(2)  3s/x*/t/
(3)  1,.2s/n.rd/time/
(4)  3s/\\\\*//p
(5)  v/e$/s/4/2/
(6)  g/ti[^m]e/s//estimate/
(7)  g/^[^i]/s/4/highest/
(8)  v/^t/s/..in..the.*//
(9)  g/[:,]/s/4/do/\
     s/\*/it/\
     s/xx*/t/
(10) g/2/s/ho/one-&/\
     s/^[^monkey]/al&/
(11) g/bb/s/..bb../unit/\
     +2s/rk/sk/
(12) w
     q
```

If you analyzed <u>each</u> command correctly, you are an EDITOR EXPERT.

<u>Problem 8</u>

Create the following files in the directory "/your_system/your_login/EXAM/":

(1) Create the file ".when", containing three lines:

```
one
t>w>>o
>thr>ee
```

where ">" represents the "tab" character, entered into text via (CONTROL-I)

(2) Create the file "was/where" (this is file "where" in an "EXAM" subdirectory called "was"), containing one line:

two

(3) Create the file "why" (enter exactly as shown with all spelling errors), containing:

If one advarces confidtly
in the directious of his dreams
and edevors to live a
life which he has iragined,
he will meet with sucess
unexeced in cummun huors.

(4) Create the file "was/words"

cane
cone
lane
lone
nine
none

Recall that files "begin", "number", "weird", "welcome" and "wisdom" in directory "EXAM" have been created previously in other problems.

Problem 9

Determine the effects of the following LIST commands issued in subdirectory "EXAM":

(1) ls
(2) ls was
(3) ls -a
(4) ls -t
(5) ls -l
(6) ls why where when
(7) ls -d was why
(8) ls -dl EXAM was
(9) ls w*
(10) ls *h*

Problem 10

Again assuming you are in subdirectory "EXAM" under your login directory, perform the following actions:

(1) Print your working directory
(2) Change to the "was" subdirectory
(3) Create a copy of "welcome" in directory "EXAM" and place it in the current directory with name "hello"
(4) Append files "where" and "hello" in the current directory to the end of file "begin" in directory "EXAM" (your current working directory is still "was")
(5) Return to the parent of your current working directory
(6) Produce a paginated output on the terminal of all ordinary "non-dot" files in directory "EXAM" and all of its subdirectories --- as an aid, printout all the names of files in "EXAM" and its subdirectories first
(7) Print out all lines containing the string "one" (including the filename and line number) in all files in directory "EXAM" (including subdirectories), whose filename contains a "w"
(8) Form a file "count" which contains the number of lines, words and characters (and the filename) contained in file "weird"
(9) Use the SPELL command to determine the spelling errors in file "why" ---- repeat using the TYPO command

Problem 11

Try the following problems using the OFFLINE PRINTER:

(1) Produce an 8x11 Xerox print of the manual pages for the commands "banner", "at", "usend" and "sleep"
(2) Produce a high quality 8x11 print of the file "weird"
(3) Produce a default computer printout of combined files "weird" and "welcome"
(4) Produce a reduced Xerox print of the file "words"
(5) Produce a paginated Xerox print (two input pages per one output page) of any file in directory "EXAM", whose name contains an "h" or "l" (no subdirectories)

Problem 12

A few more problems for your enjoyment:

(1) Combine the words "HI THERE" with file "welcome" and print it in a paginated format (do not create any temporary files and do not modify "welcome")
(2) Form a file "WWW", which contains a list of all files and subdirectories in "EXAM" beginning with the letter "w"
(3) "Cat" the file "welcome", using a "nohup" prefix ---- explain what actions UNIX will take
(4) Print a list of filenames in subdirectory "was", followed by the DATE/TIME, followed by the contents of file "welcome" on the terminal, using a single command line

(5) Deposit all spelling errors of files beginning with the letter "w" in directory "EXAM" (not subdirectories) into the file "tmp" --- create a background process
(6) Have the output of the "date" command sent to file "777" in 10 minutes and the contents of file "welcome" printed offline in 2 hours
(7) Kill the job to be run 2 hours from now
(8) Mail yourself a letter "I love myself" and the file "was/words"
(9) Read your mail --- destroy the "I love myself" message and save the file sent (with the postmark) in file "junk"
(10) Create a file "NEWS" containing the names of all news articles containing the character "2"

Problem 13

Try a few more, i.e.,

(1) Print out file ".when" on the terminal
(2) Return to your "login" directory
(3) Modify ".profile", such that the "tab" character is recognized by the terminal
(4) "Logoff" and "login" again
(5) Issue the command:

```
RS1="READY "
```

---- what does this command accomplish?
(6) Return to subdirectory "EXAM"

Problem 14

Answer the following questions, but do not issue the commands physically to UNIX.

(1) You wish to send some files to login "abc" on disk file system "b2" --- what preparations must login "abc" have completed before the files can be transferred
(2) Write the command necessary to send copies of files ".when" and "was/words" (both in directory "EXAM") to login "abc"
(3) Under what complete pathnames will the copies of the files appear in login "abc"

Problem 15

Now its time to clean up the files you've created in directory "EXAM":

(1) Change the mode of subdirectory "was" to allow only write permission to yourself --- all other permissions denied
(2) Deny all permissions to everybody for the files "welcome" and "why"
(3) Remove all "dot" files
(4) Change the name of file "hello" to "welcome"
(5) Delete as many files in directory "EXAM" and all of its subdirectories as you can, without changing "modes" --- remove interactively
(6) If you have any files left, change modes as needed, and remove "EXAM" and its contents

4.3 Examination Solutions

Below are my solutions for the preceding "exam" problems.

Problem 1

The solution is:

```
$ed             ["$" is the UNIX command prompt]
a
one
two
three
.
w begin
14              [character count]
q
```

Problem 2

The solution is:

```
$ed             ["$" is the UNIX command prompt]
a
four
five
six
seven
.
f number
number          [UNIX response]
0r begin
14              [character count]
f
number          ["r" does not change buffer filename]
w
34              [character count]
q
```

Problem 3

The results of the commands are summarized by the following editor session:

```
$e number     ["$" is the UNIX command prompt]
34            [character count]
.=            ["dot" is set to the last line read in]
7             [response #1]
8p            ["8" is non-existent line]
?             [response #2]
.=
7             [response #3, "dot" unchanged]
4,5p
four          [response
five                  #4]
+1,$p         [equivalent to "6,7p"]
six           [response
seven                 #5]
p             [equivalent to ".p"]
seven         [response #6]
----          [equivalent to ".-4p"]
three         [response #7]
.2            [equivalent to ".+2p"]
five          [response #8]
(CR)          [equivalent to ".+1p"]
six           [response #9]
$-1,3p        [equivalent to "6,3p, invalid command]
?             [response #10]
.=
6             [response #11, "dot" unchanged]
$1            [equivalent to "71"]
seven         [response #12]
1,$-2w        [equivalent to "1,5w"]
24            [character count, response #13]
Q             [command #14, unconditional quit]
```

The file "number" now contains the first <u>five</u> lines of the buffer, which were written:

```
one
two
three
four
five
```

Problem 4

The editor session necessary to accomplish the stated tasks is:

```
$e number     ["$" is the UNIX command prompt]
24            [character count]
1t$           [command #1]
1,2d          [command #2]
1,2m3         [command #3]
1,4t3         [command #4]
4c            [command #5]
eight
nine
.
1i            [command #6]
six
.
$a            [command #7]
seven
.
1,9w.         [command #8]
Q             [unconditional quit]
```

The buffer, upon quitting the editor, contained:

```
six
five
three
four
eight
nine
three
four
one
one
seven
```

and the file "number" now contains the first nine lines of the final buffer contents.

Problem 5

These are the commands that I would use (not necessarily the only way to do the problem) are:

```
[1] 2s/UNIX/unix/
[2] / e.*/s//notify/
[3] /er$/c
    humble
    servant
    .
```

```
[4] g/demand/s//would like/g
[5] g/!!*$/s///
[6] 1s/.*/& & & & & &/
[7] ,p
Hello Hello Hello Hello Hello Hello
I am unix.
This is to
inform you
that I am
your
humble
servant
and
that this
terminal
will notify
you, should you
make an
error
- - - I would like
perfection
- - - I would like
respect
Q
```

Problem 6

The commands will accomplish the following:

[1] Line #1 will be converted to:

```
If builders
```

[2] The command is equivalent to "2,9s/node/buil/" and will affect lines #2 and #3 as follows:

```
built
buildings the
```

[3] The only match to the global regular expression "w[a-dr][c-j].." is the string "write" in line #9 ------ the command is equivalent to "9s/write/destroy civilization/" and line #9 becomes:

```
destroy civilization.
```

[4] The global regular expression "sum" is matched in lines #4 and #5, and the command is equivalent to "4,5s/sum/program/", producing modified lines:

```
way programmers
wrote programs,
```

The modified file "wisdom" now contains:

```
If builders
built
buildings the
way programmers
wrote programs,
then the first
woodpecker that
came along would
destroy civilization.
```

Problem 7

Here are the answers to the "world's most difficult problem" (in great detail):

[1] The global prefix "1,-5g/d/" is equivalent to "1,3g/d/", since the relative address "-5" is equivalent to ".-5" and "dot" is "7" (the last line read into the buffer); lines #1 and #3 contain the string "d", and are subject to the substitutions that follow.

For line #1, the two substitute commands are equivalent to "s/d/e/" and "s/b/t/", recalling that the "null" regular expression associated with the first substitute is equivalent to "d", the last regular expression used by UNIX. Therefore:

```
To tire bhd nurd    [original line #1]
```

is changed to

```
To tire the nurd    [new line #1]
```

For line #3, the two substitute commands are equivalent to "s/b/e/" and "s/b/t/", noting that the "null" regular expression in the first substitute command is now equivalent to "b", the last regular expression used by UNIX (during line #1 substitutions). Thus:

```
hb nord you bhink \\* should xxxake  [original line #3]
```

becomes

```
he nord you bhink \\* should xxxake
```

after the first substitution and

```
he nord you think \\* should xxxake  [new line #3]
```

after the second substitution.

Wasn't that fun!!!!!!

[2] The command "3s/x*/t/" changes the leftmost occurrence of zero or more consecutive "x's" to a "t", in current line #3. Thus:

he nord you think * should xxxake [current line #3]

is changed to

the nord you think * should xxxake [new line #3]

as UNIX found zero "x's" between the "beginning" of the line (a "new line" character) and the first letter "h", and placed a "t" there. Were you fooled (again)???

[3] The command "1,.2s/n.rd/time/" is equivalent to "1,5s/n.rd/time", since the relative address ".2" is equivalent to ".+2" and "dot" was set to line #3 after the last substitution. Matches to the strings "nurd" in line #1 and "nord" in line #3 are the only ones found in the designated range; the "." in the regular expression represents any single character. Thus:

To tire the nurd [current line #1]

is changed to

To tire the time [new line #1]

and

the nord you think * should xxxake [current line #3]

is changed to

the time you think * should xxxake [new line #3]

That was an easy one!!!

[4] The command "3s/*//" changes

the time you think * should xxxake [current line #3]

to

the time you think * should xxxake [new line #3]

Again, I've tried to trick you. The regular expression "*" represents one or more consecutive "\" characters; remember that "\\" is a literal "backslash" and "*" is a metacharacter.

[5] The command "v/e$/s/4/2/" is straight-forward. The entire buffer is searched for lines not ending in the letter "e", i.e., lines #3, #4, #6 and #7; in these matched lines, the leftmost "4" (if one exists) is changed to a "2". Thus:

multiply by 4, and find the hair [original line #4]

is changed to

multiply by 2, and find the hair [new line #4]

and

locate 4 days for a hour [original line #7]

is changed to

locate 2 days for a hour [new line #7]

Lines #3 and #6 do not contain the string "4", and, hence, were not affected by the substitutions.

[6] The command "g/ti[^m]e/s//estimate/" is also quite straight-forward. The buffer is searched for lines containing the string "ti", followed by any character except "m", followed by "e"; lines #1 and #2 provide matches. In these lines, the first occurrence of the matched strings are replaced by "estimate". Thus:

To tire the time [current line #1]

is changed to

To estimate the time [new line #1]

and

it takes to 4 a task: tile [original line #2]

is changed to

it takes to 4 a task: estimate [original line #2]

[7] The command "g/^[^i]/s/4/highest/" will affect any lines beginning with any character except "i". All lines except #2 qualify, but the replacement of "4" with the string "highest" can only be accomplished in line #6, i.e.,

to the next 4 tibble. Thus we [original line #6]

is changed to

to the next highest tibble. Thus we [new line #6]

[8] The command "v/^t/s/..in..the.*//" can affect any line not beginning with the letter "t"; this includes lines #1, #2, #4, #5 and #7. The only match to the string "..in..the.*" occurs in line #4, i.e., " find the hair", and

multiply by 2, and find the hair [current line #4]

is changed to

multiply by 2, and [new line #4]

[9] This global command seeks lines containing the character ":" or ","; matches are found on lines #2, #3 and #4. On each of those lines, any or all of the substitutions "s/4/do/", "s/*/it/" and "s/xx*/t/" may be made, in that order. Note that the regular expression "*" represents the character "*" literally, and "xx*" matches one or more "x's" literally. Therefore:

it takes to 4 a task: estimate [current line #2]

is changed to

it takes to do a task: estimate [new line #2]

by the first substitute command, and

the time you think * should xxxake [current line #3]

is changed to

the time you think it should take [new line #3]

via the second and third substitute command. No substitutions are possible in line #4.

[10] This global command seeks lines containing the character "2"; lines in the current buffer that match are #4 and #7. In each of those lines, the commands "s/ho/one-&/" and "s/^[^monkey]/al&/" will be attempted; the first command replaces the string "ho" with "one-ho", due to the replacement text metacharacter "&", and the second command will place the string "al" at the beginning of a line not beginning with an "m", "o", "n", "k", "e" or "y", due to the "^" and "&" metacharacters. Therefore:

locate 2 days for a hour [current line #7]

is changed to

allocate 2 days for a one-hour [new line #7]

via both substitute command, and line #4 is unaffected.

[11] The last global command seeks lines containing the string "bb", and lines #5 and #6 qualify. In those lines, a six character string with the two center letters being "b's" (if one exists) is converted to "unit" via the command "s/..bb../unit/". Two lines after each matched line are also (possibly) affected by the command "s/rk/sk/". For line #5, the first substitution is effective, i.e.,

change the tibble of measure [original line #5]

is changed to

change the unit of measure [new line #5]

and, in this case, the second substitution has no effect on line #7 (i.e.,".+2"). The first substitution also has an effect on line #6, i.e.,

to the next highest tibble. Thus we [current line #6]

becomes

to the next highest unit. Thus we [new line #6]

and line #8 (i.e., ".+2") is effected by the second substitution, i.e.,

tark. [original line #8]

is changed to

task. [new line #8]

I knew you'd make it!!!!!

[12] The "w" command is obvious, and a recap of all the changes yields a new file "weird" containing:

To estimate the time
it takes to do a task: estimate
the time you think it should take,
multiple by 2, and
change the unit of measure
to the next highest unit. Thus we
allocate 2 days for a one-hour
task.

Problem 8

Assuming you are in your subdirectory "EXAM", the following editor sessions will create the files requested:

```
$ed                        ["$" is the UNIX command prompt]
a
one
t(CTRL-I)w(CTRL-I)(CTRL-I)o
(CTRL-I)thr(CTRL-I)ee
.
w .when
4                          [character count]
,d                         [empties buffer]
a
two
.
w was/where                [partial pathname]
4                          [character count]
,d                         [empties buffer]
a
If one advarces confidtly
in the directious of his dreams
and edevors to live a
life which he has imagined,
he will meet with sucess
unexeced in cummun huors.
.
w why
158                        [character count]
,d                         [empties buffer]
a
cane
cone
lane
lone
nine
none
.
w was/words
30                         [character count]
q
```

This editor session will only work properly, if the subdirectory "was" is created beforehand, via the command:

```
mkdir was
```

issued in directory "EXAM".

Problem 9

The list commands will yield the following responses, if issued in subdirectory "EXAM":

[1] The command "ls" produces an alphabetized listing of all "non-dot" files/subdirectories in "EXAM", i.e.,

begin, number, was, weird, welcome, why, wisdom

[2] The command "ls was" produces an alphabetized listing of all "non-dot" files in subdirectory "was", i.e.,

where words

[3] The command "ls -a" will yield the same response as "ls", except the list headed by the "dot" files:

```
.       [abbreviation for your current directory "EXAM"]
..      [abbreviation for the parent of "EXAM"]
.when
```

[4] The command "ls -t" will yield the same response as "ls", except the files are put in the order in which they were created or modified, most recent first, i.e.,

was, why, weird, wisdom, welcome, number, begin

[5] The command "ls -l" will list the same files as "ls", except that "long listing" information (which varies from user to user) is provided

[6] The command "ls why where when", noting file "where" is a file in directory "EXAM/was" not "EXAM", and "when" does not exist, produces a printout:

```
where not found
when not found
why
```

[7] The command "ls -d was why" produces a list:

```
was
why
```

noting the "-d" option tells you of the existence of subdirectory "was" (not the list of files contained in it), and has no effect on "why", which is not a directory.

[8] The command "ls -dl EXAM was" will indicate that "EXAM not found", as "EXAM" is not a file/subdirectory within itself, and "long listing" information about the subdirectory "was" is produced (due to the "-dl" option).

[9] The command "ls w*" is equivalent to "ls was weird welcome why wisdom", due to the expansion of the "*" metacharacter. The listing produced is:

```
weird
welcome
why
wisdom

was:
where
words
```

noting the contents of subdirectory "was" is displayed (in the absence of the "-d" option).

[10] The command "ls *h*" expands to "ls why", as "why" is the only "non-dot" file or subdirectory in "EXAM" that contains an "h", and UNIX verifies that the file exists by printing its name.

Problem 10

My solutions to the problem are listed below, but other answers may exist:

[1] To print your working directory, issue the command "pwd", which will yield a response "/your_system/your_login/EXAM".

[2] To change to the "was" subdirectory, issue the command "cd was".

[3] You are in directory "was", whose parent is "EXAM", and the proper command is "cp ../welcome hello" (noting ".." is an abbreviation for the parent of whatever directory you are currently working in).

[4] The proper command is

```
cat where hello >>../begin
```

noting the "cat where hello" portion combines files "where" and "hello", and, instead of printing them on the terminal, they are appended to file "EXAM/begin" (again, ".." is an abbreviation for the parent of "was") via the redirection ">>../begin".

[5] To return to the parent of "was", issue the command "cd ..".

[6] A possible command is

```
pr b* n* w[e-i]* was/*
```

noting the metacharacter expansions are equivalent to the files in "EXAM" beginning with "b", "n" or "w", and all files in subdirectory "was". If "*" was used rather than the string "b* n* w[e-i]*", we would have attempted to print the control information contained in "was", as "was" would have been one component of the metacharacter expansion.

[7] A possible solution is

```
grep -r one .w* w[e-i]* was/*w*
```

(again, trying to avoid "grepping" the directory "was" itself) and the resultant response from UNIX is:

```
.when:1:one
weird:7:allocate 2 days for a one-hour
why:1:If one advarces confidtly
was/words:2:cone
was/words:4:lone
was/words:6:none
```

[8] This task is easily accomplished by redirecting the output of the "word count" command, i.e.,

```
wc weird >count
```

[9] The output of the command "spell why" produces the list:

```
advarces
confidtly
cummun
directious
edevors
huors
sucess
```

missing the spelling error "unexeced". The command "typo why" produces a longer list, including "unexeced" and two correctly spelled words. If spelling errors in "why" were corrected, the quotation would read:

```
If one advances confidently
in the direction of his dreams
and endeavors to live a
life which he has imagined,
he will meet with success
unexceeded in common hours.
```

Problem 11

My solutions (assuming all commands are issued within directory "EXAM") are:

[1] man banner at usend sleep | opr -f x8by11

[2] opr -f 8by11 weird

[3] cat weird welcome | opr

[4] opr -f xerox was/words

[5] pr *h* *l* | opr -f x2on1

Problem 12

My solutions (again assuming all commands are issued within directory "EXAM") are:

[1] My way of handling this problem is the sequence:

```
cat - welcome | pr
HI THERE
(CONTROL-D)
```

recalling that the "-" argument expects input from the terminal, and the (CONTROL-D) terminates that input.

[2] The solution is:

```
ls -d w* >WWW
```

noting the "-d" option is necessary to force the directory name "was" itself to be listed, rather than its conterts.

[3] The command "nohup cat welcome" will cause a copy of the file "welcome" to be placed in a file "nohup.out" in directory "EXAM", which UNIX created (any output destined for the terminal will be redirected to file "nohup.out", if the "nohup" prefix is used).

[4] The multi-command line:

```
ls was; date; cat welcome
```

will provide the desired response

[5] The required command is:

```
spell w[e-i]* >trr
```

where the subdirectory "was" is avoided by judicious use of

metacharacters in the filename list "w[e-i]*".

[6] The solution to this problem requires the "at" command, as follows:

```
at now + 10minutes
date >777
(CONTROL-D)
at job number        [provided by UNIX]

at now + 2hours
opr welcome
(CONTROL-D)
at job number        [provided by UNIX]
```

recalling (CONTROL-D) terminates a particular "at job list", and UNIX provides a unique job number for each "at" command invoked.

[7] To "kill" a pending "at" job, use the following command:

```
atz number_of_job_to_be_run_in_2_hours
```

[8] To mail yourself the required message or file, try the following:

```
mail your_login_name
I love myself
.

mail your_login_name <was/words
```

[9] To read your mail, simply type "mail" with no arguments and respond to the queries by UNIX properly, i.e.,

```
From your_login_name date
{contents of file "was/words"}

? s junk
```

noting your response "s junk" will cause the mailed file headed by a postmark to be stored in file "junk". Then another message is then printed:

```
From your_login_name date
I love myself

? d
```

noting the "d" response destroys the message.

[10] The proper command is:

```
ls /usr/news | grep 2 >NEWS
```

noting "ls /usr/news" produces a list of all "news articles" available, "grep 2" accepts that list from the "pipe" and isolates those names containing the character "2", and those names are placed into file "NEWS" via redirection

Problem 13

My solutions are:

[1] You are in directory "EXAM", and the command:

```
cat .when
```

will yield a printout:

```
one
two
three
```

assuming your terminal does not respond to the "tab" character.

[2] The proper command is:

```
cd    [without a directory argument]
```

[3] The following sequence will accomplish the task:

```
e .profile
(character count of file read)
a
stty -tabs
.
w
(character count of buffer written)
q
```

[4] Hitting (CONTROL-D) will log you off and provide another "login sequence". This action allows ".profile" to be read, and instruct the terminal to react to "tab" characters.

[5] Now the command:

```
cat EXAM/.when
```

will result in a printout, in which "tab" characters are activated, i.e.,

one
t w o
thr ee

[6] This command forces the UNIX command prompt to be "READY ".

[7] The proper command is

```
cd EXAM
```

Problem 14

My solutions are:

[1] Login "abc" must have a subdirectory "rje" in "mode 777" created before the file transfer takes place. The required commands issued in directory "/b2/abc" are:

```
mkdir rje
chmod 777 rje
```

[2] The required "usend" command is:

```
usend -d ihuxb -u abc .when was/words
```

where "ihuxb" is the "destination code" for the UNIX-B system, and the "usend" command was issued in your "EXAM" directory.

[3] The received files are "/b2/abc/rje/.when" and "/b2/abc/rje/words", respectively.

Problem 15

The following scenario (all commands issued in directory "EXAM") is valid, if you've completed the previous questions exactly as instructed:

[1] The command:

```
chmod 200 was
```

will deny all permissions for subdirectory "was", except "write" permission to the owner of the directory.

[2] The command:

```
chmod 000 welcome why
```

will deny all permissions for the files "welcome" and "why".

[3] The command:

```
rm .*
```

will remove all "dot" files in "EXAM", noting you will be advised by UNIX that "." (the current directory "EXAM") is a directory and cannot be removed by "rm" without the "-r" option, and that ".." (the parent of "EXAM") cannot be removed from within the "child".

[4] The following command will change the name of "junk" to "welcome":

```
mv junk welcome
```

noting you will receive a "mode warning" from UNIX of the form "mv:welcome:0 mode", to which a "y" response will force the move. The new file "welcome" will possess the permissions of the old file "junk".

[5] In "EXAM", the following command will set up the "interactive deletion" process:

```
rm -ir .
```

noting "." (an abbreviation for the current directory "EXAM") must be explicitly designated. The interactive dialogue with UNIX is as follows:

```
./begin: y
./number: y
./welcome: y
./wisdom: y
./weird: y
./777: y
directory ./was: (CR)
./why: y
./count: y
./WWW: y
./nohup.out: y
./tmp: y
./NEWS: y
```

noting a "y" response removes the named fil. The (CR) response to the entry request for directory "was" causes that directory to be skipped in the deletion process; had we typed a "y", entry to the directory would have been denied, due to its 200 mode, and the "rm" command would be exited.

[6] The final step is to change the mode of subdirectory "was", return to your "login" directory, and call for the elimination of "EXAM" and its contents, i.e.,

```
chmod 700 was
cd
rm -r EXAM
```

CONGRATULATIONS TO ALL THE MASOCHISTS WHO COMPLETED THIS EXAM!!!!

IH-5613-KEW-kew

K. E. Wendland