Manual of the Thompson Shell on Version 3

released under the license of Caldera

original sh.1 massaged, nroffed and htmlized.

SH (I) 1/15/73 SH (I)

NAME sh -- shell (command interpreter)

SYNOPSIS sh [name [arg ... [arg]]]
1 9

DESCRIPTION

sh is the standard command interpreter. It is the program which reads and arranges the execution of the command lines typed by most users. It may itself be called as a command to interpret files of commands. Before discussing the arguments to the shell used as a command, the structure of command lines themselves will be given.

Command lines

Command lines are sequences of commands separated by command delimiters. Each command is a sequence of non-blank command arguments separated by blanks. The first argument specifies the name of a command to be executed. Except for certain types of special arguments discussed below, the arguments other than the command name are passed without interpretation to the invoked command.

If the first argument is the name of an executable file, it is invoked; otherwise the string "/bin/" is prepended to the argument. (In this way most standard commands, which reside in "/bin", are found.) If no such command is found, the string "/usr" is further prepended (to give "/usr/bin/command") and another attempt is made to execute the resulting file. (Certain "overflow" commands live in "/usr/bin".) If the "/usr/bin" file exists, but is not executable, it is used by the shell as a command file. That is to say it is executed as though it were typed from the console. If all attempts fail, a diagnostic is printed.

The remaining non-special arguments are simply passed to the command without further interpretation by the shell.

Command delimiters

There are three command delimiters: the new-line, ";", and "&". The semicolon ";" specifies sequential execution of the commands so separated; that is,

coma; comb

causes the execution first of command coma, then of comb. The ampersand "&" causes simultaneous execution:

```
coma & comb
```

causes coma to be called, followed immediately by comb without waiting for coma to finish. Thus coma and comb execute simultaneously. As a special case,

```
coma &
```

causes coma to be executed and the shell immediately to request another command without waiting for coma.

Termination Reporting

If a command (not followed by "&") terminates abnormally, a message is printed. (All terminations other than exit and interrupt are considered abnormal.) The following is a list of the abnormal termination messages:

```
Bus error
Trace/BPT trap
Illegal instruction
IOT trap
Power fail trap
EMT trap
Bad system call
Quit
PIR trap
Floating exception
Memory violation
Killed
User I/O
Error
```

If a core image is produced, " -- Core dumped" is appended to the appropriate message.

Redirection of I/O

Three character sequences cause the immediately following string to be interpreted as a special argument to the shell itself, not passed to the command.

An argument of the form "<arg" causes the file arg to be used as the standard input file of the given command.

An argument of the form ">arg" causes file "arg" to be used as the standard output file for the given command. "Arg" is created if it did not exist, and in any case is truncated at the outset.

An argument of the form ">>arg" causes file "arg" to be used as the standard output for the given command. If "arg" did not exist, it is created; if it did exist, the command output is appended to the file.

Pipes and Filters

A pipe is a channel such that information can be written into one end of the pipe by one program, and read at the other end by another program. (See pipe (II)). A filter is a program which reads the standard input file, performs some transformation, and writes the result on the standard output file. By extending the syntax used for redirection of I/O, a command line can specify that the output produced by a command be passed via a pipe through another command which acts as a filter. For example:

```
command >filter>
```

More generally, special arguments of the form

```
>f >f >...>
  1  2
```

specify that output is to be passed successively through the filters f1, f2, ..., and end up on the standard output stream. By saying instead

```
>f >f >...>file
  1  2
```

the output finally ends up in file. (The last ">" could also have been a ">>" to specify concatenation onto the end of file.)

In exactly analogous manner input filtering can be specified via one of

```
<f <f <...<         <f <f <...<file
  1  2                1  2
```

Both input and output filtering can be specified in the same command, though not in the same special argument.

For example:

```
ls >pr>
```

produces a listing of the current directory with page headings, while

```
ls >pr>xx
```

puts the paginated listing into the file xx.

If any of the filters needs arguments, quotes can be used to prevent the required blank characters from violating the blankless syntax of filters. For example:

```
ls >"pr -h 'My directory'">
```

uses quotes twice, once to protect the entire pr command, once to protect the heading argument of pr. (Quotes are discussed fully below.)

Generation of argument lists

If any argument contains any of the characters "?", "*" or '[', it is treated specially as follows. The current directory is searched for files which match the given argument.

The character "*" in an argument matches any string of characters in a file name (including the null string).

The character "?" matches any single character in a file name.

Square brackets "[...]" specify a class of characters which matches any single file-name character in the class. Within the brackets, each ordinary character is taken to be a member of the class. A pair of characters separated by "-" places in the class each character lexically greater than or equal to the first and less than or equal to the second member of the pair.

Other characters match only the same character in the file name.

For example, "*" matches all file names; "?" matches all one-character file names; "[ab]*.s" matches all file names beginning with "a" or "b" and ending with ".s"; "?[zi-m]" matches all two-character file names ending with "z" or the letters "i" through "m".

If the argument with "*" or "?" also contains a "/", a slightly different procedure is used: instead of the current directory, the directory used is the one obtained by taking the argument up to the last "/" before a "*" or "?". The matching process matches the remainder of the argument after this "/" against the files in the derived directory. For example: "/usr/dmr/a*.s" matches all files in directory "/usr/dmr" which begin with "a" and end with ".s".

In any event, a list of names is obtained which match the argument. This list is sorted into alphabetical order, and the resulting sequence of arguments replaces the single argument containing the "*", "[", or "?". The same process is carried out for each argument (the resulting lists are not merged) and finally the command is called with the resulting list of arguments.

For example: directory /usr/dmr contains the files a1.s, a2.s, ..., a9.s. From any directory, the command

```
as /usr/dmr/a?.s
```

calls as with arguments /usr/dmr/a1.s, /usr/dmr/a2.s, ... /usr/dmr/a9.s in that order.

Quoting

The character "\" causes the immediately following character to lose any special meaning it may have to the

shell; in this way "<", ">", and other characters meaningful to the shell may be passed as part of arguments. A special case of this feature allows the continuation of commands onto more than one line: a new-line preceded by "\" is translated into a blank.

Sequences of characters enclosed in double (") or single (') quotes are also taken literally.

Argument passing

When the shell is invoked as a command, it has additional string processing capabilities. Recall that the form in which the shell is invoked is

```
sh [ name [ arg  ... [ arg  ] ] ]
                 1          9
```

The name is the name of a file which will be read and interpreted. If not given, this subinstance of the shell will continue to read the standard input file.

In command lines in the file (not in command input), character sequences of the form "$n", where n is a digit 0, ..., 9, are replaced by the nth argument to the invocation of the shell (arg). "$0" is replaced by name.
n

End of file

An end-of-file in the shell's input causes it to exit. A side effect of this fact means that the way to log out from UNIX is to type an end of file.

Special commands

Two commands are treated specially by the shell.

"Chdir" is done without spawning a new process by executing the sys chdir primitive.

"Login" is done by executing /bin/login without creating a new process.

These peculiarities are inexorably imposed upon the shell by the basic structure of the UNIX process control system. It is a rewarding exercise to work out why.

Command file errors; interrupts

Any shell-detected error, or an interrupt signal, during the execution of a command file causes the shell to cease execution of that file.

FILES /etc/glob, which interprets "*", "?", and "[".

SEE ALSO "The UNIX Time-sharing System", which gives the theory of operation of the shell.

DIAGNOSTICS

"Input not found", when a command file is specified which

cannot be read;
"Arg count", if the number of arguments to the chdir pseudo-command is not exactly 1, or if "*", "?", or "[" is used inappropriately;
"Bad directory", if the directory given in "chdir" cannot be switched to;
"Try again", if no new process can be created to execute the specified command;
""' imbalance", if single or double quotes are not matched;
"Input file", if an argument after "<" cannot be read;
"Output file", if an argument after ">" or ">>" cannot be written (or created);
"Command not found", if the specified command cannot be executed.
"No match", if no arguments are generated for a command which contains "*", "?", or "[".
Termination messages described above.

BUGS

If any argument contains a quoted "*", "?", or "[", then all instances of these characters must be quoted. This is because sh calls the glob routine whenever an unquoted "*", "?", or "[" is noticed; the fact that other instances of these characters occurred quoted is not noticed by glob.

When output is redirected, particularly through a filter, diagnostics tend to be sent down the pipe and are sometimes lost altogether.