NAME

tsh - Thompson shell (command interpreter)

SYNOPSIS

tsh [**-** | **-c** [*string*] | **-t** | *file* [*arg1 ...*]]

DESCRIPTION

Tsh is a port of the standard command interpreter from Version 6 (V6) UNIX. It may be used either as an interactive shell or as a non-interactive shell. Throughout this manual, '(+)' indicates those cases where **tsh** is known to differ from the original *sh*(1), as it appeared in Version 6 (V6) UNIX.

The options are as follows:

- The shell reads and executes command lines from the standard input until end-of-file or **exit**.

-c [*string*]
If a *string* is specified, the shell executes it as a command line and exits. Otherwise, the shell treats it as the - option.

-t The shell reads a single line from the standard input, executes it as a command line, and exits.

The shell may also be invoked non-interactively to read, interpret, and execute an ASCII command file. The specified *file* and any arguments are treated as positional parameters (see *Parameter substitution* below) during execution of the command file.

Otherwise, if no arguments are specified and if both the standard input and standard error are connected to a terminal, the shell is interactive. An interactive shell prompts a regular user with a '% ' or with a '# ' for the superuser before reading each command line from the terminal.

Metacharacters

A *syntactic metacharacter* is any one of the following:

| ^ ; & ( ) < > space tab

When such a character is unquoted, it has special meaning to the shell. The shell uses it to separate words (see *Commands* and *Command lines* below). A *quoting metacharacter* is any one of the following:

" ' \

See *Quoting* below. The *substitution metacharacter* is a:

$

See *Parameter substitution* below. Finally, a *pattern metacharacter* is any one of the following:

*** ? [**

See *File name generation* below.

Commands

Each command is a sequence of non-blank command arguments, or words, separated by one or more blanks (**spaces** or **tabs**). The first argument specifies the name of a command to be executed. Except for certain special arguments described below, the arguments other than the command name are passed without interpretation to the invoked command.

If the first argument names a special command, the shell executes it (see *Special commands* below). Otherwise, the shell treats it as an external utility or command, which is located as follows.

(+) **tsh** expects to find its external utilities (**glob**, **if**, **goto**, and **fd2**) in the */usr/local/libexec/etsh-5.4.0/tsh* directory, not by searching the environment variable PATH. Notice that these external utilities are special to the shell and are required for full functionality.

(+) Otherwise, if the command name contains no '/' characters, the sequence of directories in the environment variable PATH is searched for the first occurrence of an executable file by that name, which the shell attempts to execute. However, if the command name contains one or more '/' characters, the shell attempts to execute it without performing any PATH search.

(+) If an executable file does not begin with the proper magic number or a '#!shell' sequence, it is assumed to be an ASCII command file, and a new shell is automatically invoked to execute it. The environment variable EXECSHELL specifies the shell which is invoked to execute such a file.

If a command cannot be found or executed, a diagnostic is printed.

Command lines

Commands separated by **|** or **^** constitute a chain of *filters*, or a *pipeline*. The standard output of each command but the last is taken as the standard input of the next command. Each command is run as a separate process, connected by pipes (see *pipe*(2)) to its neighbors.

A *command line*, or *list*, consists of one or more pipelines separated, and perhaps terminated by **;** or **&**.

The semicolon designates sequential execution. The ampersand designates asynchronous execution, which causes the preceding pipeline to be executed without waiting for it to finish. The process ID of each command in such a pipeline is reported, so that it may be used if necessary for a subsequent *kill*(1).

A list contained within parentheses such as **(** *list* **)** is executed in a subshell and may appear in place of a simple command as a filter.

If a command line is syntactically incorrect, a diagnostic is printed.

Termination reporting

All terminations other than exit and interrupt are considered to be abnormal. If a sequential process terminates abnormally, a message is printed. The termination report for an asynchronous process is given upon execution of the first sequential command subsequent to its termination, or when the **wait** special command is executed. The following is a list of the possible termination messages:

Hangup
Quit
Illegal instruction
Trace/BPT trap
IOT trap
EMT trap
Floating exception
Killed
Bus error
Memory fault
Bad system call
Broken pipe

For an asynchronous process, its process ID is prepended to the appropriate message. If a core image is produced, ' -- Core dumped' is appended to the appropriate message.

I/O redirection

Each of the following argument forms is interpreted as a *redirection* by the shell itself. Such a redirection may appear anywhere among the arguments of a simple command, or before or after a parenthesized command list, and is associated with that command or command list.

A redirection of the form **<***arg* causes the file *arg* to be used as the standard input (file descriptor 0) for the associated command.

A redirection of the form >*arg* causes the file *arg* to be used as the standard output (file descriptor 1) for the associated command. If *arg* does not already exist, it is created; otherwise, it is truncated at the outset.

A redirection of the form >>*arg* is the same as >*arg*, except if *arg* already exists the command output is always appended to the end of the file.

For example, either of the following command lines:

```
% date >index.txt ; pwd >>index.txt ; ls -l >>index.txt
% ( date ; pwd ; ls -l ) >index.txt
```

creates on the file 'index.txt', the current date and time, followed by the name and a long listing of the current working directory.

A >*arg* or >>*arg* redirection associated with any but the last command of a pipeline is ineffectual, as is a <*arg* redirection with any but the first.

The standard error (file descriptor 2) is never subject to redirection by the shell itself. Thus, commands may write diagnostics to a location where they have a chance to be seen. However, *fd2*(1) provides a way to redirect the diagnostic output to another location.

If the file for a redirection cannot be opened or created, a diagnostic is printed.

Quoting

The shell treats all *quoted* characters literally, including characters which have special meaning to the shell (see *Metacharacters* above). If such characters are quoted, they represent themselves and may be passed as part of arguments.

Individual characters, and sequences of characters, are quoted when enclosed by a matched pair of *double* (**"**) or *single* (**'**) quotes. For example:

```
% awk '{ print NR "\t" $0 }' README ^ more
```

causes *awk*(1) to write each line from the 'README' file, preceded by its line number and a tab, to the standard output which is piped to *more*(1) for viewing. The outer single quotes prevent the shell from trying to interpret any part of the string, which is then passed as a single argument to awk.

An individual *backslash* (\) quotes, or *escapes*, the next individual character. A backslash followed by a newline is a special case which allows continuation of command-line input onto the next line. Each backslash-newline sequence in the input is translated into a blank.

If a double or single quote appears but is not part of a matched pair, a diagnostic is printed.

Parameter substitution

When the shell is invoked as a non-interactive command, it has additional string processing capabilities which are not available when it is interactive. A non-interactive shell may be invoked as follows:

> **tsh** *name* [*arg1 ...*]

If the first character of *name* is not **-**, it is taken as the name of an ASCII *command file*, or *shell script*, which is opened as the standard input for a new shell instance. Thus, the new shell reads and interprets command lines from the named file.

Otherwise, *name* is taken as one of the shell options, and a new shell instance is invoked to read and interpret command lines from its standard input. However, notice that the **-c** option followed by a *string* is the one case where the shell does not read and interpret command lines from its standard input. Instead, the string itself is taken as a command line and executed.

In each command line, an unquoted character sequence of the form **$***N*, where *N* is a digit, is treated as a *positional parameter* by the shell. Each occurrence of a positional parameter in the command line is substituted with the value of the *N*th argument to the invocation of the shell (*argN*). **$***0* is substituted with *name*.

In both interactive and non-interactive shells, **$$** is substituted with the process ID of the current shell. The value is represented as a 5-digit ASCII string, padded on the left with zeros when the process ID is less than 10000.

All substitution on a command line is performed *before* the line is interpreted. Thus, no action which alters the value of any parameter can have any effect on a reference to that parameter occurring on the *same* line.

A positional-parameter value may contain any number of metacharacters. Each one which is *unquoted*, or *unescaped*, within a positional-parameter value retains its special meaning when the value is substituted in a command line by the invoked shell.

Take the following two shell invocations for example:

```
% tsh -c '$1' 'echo Hello World! >/dev/null'
% tsh -c '$1' 'echo Hello World! \>/dev/null'
```

Hello World! >/dev/null

In the first invocation, the **>** in the value substituted by **$***1* retains its special meaning. This causes all output from *echo*(1) to be redirected to */dev/null*. However, in the second invocation, the meaning of **>** is *escaped* by \ in the value substituted by **$***1*. This causes the shell to pass ‘>/dev/null’ as a single argument to echo instead of interpreting it as a redirection.

File name generation

Prior to executing an external command, the shell scans each argument for unquoted *****, **?**, or **[** characters. If one or more of these characters appears, the argument is treated as a *pattern*, and the shell uses *glob*(1) to search for file names which *match* it. Otherwise, the argument is used as is.

The meaning of each pattern character is as follows:

- The ***** character in a pattern matches any string of characters in a file name (including the null string).
- The **?** character in a pattern matches any single character in a file name.
- The **[...]** brackets in a pattern specifies a class of characters which matches any single file-name character in the class. Within the brackets, each character is taken to be a member of the class. A pair of characters separated by an unquoted - specifies the class as a range which matches each character lexically between the first and second member of the pair, inclusive. A - matches itself when quoted or when first or last in the class.

Any other character in a pattern matches itself in a file name.

Notice that the ‘.’ character at the beginning of a file name, or immediately following a ‘/’, is always special in that it must be matched explicitly. The same is true of the ‘/’ character itself.

If the pattern contains no ‘/’ characters, the current directory is always used. Otherwise, the specified directory is the one obtained by taking the pattern up to the last ‘/’ before the first unquoted *****, **?**, or **[**. The matching process matches the remainder of the pattern after this ‘/’ against the files in the specified directory.

In any event, a list of file names is obtained from the current (or specified) directory which match the given pattern. This list is sorted in ascending ASCII order, and the new sequence of arguments replaces the given pattern. The same process is carried out for each of the given pattern arguments; the resulting lists are *not* merged. Finally, the shell attempts to execute the command with the resulting argument list.

If a pattern argument refers to a directory which cannot be opened, a 'No directory' diagnostic is printed.

If a command has only *one* pattern argument, a 'No match' diagnostic is printed if it fails to match any files. However, if a command has more than one pattern argument, a diagnostic is printed only when they *all* fail to match any files. Otherwise, each pattern argument failing to match any files is removed from the argument list.

End of file

An end-of-file in the shell's input causes it to exit. If the shell is interactive, this means it exits by default when the user types an EOT (^D) at the prompt. If desired, the user may change or disable interactive shell EOT exit behavior with *stty*(1).

Special commands

The following commands are special in that they are executed by the shell without creating a new process.

: [*arg* ...]
: Does nothing and sets the exit status to zero.

chdir *dir* [...]
: Changes the shell's current working directory to *dir*.

exit Causes the shell to cease execution of a file. This means exit has no effect at the prompt of an interactive shell.

login [*arg* ...]
: Replaces the current interactive shell with *login*(1).

newgrp [*arg* ...]
: Replaces the current interactive shell with *newgrp*(1).

shift
: Shifts all positional-parameter values to the left by 1, so that the old value of **$***2* becomes the new value of **$***1* and so forth. The value of **$***0* does not shift.

wait
: Waits for all asynchronous processes to terminate, reporting on abnormal terminations.

Signals (+)

If the shell is interactive, it ignores the SIGINT, SIGQUIT, and SIGTERM signals (see *signal*(3)). However, if the shell is invoked with any option argument, it only ignores SIGINT and SIGQUIT.

If SIGINT, SIGQUIT, or SIGTERM is already ignored when the shell starts, it is also ignored by the current shell and all of its child processes. Otherwise, SIGINT and SIGQUIT are reset to the default action for sequential child processes, whereas SIGTERM is reset to the default action for all child processes.

For any signal not mentioned above, the shell inherits the signal action (default or ignore) from its parent process and passes it to its child processes.

Asynchronous child processes always ignore both SIGINT and SIGQUIT. Also, if such a process has not redirected its input with a **<**, **|**, or **^**, the shell automatically redirects it to come from */dev/null*.

EXIT STATUS (+)

The exit status of the shell is generally that of the last command executed prior to end-of-file or **exit**.

However, if the shell is interactive and detects an error, it exits with a non-zero status if the user types an EOT at the next prompt.

Otherwise, if the shell is non-interactive and is reading commands from a file, any shell-detected error causes the shell to cease execution of that file. This results in a non-zero exit status.

A non-zero exit status returned by the shell itself is always one of the values described in the following list, each of which may be accompanied by an appropriate diagnostic:

2 The shell detected a syntax, redirection, or other error not described in this list.

125 An external command was found but did not begin with the proper magic number or a '#!shell' sequence, and a valid shell was not specified by EXECSHELL with which to execute it.

126 An external command was found but could not be executed.

127 An external command was not found.

>128
An external command was terminated by a signal.

ENVIRONMENT (+)

Notice that the concept of 'user environment' was not defined in Version 6 (V6) UNIX. Thus, use of the following environment variables by this port of the shell is an enhancement:

EXECSHELL

If set to a non-empty string, the value of this variable is taken as the path name of the shell which is invoked to execute an external command when it does not begin with the proper magic number or a '#!shell' sequence.

PATH

If set to a non-empty string, the value of this variable is taken as the sequence of directories used by the shell to search for external commands. Notice that the Version 6 (V6) UNIX shell always used the equivalent of '.:/bin:/usr/bin', not PATH.

TES

The shell sets TES in the environment in order to convey the current tsh exit status to the user as an unsigned decimal integer. The exit status was always here before. It was simply not conveyed directly, via '$?' for example, requiring the user to find another way to see it.

> NOTE: '$?' is not available here now, as it would be incompatible with the original V6 Thompson shell.

FILES

/dev/null

default source of input for asynchronous processes

SEE ALSO

awk(1), echo(1), env(1), etsh(1), expr(1), fd2(1), glob(1), goto(1), grep(1), if(1), kill(1), login(1), newgrp(1), stty(1)

Etsh home page: https://etsh.nl/

'The UNIX Time-Sharing System' (CACM, July, 1974):

> https://etsh.nl/history/unix/

gives the theory of operation of both the system and the shell.

HISTORY

A **sh** command appeared as */bin/sh* in Version 1 (V1) UNIX.

The Thompson shell was used as the standard command interpreter through Version 6 (V6) UNIX.

Then, in Version 7 (V7) UNIX, it was replaced by the Bourne shell. However, the Thompson shell was still distributed with the system as **osh** because of known portability problems with the Bourne shell's memory management in Version 7 (V7) UNIX.

AUTHORS

This port of the Thompson shell is derived from Version 6 (V6) UNIX /usr/source/s2/sh.c, which was principally written by Ken Thompson of Bell Labs. Jeffrey Allen Neitzel <*jan@etsh.nl*> ported and maintains it as *tsh*(1).

LICENSE

See either the LICENSE file which is distributed with **etsh** or https://etsh.nl/license/ for full details.

CAVEATS

Since **tsh** does not read any startup files, it should not be added to the shell database (see *shells*(5)) unless the system administrator is willing to deal with this fact.

Tsh has no facilities for setting, unsetting, or otherwise manipulating environment variables within the shell. This must be accomplished by using other tools such as *env*(1).

Like the original, **tsh** is not 8-bit clean as it uses the high-order bit of characters for quoting. Thus, the only complete character set it can handle is 7-bit ASCII.

Notice that certain shell oddities were historically undocumented in this manual page. Particularly noteworthy is the fact that there is no such thing as a usage error. Thus, the following shell invocations are perfectly valid:

```
tsh -cats_are_nice!!! ': "Good kitty =)"'
tsh -tabbies_are_too!
tsh -s
```

The first two cases correspond to the **-c** and **-t** options respectively; the third case corresponds to the **-** option.

BUGS

The shell makes no attempt to recover from *read*(2) errors and exits if this system call fails.

SECURITY CONSIDERATIONS

This port of the Thompson shell does not support set-ID execution. If the effective user (group) ID of the shell process is not equal to its real user (group) ID when an instance of it starts up, the shell prints the following diagnostic and exits with a non-zero status.

Set-ID execution denied

If the shell did support set-ID execution, it could possibly allow a user to violate the security policy on a host where the shell is used. For example, if the shell were running a setuid-root command file, a regular user could possibly invoke an interactive root shell as a result.

This is *not* a bug. It is simply how the shell works. Thus, **tsh** does not support set-ID execution. This is a proactive measure to avoid problems, nothing more.