CSE 410: Systems Programming

The UNIX Shell

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo

The UNIX Shell

The UNIX shell was historically the user's primary interface to the system.

It provides direct but safe access to many of the system calls.

The shell was rather revolutionary when introduced [3, 2].

You don't need to learn shell programming for this course. Understanding something about the shell will be helpful.

The Interactive Shell

The shell has a dual nature:

- An interactive command prompt
- A programming environment

Interactive sessions prompt for input and execute immediately.

Modern shells include interactive facilities for:

- Command aliasing
- Recall and modification of recent commands

However, the entire programming language is also available interactively!

The Shell as a Programming Environment

The shell programming language contains:

- Variables
- Conditionals
- Loops
- Procedures
- Exceptions

The primary feature remains execution of other programs.

Shell “programs” are usually combinations of external programs.

Words

The shell breaks its input up into words, which are strings.
(Everything in the shell is a string!)

Words are separated by whitespace.

The first word[1] in a command tells the shell what to do with it.

Words can contain whitespace if it is quoted with either:

- Single quotes: '
- Double quotes: "
- Backslash: \

[1] …or sometimes two.

Statements

A single statement:

- Starts after the previous command
- Ends with: newline, `;`, `&`

After parsing a statement, the shell will determine if it is:

- A variable assignment (possibly with a command)
- A builtin command
- A control statement (`if`, `while`, *etc.*)
- An external program or programs to be run

Variables

Shell variables are strings.

Variables need not be declared and are global.[2]

You can create or assign a variable with =:
`VAR=value`

This will:

- Create a variable named `VAR` if it does not exist
- Assign the value “value” to `VAR`

Note that there must be no space around the = symbol!

[2] Many modern shells have an extension for local variables.

Builtin Commands

Certain “commands” are shell builtin commands.

The shell does not execute an external program, it runs internal code for these commands.

There are several possible reasons:

- Efficiency
- The shell’s internal state must be changed
- The statement is a control flow construct

In particular, changing internal state cannot be done after fork.

Therefore, commands like `cd` must be builtin commands.

Control Statements

The shell has control statements that affect program flow:

- Conditional statements and operators (`if`, `case`, `&&`, `||`)
- Loops (`for`, `while`, `until`)

These statements allow the shell to implement program logic.

These statements make their decisions based on command exit statuses.

External Commands

Any other statements are external commands.

The shell will `fork()` and then `exec()` the external commands.

The first word on the line is the binary to execute.

The remaining words are arguments to that binary.

Variable Interpolation

Variables are interpolated into words.

The contents of variables can create new words.

Interpolation takes one of two basic forms:

- `$VAR`: Interpolate the simple variable named `VAR`
- `${VAR}`: Interpolate the variable named `VAR`, which might have a "complicated" name or perform some extra actions

Unless the word containing a variable interpolation is quoted:

- Variables may create new words
- The variable `IFS` will be used to determine how
  (Don't worry about IFS yet.)

Command Interpolation

The output of a command can also be inserted into a command.

The POSIX syntax `$(command)`:

- Runs the command between parenthesis
- Inserts its output into the command in place of the `$()`

The older Bourne syntax `` `command` `` does the same, but:

- Cannot be nested
- Has some strange quoting rules

Globbing

The shell performs globbing, or pattern matching of filenames.

A glob will be expanded to a list of one or more filenames if it matches any such filenames.

The basic glob matching tools are:

- * matches any sequence of 0 or more characters
- ? matches any one character
- [] matches any character between the braces; ranges of characters can be represented as, *e.g.*, [a-z], which matches any lowercase letter

Pipes and Redirection

The file descriptors of the shell itself and the processes it executes can be manipulated.

- Pipes can be created (using `pipe()`) with `|`
- Files can be opened on file descriptors (using `dup2()`) with `<`, `<<`, `>`, and `>>`
- File descriptors can be copied (using `dup2`) with `>&`

Shell versus Environment Variables

Every process on a POSIX system has an environment.

The environment is a set of key-value pairs.

By default, a process inherits a copy of its parent's environment.

The shell allows shell variables to be placed in the environment.

The shell builtin command `export` accomplishes this.

Unless a variable is exported it is private to the shell.

The Environment

The syntax for `export` is:
`export VAR [VAR2 ...]`

Every variable named as an argument to `export` will be copied into the environment for child processes.

The `env` command will print its environment and exit.

The shell uses `setenv()` or `putenv()` to manipulate its environment.

Special Variables

The shell recognizes quite a few special variables, including:

- `$0`: the name of the current executable
- `$1`-`$9`: the first 9 arguments to the shell (or a function)
- `$#`: The number of arguments `$1`-`$9` that are valid
- `$*` and `$@`: All of the arguments to the shell (or a function)
- `$?`: The return value of the previous command
- `$!`: The process ID of the previous command[3]
- `$PS1`: The prompt given in interactive use
- `$IFS`: The input field separator used to determine if an expansion creates new words

[3] sometimes…

IFS

The input field separator is used by the shell to determine when any expansion (variable or other) should create new words.

If an expansion contains characters in `$IFS`, they split the word.

The default value of IFS is newline, tab, and space.

This means that the following command will have two arguments:

```
$ VAR="arg1 arg2"
$ ./writeargs $VAR
```

```
./writeargs
arg1
arg2
```

Simple File Redirections

Standard input, output, and error can be redirected simply.

- `< file` will connect standard input to the named file
- `> file` will do the same for standard output
- `2> file` will redirect standard error

The final syntax is general; `N>` and `N<` connect the named file to file descriptor N using `dup2()`.

To append to a redirected output, use `>>`.

These operators are placed within or after a command.

Using Redirections

To cause `wc -w` to read from `/usr/share/dict/words`:

```
wc -w < /usr/share/dict/words
```

To send the output of `cut` to `totals.txt`:

```
cut -d' ' -f5 > totals.txt
```

To put the output of two different commands into `means.txt`:

```
stats -bmean variant-a.txt > means.txt
stats -bmean variant-b.txt >> means.txt
```

Here Documents

Standard input can be redirected from a here document.

A here document is a file embedded in a shell script.

Here documents use the syntax `<<word`, and the document contains everything from the end of the command to a line with `word` by itself.

```
cat <<EOF
For example, all of this up until the word EOF
on a line by itself will be readable by cat on
its standard input file descriptor.
EOF
```

Pipes

A pipeline may be the most powerful feature of the shell.

A pipeline is a series of commands connected by pipes.

Each command:

- writes to standard output
- reads from standard input

The shell uses `pipe()` and `dup2()` to connect one to the other.

The vertical bar (`|`), often called pipe, accomplishes this.

Using Pipes

A pipeline is built by putting | between commands:
`cmd1 | cmd1`

This will:

- Create a pipe with `pipe()`
- Fork twice (once for `cmd1` and once for `cmd2`)
- Use `dup2()` to connect:
  - `pipefd[1]` to file descriptor 1 (standard output) of `cmd1`
  - `pipefd[0]` to file descriptor 0 (standard input) of `cmd2`
- Call `exec()` appropriately in each child
- Wait for `cmd2` to exit

Duplicating Descriptors

The shell can duplicate descriptors without opening new files.

The operator `N>&M` does this, and it means: `dup2(N, M)`

Thus, to print an error message to standard error:

```
echo Could not open file 1>&2
```

The special syntax `N>&-` or `<&-` closes a descriptor.

This is sometimes used to detach a process from the terminal.

Redirections are processed in order:
duplicating a redirected file must occur after the redirection.

```
echo No output or errors > /dev/null 2>&1
```

Globbing

Unquoted words are subject to globbing.

If they contain certain characters, they will be used as patterns that match filenames.

The single globbing word will be replaced with one word for reach matching file.

If no files match, the glob will be passed unchanged.

Globbing Syntax

Everyone is familiar with the bare *.

It is a glob that means:
all files with zero or more characters in their filenames.

It can be combined with other globs or characters: `*.c`

The character ? matches any one character: `*.?`
(All files with a one-character extension)

A range of characters can be matched with `[]`:
`*.[ch]`: All files ending in `.c` or `.h`
`variant-[a-d].pdf`

Globs can appear anywhere in a path: `lectures/*/*.pdf`

Shell Control Structures

The shell control structures share behaviors:

- Except for `case`, the condition is the exit value of a command.
- Strange ALGOL68-style syntax: `if`/`fi`, `case`/`esac`, `do`/`done`
- Usable in pipelines

Conditions: if/then

```
if condition; then
    commands
elsif condition; then
    commands
else
    commands
fi
```

Each of the conditions is a command.

The `test` command is common here!

Conditions: case

```
case word in
[g]lob?)
    commands
    ;;
*)
    commands
    ;;
esac
```

The case structure matches a word against a pattern.

The pattern uses globbing rules.

Conditions: boolean

```
command1 && command2
command1 || command2
```

These are equivalent to:

```
if command1; then
    command2
fi

if ! command1; then
    command2
fi
```

Loops: while

```
while condition; do
    commands
done
```

The command specified as a condition will be executed repeatedly.

As long as it returns success, the body commands will be executed.

Loops: for

```
for variable in words; do
    commands
done
```

The shell `for` is an iterator-style loop.

The specified variable name will be assigned to each given word in turn, and the body commands executed.

Summary

- The shell almost directly exposes several system calls:
  - `fork()`/`exec()`
  - `open()`
  - `close()`
  - `dup2()`
  - `wait()`
- It provides both interactive and programmatic facilities.
- Your project is similar to but different from the POSIX shell.
- This only scratches the surface of the POSIX shell.

Next Time …

- …

References I

Optional Readings

[1] Bruce Blinn. *Portable Shell Programming. An Extensive Collection of Bourne Shell Examples*. Prentice Hall PTR, 1996.

[2] Stephen R. Bourne. "The Unix Shell". In: *Byte Magazine* 8.1 (Oct. 1983), pp. 187–204. url: https://archive.org/stream/byte-magazine-1983-10/1983_10_BYTE_08-10_UNIX#page/n187/mode/1up.

[3] D. M. Ritchie and K. Thompson. "The UNIX Time-Sharing System". In: *Communications of the ACM* 17.7 (July 1974), pp. 365–375. url: https://www.bell-labs.com/usr/dmr/www/cacm.pdf.

License

To retrieve a copy of this material, or related materials, see https://www.cse.buffalo.edu/~eblanton/.