Shell–Bash

Introduction to the Bash Shell

Kurt Schmidt
Shell as a User Interface

- A shell is a command interpreter
- Interface between a human (or another program) and the OS
  - Runs a program (say, `ls`, or a Solitaire game or Web browser)
  - Can establish alternative sources of input and destinations for output of programs
- Is, itself, just another program
Shell as a Scripting Language

Has features commonly found in languages for structured programs

- Allow shell scripts to be used as filters
- Control flow, variables
- Control over all I/O file descriptors
- Control over signal handling
- The environment allows context to be established at startup
  - Provides a way for scripts to pass information to processes w/out using positional parameters
We’ll teach Bash in this course

- Extension of the Bourne Shell
- Contains many of the Korn Shell (ksh) extensions
- There are other shells: tcsh (Tenex C Shell), ksh (Korn Shell), zsh, dash
The shell supports various customisations

Set through shell options or environment variables

- User prompt
- Bindings for command-line editing
- Aliases (shortcuts)
- Functions – like little scripts\(^1\)
- Other behaviors

\(^1\)But they run in the *current* shell
Place customisations in *startup* files

- /etc/profile – system-wide
- /etc/bash.bashrc – system-wide
- ~/.bash_profile – user
- ~/.bashrc – user

Read the Bash manpages to see when each is invoked
Bash `set` Command

- The `set` builtin with no args displays all shell variables and functions
- Can be used to set various options. E.g.,
  - `-o noclobber` – Won’t let re-direct overwrite an existing file
  - `-o ignoreeof` – Shell won’t exit on `^D`
  - `-o vi` – Use vi-like keybindings for editing the command line. `emacs` is the default
  - `-n` – Dry run. Just parse, but don’t execute. Handy for debugging scripts
  - `-x` – Echo on. Shows commands in script as they execute
Interpreting Commands

- Shell prints a prompt, awaits a command
- When the shell gets a line of input
  1. It expands aliases (recursively)
  2. Checks to see if command is a shell *builtin* (or a function)
  3. If not, assumes it is a program (or script) on disk (e.g., `ls`)
A *shell builtin* is a command the shell will do for you

- `cd`, `type`, `pushd`, `set`, `pwd`, ...

They are faster

The shell provides builtins for some common disk utilities

- `echo`, `printf`, `test`
  - Use a path to invoke the disk utility (`/bin/echo`)

The builtin `type` will determine if a command is a builtin, or tell you where the utility is on disk

The `help` builtin will give you help on any builtin, or show you all of the shell builtins
Running Programs from Disk

- Disk programs are run in a subshell
- The subshell **execs** the program
  - Replaces itself with the program
- If the command isn’t a shell builtin, the shell will search for a disk utility (using your `$PATH`)
- If the command token contains a path, then that utility will simply be run

```
$ /usr/bin/firefox & # kick firefox off in the background
$ /usr/bin/python myScript.py # invoke the python interpreter
$ ~/bin/cow-sample # Invoke my script to see cows
$ ./a.out # run a program I just compiled, in this directory
```
Logging Off

Use the `exit` builtin

- Exits the shell
- If it is the login (top-level) shell, then it disconnects you
- A shell is just another program
- Can recursively invoke shells
- Don’t just disconnect w/out exiting
- `ctrl-D` (end-of-file) will also log you out
  - Unless you have the `ignoreeof` shell option set
Standard I/O

- Shell manages I/O
  - Programs and scripts run in a subshell
- The shell establishes 3 I/O channels:
  - stdin, file descriptor 0, default is the keyboard
  - stdout, file descriptor 1, default is the screen
  - stderr, file descriptor 2, default is the screen
- These streams may be redirected to or from another file
- Can also be redirected to or from another process
Terminating Input

- `stdin` is read like any other file
- If `stdin` is the keyboard, use Ctrl-D (\^D) to signal EOF
- Many utilities, filters, will read `stdin` if not given a filename(s) to open
  - `cat` `head` `grep` `awk` `sort` ...
- If it appears a program “isn’t doing anything”, it’s possible that it’s waiting on you

```bash
$ grep the  # no filename
What’s this?
Is this the line?
Is this the line?
That’s not funny.
Maybe there should be a law
Maybe there should be a law
\^D  # ctrl-D, EOF
```

\(^1\)Sorta. You can’t back up
Shell Metacharacters

A *metacharacter* is a character which has special meaning to the shell.

Here are some:

- **Wildcards**
  - * ? [ ]

- **I/O redirection**
  - < > |

- **Others**
  - & ; $ # ! \ ( ) " '

These characters must be escaped or quoted to inhibit their special behavior.

```bash
$ ls "some file" another\&file 'and;yet;a;third'
some file another&file and;yet;a;third
```
Wildcards

Also known as *name globbing* and *pattern matching*; used in *filename expansion*

- `*` – matches 0 or more characters
- `?` – matches exactly 1 character
- `[list]` – matches any single character from `list`
- Wildcards are *not* regular expressions

```
ls *.cc   list all C++ source files in directory
ls ?a*    list all files whose 2nd letter is ’a’
ls [a-cf]*.jpeg  list all JPEGs that start with a, b, c, or f
ls [!ac-e]*.jpeg  list all JPEGs that do *not* start with a, c, d, e
ls *      Try it with non-empty subdirectories present
```
Shell Variables

Called *parameters*

- Bash uses shell variables to store information
- Used to affect the behavior of the shell, and other programs
- Simple mechanism, just stores text
- Bash does have arrays and associative arrays (see *declare builtin*)
Setting & Viewing Parameters

- To assign a variable (in sh, ksh, bash)
  - Note, *no* whitespace around the =

```
VAR=something
OTHER_VAR="I have whitespace"
```

- Precede with $ to view (dereference) a parameter:

```
$ echo $OTHER_VAR
I have whitespace
$ echo "My name is $USER"
My name is kschmidt
```
Common Parameters

- **PATH** – list of directories searched by shell for disk utilities
- **PS1** – primary prompt
- **USER** – user’s login name
- **HOME** – user’s home directory
- **PWD** – current working directory
Other Useful Shell Variables

- **SHELL** – The *login* shell
- **$$** – The PID of the current shell
- **$$?** – The return value of the last command
- **TERM** – Terminal type (what the shell thinks the terminal interface is)
- **HOSTNAME** – Machine’s hostname (see `uname`)
- **EDITOR** – Some programs (mutt, sudoedit, git, etc.) might look here, when opening a text file
- **SHELLOPTS** – Status of various Bash options (see the `set builtin`)
Command Substitution

- Replaces the command with the output of the command
  $$\text{name} \ \text{cmd} \ \text{name}$$

  ```
  $ echo Today is $(date '+%d %B')
  04 October
  ```

- Command can also be enclosed in back-tics, ‘
  ‘cmd‘

  ```
  $ echo Today is ‘date ’+%d %B’‘
  04 October
  ```

- This is Bourne syntax
- Tougher on the eyeballs
Use the backslash to inhibit the special meaning (behavior) of the metacharacter that follows.

```bash
$ echo $USER
kschmidt
$ echo \$USER
$USER
```

So, now \ is a metacharacter. Escape it to get just the character:

```bash
$ echo a\b
a\b
```
The backslash, when followed immediately by a newline, effectively removes the newline from the stream

```
$ echo On the bloody morning after\
> One tin soldier rides away
On the bloody morning after One tin soldier rides away
```

Use quotes, if you want the newline in the output:

```
$ echo "On the bloody morning after"
> One tin soldier rides away"
On the bloody morning after
One tin soldier rides away
```
Weak Quoting

Double quotes inhibit all but \‘ $ !

\begin{verbatim}
$ echo "$USER is $USER"
kschmidt is kschmidt
$ echo "\$USER is $USER"
$USER is kschmidt
$ echo "I said, "Well, we shan’t"
I said, "Well, we shan’t"
$ echo "It is now $(date ’+%H:%M’)"
It is now 19:27
\end{verbatim}

\footnote{If history expansion is enabled}
Strong Quoting

Single quotes preserve the literal value of all enclosed characters

- May not contain a single quote (can’t be escaped)

```
$ echo 'I said, "Wait!"
I said, "Wait!"
$ echo 'My name is $USER'
My name is $USER
```
String Concatenation

- Strings are concatenated simply by juxtaposition
- You needn’t restrict yourself to one set of quotes
  - Use the convenient quotes for a part of the string, other quotes for another bit

```bash
$ echo "$USER is "$USER"
$USER is kschmidt
$ echo 'He said it’\’s fine, "$USER"
He said it’s fine, kschmidt
```
Redirecting I/O

The shell can read stdin from sources other than your keyboard

- Input can be redirected from a file
- Input can even be taken from the output of another process, though a pipe

Similarly, stdout (and stderr) can go places other than your screen

- Redirected to a file
- Piped to another process to read as input
Redirecting `stdout`

- `stdout` is file descriptor 1
- Use `>` after a command (and its arguments) to redirect the output to a file:
  
  ```
  $ ls > list.out
  ```

- If `list.out` previously existed it will be truncated (gone)
- Use `>>` to append the output to the file.

  ```
  $ ls >> list.out
  ```
Using `echo` to Write Files

- `echo` (builtin, and disk utility) writes a line to `stdout`
  - `-n` suppresses the newline
  - `-e` permits expansion of escape chars (\t, \n, etc.)
- The `printf` utility is handy for formatting output

```
$ idx=127
$ echo "First line" > "$logfile"
$ echo "Another line" >> "$logfile"
$ printf '%-15s formatted line %5x\n' "$USER" $idx >> "$logfile"
$ cat "$logfile"
First line
Another line
kschmidt formated line 7f
```
Create Files with `cat`

- `cat`, in the absence of command-line args, reads from stdin, writes to stdout
- We can use this to write to a file
  - Use `^D` (Ctrl-D) to end input

```
$ cat > "$ofile"
This is line one
Another line

Okay, that’s enough
^D
```

- Handy way to concatenate files:

```
$ cat part1 > result
$ cat part2 >> result
```
Redirecting **stderr**

**stderr** is file descriptor 2, so:

```
$ gcc buggy.c 2> error.log
$ grep '[Vv]era' *.html > results 2> error.log
```

To send both to the same place:

```
$ find . -name 'core*' > core.list 2>&1
```

- Note, the order matters
- Bash has syntactic sugar for this move:

```
$ find . -name 'core*' &> core.list
```
Redirecting `stdin`

- `<` redirects `stdin` from a file
  - File descriptor 0

```
$ sort < nums
$ mail -s"Meaningful subject" $id < msg
```

- You can do both

```
$ sort < nums > sortednums 2> sort.errors
$ tr 'a-z' 'n-za-m' < code.rot13 > decoded
```
Here Documents/Strings

- Here documents are helpful in scripts
- Input is redirected using « [–] WORD
  - WORD signals end of input
- We’ll examine these further in a subsequent lecture

```
$ cat << EOS
Dear $NAME:
I am writing this slowly, since I know you can’t read fast.

It was so windy here Tuesday the chicken laid the same egg
$EGG_CNT times.
EOS
```

- Here strings are convenient on the command line:

```
$ bc -l <<< "$x + s($d)"
```
Unnamed Pipes

- A redirector links a process to a file
- A *pipe* links a process to a *process*
- It's a stream of data

```
prog1    Pipe    prog2
  STDOUT  |      |  STDIN
```

- Data written to `stdout` by `prog1` is read on `stdin` by `prog2`
- *Much* faster than writing, then reading, intermediate files
Asking for a Pipe

- Separate 2 commands with |
- The shell does all the work

```
$ du -s * | sort -n
$ du -s * | sort -n > sorted.lst
```

- Processes can be strung together with pipes:

```
$ du -s * | sort -nr | head -n10 > 10_biggest_files
```
The Unix Philosophy

The use of pipes and other features to combine “small, sharp tools” to accomplish larger tasks
– Ken Thompson (father of Unix)

“...at its heart is the idea that the power of a system comes more from the relationship among programs than from the programs themselves.”
– Brian Kernighan & Rob Pike

“This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.”
– Doug McIlroy (inventor of the Unix pipe)
- **history** shows an enumerated list

- `! n` executes the $n^{th}$ command
  - `!!` executes the last command
  - `!−n` works backwards, from last command
  - `!str` – last command that starts w/`str`
  - `~src~targ~` substitute `targ` for `src` in last command
Searching the History

- You might find `history | grep str` to be useful
- You might also read up on the `fc` builtin
  - I have, in my `.bashrc`:
    ```bash
    alias r='fc -s'
    ```
  - so I can use
    ```bash
    r str
    ```
  - to execute the last command that started with str
- I often use the Readline search
  - In Vim mode
    ```bash
    ESC / regex
    ```
  - In Emacs mode
    ```bash
    C-r regex
    ```
Processes, Jobs
Process Control

- Processes are run in a subshell
- Subshells inherit exported environment
- Each process has an ID (PID) and a parent (PPID)
- Use the `ps` utility to look at processes:

```sh
$ ps
  PID  TTY     TIME CMD
 350 pts/4   00:00:00 bash
22251 pts/4  00:00:00 vim
22300 pts/4  00:00:00 ps
```
Use the `-f` option for a long (full) listing

$ ps -f

```
UID   PID  PPID  C   STIME TTY   TIME CMD
kschmidt  350  349  0  10:06 pts/4 00:00:00 -bash
kschmidt 22251 350  0  17:32 pts/4 00:00:00 vim myHomework
kschmidt 22437 350  0  17:36 pts/4 00:00:00 ps -f
```

Use the `-e` option to see all of the processes (not just yours)

---

Caution: The entire command line, command plus args, are visible here. Be careful what you type in the shell.
Killing a process

- The `kill` (built-in and utility) sends a signal to a process
  - By default, sends the `SIGTERM` (signal 15)
  - Send `SIGKILL` (9), won’t be ignored, but, no cleanup
- To kill a process using its PID:

  ```
  $ kill 29940
  $ kill -n9 29940 # if it ignored your previous request
  ```

- See also `pgrep` and `pkill`
The shell allows you to manage jobs

- Place a job in the background
- Move a job to the foreground
- Suspend a job
- Kill a job
- Use jobs to view current jobs (in a given shell)

```
$ jobs
[6]- Running soffice CS265/Lectures/Unix/intro.ppt & (wd: ~)
[7]+ Stopped vi hello.tex
```
Job Control

- When a process is running, the shell is blocked
- So, we run, e.g., GUI programs in the background
- Processes that might take a while we can place in the background
- Place a & after a command to run it in the background:

```
$ firefox &
$ evince unix.pdf &
$ find ~/ -type f -mtime -1 > find.out # this might run a while
$ # Save output to a file
```
Suspending and Resuming a Process

- Use `^Z` to suspend the process in the foreground
- Use `fg` to bring the most recent process back to the foreground
  - Working in an editor, I’ll save, `^Z` out, compile, then `fg` back to my editing session
- Or, type `%n`, where `n` is the index, from the `job` listing
- Use `bg` to put the most recently suspended process into the background

```
$ evince unix.pdf  # Whoops! Forgot to put in background
^Z  # Suspend evince
$ bg    # Set it running in the background
```
Killing a job

- You can kill a job much as you might a process
- `SIGTERM` is often like closing the window, or choosing "Quit"
  - `SIGKILL`, on the other hand, can’t actually be trapped
  - The plug will be pulled on the process, no chance to clean up
  - Not really nice
- Specify a job using %:

```bash
$ kill %4  # Give it a chance to exit itself
$ kill -n9 %4  # Just pull the plug
```
Command Execution
Commands in a sequence can be joined several ways:

- Sequenced
- Grouped
- Subshell Group
- Conditional
Command Sequences

- Commands to be executed serially
- No direct relationship between them
- Commands can be separated by a newline or ;
  - Note, \n is a metacharacter

```bash
cmd1 ; cmd2 ; cmd3
```
Grouped Commands

{ cmd1 ; cmd2 ; cmd2 ; }

- Sequences can be grouped using {}
- List must be terminated by a ;
- Runs in the context of the current shell
- Useful for redirecting I/O
- Return value is the status of the last command executed

```bash
$ echo a ; echo b ; echo c > out
a
b
c
$ cat out
$ { echo a ; echo b ; echo c ; } > out
$ cat out
a
b
c
```
Grouped Commands for Subshell

( cmd1 ; cmd2 ; cmd2 )

- Sequences grouped with ()
- Also handy for redirecting I/O
- Runs in a subshell
  - Can be run in the background
  - No changes persist
- Return value is the status of the last command executed

```
$ y=30 ; ( y=20 ; echo $y ) ; echo $y
20
30
```
Conditional Execution

Operators && and ||

- Conditional execution depends on the return value of the command on the left
- This value available to the caller (parent shell)
  - Look at special variable $$?$$ for return status of last command
- Value on \([0, 255]\)^1
  - Zero (0) signals success; is true
  - Now you know why we told you to put `return( 0 );` at the end of your programs
  - We enumerate errors (failure), starting at 1
- `&&` and `||` have the same precedence, associate left-to-right

---

^1 Though I’ve used a shell that used only 7 bits
Conditional Execution Operators

```
cmd1 && cmd2
  - cmd1 is executed first
  - If cmd1 succeeds, cmd2 is executed

cmd1 || cmd2
  - cmd1 is executed first
  - Only if cmd1 fails is cmd2 executed
```

```
$ cp file1 file2 && echo "Copy succeeded"
Copy succeeded
$ cp no_such_file file2 2> /dev/null || echo "Copy failed"
Copy failed
```
Subshells
Subshells

- A *child process* launched by a shell
- A separate instance of the command processor (the shell)
- Inherits *copies* of a subset of the parent’s environment
  - Only those items marked for export:
    ```bash
    export myVar
    ```
  - Once marked for export, remains so marked
Subshells – Who Gets ’Em?

- Shell builtins are handled by that shell (no child)
- Functions execute in the current context
- Scripts run in a subshell
- Child shell forked off for an external process (disk program)
  - Child `execs` process
- Statements inside parentheses run in the same subshell

---

1Maybe. Bash seems to emulate a subshell these days. PID, e.g., is the same
Subshells – Environment Variables

- Exported variables (functions, aliases) are copied into the subshell
- Changes made to the environment in a subshell do not persist upon exit
- Variables (functions, etc.) created in the subshell will be gone at exit

```
$ myA=chocolate
$( myA=strawberry ; echo $myA ; lowerB=Jupiter )
strawberry
$ echo $myA
chocolate
$ echo "\$lowerB is: $lowerB"
$lowerB is:
```