Variables and Processes

- Positional and Special Parameters
- Quoting and the Evaluation of Variables
- Removing a Variable's Value
- Variable Attributes
- Processes
  - Process Structure
  - Process Identification
  - Executing a Command

Positional and Special Parameters

- Positional and special parameters are *variables* set by Unix that change each time you enter a command
- We have already encountered the special parameter **?**
- It contains the status code returned by the *most recent* command

```
$ ls bar.txt
bar.txt

$ echo $?
0

$ ls xxx
ls: cannot access xxx: No
such file or directory

$ echo $?
2
```

Positional and Special Parameters

- **Positional parameters** are used by shell scripts to get *arguments* from the command line
- Each word (or "token") on the command line is assigned to a positional parameter
- The first word is the pathname of the script and is assigned to positional parameter `0`
- Each succeeding word on the command line is assigned to *the next integer*…

Positional and Special Parameters

- *Positional numbers*:

```
$ cat print_positionals.sh
#!/bin/bash
#
# Prints the value of the
# first four positional
# arguments

echo

echo 0: $0
echo 1: $1
...
```

```
...
echo 2: $2
echo 3: $3

$ ./print_positionals.sh foo
bar bletch
0: ./print_positionals.sh
1: foo
2: bar
3: bletch
```

Positional and Special Parameters

- Positional parameters are the usual way input is given to a script
- If you need the value of argument number *10 or higher* you have to enclose the number in *curly braces*:

```
echo ${10} ${11} ${12} # print the value of
                       # arguments 10, 11 & 12
                       # within a script
```

Positional and Special Parameters

- Another special parameter is **#**
- **#** contains the number of arguments passed *to* a program *from* the command line
- Notice that **#** counts the **arguments** to the script – not the name of the script itself

```
$ cat print_arg_numbers.sh
#!/bin/bash
#
# Prints the number of arguments
# sent to this script

echo
echo This script received $#
arguments

$ ./print_arg_numbers.sh foo bar
     bletch
This script received 3 arguments
```

Quoting and the Evaluation of Variables

- Whenever the value of a variable contains **spaces** or **tabs**, you must *quote* the string or *escape* the whitespace character
- There are three ways to do this:
  - **Single quotes:** ' '
  - **Double quotes:** " "
  - **Backslash:** \

Quoting and the Evaluation of Variables

- Single quotes are the most restrictive
- Everything surrounded by single quotes appears in the variable *exactly* as you typed it
- This means that special meaning of characters like **$** before a variable name are ignored

```
$ team='Red Sox'

echo $team
Red Sox

$ cheer='Go $team'

$ echo $cheer
Go $team
```

Quoting and the Evaluation of Variables

- Double quotes also preserve spaces and tabs in the strings they contain
- But, you can use a **$** in front of a variable name to get *the value of a variable* inside double quotes:

```
$ cheer="Go $team"

$ echo $cheer
Go Red Sox
```

Quoting and the Evaluation of Variables

- Quotes affect *everything* they enclose
- The backslash **** only effects the character *immediately* following it

```
$ foo=bar

$ echo $foo
bar

$ foo3=\$foo

$ echo $foo3
$foo
```

Removing a Variable's Value

- There are *two* ways of removing the value of a variable
- You can use the ***unset*** command
- Notice that the variable name was **not** preceded by a **$**
- That's because we are dealing with *the variable itself*, not its value.

```
$ echo $foo
FOO

$ unset foo

$ echo $foo

$
```

Removing a Variable's Value

- The other way of removing a variable's value is to set the value of the variable to *the empty string*:

```
$ echo $foo
FOO

$ foo=

$ echo $foo

$
```

Variable Attributes

- Variables can have *attributes* such as being ***read-only*** or ***global***
- We have already seen one way to set the attribute of a variable
- If you precede the name of a variable with **`export`** , it makes the variable global

```
$ export foo=FOO
```

Variable Attributes

- You can make a variable read only by using the ***readonly*** command:

```
$ echo $foo
FOO

$ readonly foo

$ foo=bar
-bash: foo: readonly variable
```

Variable Attributes

- You must set the value of a variable *before* you make it read-only. Once you make a variable read-only, it *cannot* be changed
- There are many more variable attributes
- They can be set using one of two commands:
  - ***declare***
  - ***typeset***

Variable Attributes

- ***declare*** and ***typeset*** have different names but do the *same thing* and have the same *options*…

Option	Meaning
`a`	Declares a variable to be an array
`f`	Declares a variable to be a function name
`i`	Declares a variable to be an integer
`r`	Makes a variable read only
`x`	Makes a variable global

Variable Attributes

- Let's look at some examples...

```
$ foo=bar

$ echo $foo
bar

$ foo=bletch
...
```

```
...
$ echo $foo
bletch

$ declare -r foo

$ foo=bling
-bash: foo: readonly
variable
```

Processes

- A **process** is a running program
- Every process has *resources* that it needs to do its job
- Unix is a *multitasking* operating system, so many processes can run at the same time
- The shell runs in a process like any other command
- Every time you run a program (except a built-in), a new process is created

Processes

- Running a built-in command ***does not*** create a process because…
  - The built-in is part of the shell
  - The shell already has a process
- When a shell script is run your current shell creates a sub-shell to run the script
- This sub-shell runs in a ***new*** process

Process Structure

- There is a structure to the creation of processes
- They are created in a *hierarchical* fashion
- When the machine is started there is only one process. This process is called **`init`**
- **`init`** then creates all the other processes needed to run the machine
  - These new processes are *child* processes of **`init`**
  - These child processes can create *other* processes

Process Structure

- This is what happens when the shell creates a sub-shell
  - The process that creates the new process is called the *parent* process and the processes it creates are called its *child* processes
  - A new process is created by calling an *operating system routine*
- When a process calls an operating system routine it is said to make a ***system call***

Process Structure

- System calls are used by programs to have the operating system perform some action that only the operating system can do, like create a file
- When Unix is booted, a single process called `init` is started:
  - `init` is a spontaneous process
  - It does not need a parent process to create it
  - `init` has **PID** (Process ID) of 1

Process Structure

- `init` is the *ancestor* of every other processes that ever runs on the machine
- Just as the filesystem has a single directory **/** at the top of the filesystem hierarchy, so the `init` process is at the top of the process hierarchy
- When a Unix system is run in multiuser mode, `init` runs ***getty*** or ***mingetty***
- These programs allow users to login and display the prompts, which ask for a *user name and password*

Process Structure

- When the user's responses control is handed over to the ***`login`*** utility, ***`login`*** checks the password against the user ID
- If the password is *correct*, the ***`login`*** process becomes the user's shell process

Process Identification

- Each process has a unique ***Process ID*** (PID) number
- As long as the process runs, it has the same PID
- After a process *terminates*, its PID can be assigned to a new process
- ***ps -f*** displays a full listing of information about each process running for the user:

```
$ ps -f
UID          PID  PPID  C STIME TTY          TIME CMD
it244gh   26374 26373  0 13:41 pts/5    00:00:00 -bash
it244gh   27891 26374  0 13:57 pts/5    00:00:00 ps -f
```

Process Identification

- Column meanings:
  - **UID**: The user's Unix *username*
  - **PID**: The process ID of the *process*
  - **PPID**: The process ID of the *parent process* – the process that created this process
  - **CMD**: The *command* that is running in the process

Process Identification

- If I were to run ***sleep*** in the background, and then run ***ps -f***, I would see

```
$ sleep 10 & ps -f
[1] 27352
UID         PID  PPID  C STIME TTY          TIME CMD
ghoffmn   27292 27287  0 15:12 pts/1    00:00:00 -bash
ghoffmn   27352 27292  0 15:13 pts/1    00:00:00 sleep 10
ghoffmn   27353 27292  0 15:13 pts/1    00:00:00 ps -f
```

- Notice that the parent process of both ***sleep*** and ***ps -f*** is my login shell

Process Identification

- **pstree** will display a *tree* of all currently running processes
- If I run **pstree** on it244a , then I can see the process structure
- Where you see a number followed by a * , it means there are *multiple* versions of that software running *in different processes*

```
$ pstree
init─┬─acpid
     ├─atd
     ├─automount───10*[{automount}]
     ├─5*[bother.sh───sleep]
     ├─cron
     ├─dbus-daemon
     ├─dhclient
     ├─6*[getty]
     ├─master─┬─pickup
     │        └─qmgr
     ├─rpc.idmapd
     ├─rpc.statd
     ├─rpcbind
     ├─rsyslogd───3*[{rsyslogd}]
     ├─rwhod───rwhod
     ...
```

Process Identification

- You can see this more clearly if you run ***pstree*** with the **-p** option – which will show each process, along with its process ID:

```
$ pstree -p

init(1)─┬─acpid(1002)
        ├─atd(1157)
        ├─automount(1177)─┬─{automount}(1179)
        │                 ├─{automount}(1180)
        │                 ├─{automount}(1183)
        ...
```

```
$ pstree -p
init(1)─┬─acpid(1002)
        ├─atd(1157)
        ├─automount(1177)─┬─{automount}(1179)
        │                 ├─{automount}(1180)
        │                 ├─{automount}(1183)
        │                 ├─{automount}(1186)
        │                 ├─{automount}(1195)
        │                 ├─{automount}(1196)
        │                 ├─{automount}(1197)
        │                 ├─{automount}(1198)
        │                 ├─{automount}(1199)
        │                 └─{automount}(1200)
        ├─bother.sh(6166)───sleep(7363)
        ├─bother.sh(6170)───sleep(7362)
        ├─bother.sh(6173)───sleep(7378)
        ├─bother.sh(10606)───sleep(7364)
        ├─bother.sh(10607)───sleep(7365)
        ├─cron(1009)
        │
        ...
```

```
...
 ├─dbus-daemon(391)
 ├─dhclient(610)
 ├─getty(955)
 ├─getty(958)
 ├─getty(961)
 ├─getty(962)
 ├─getty(964)
 ├─getty(1267)
 ├─master(1118)─┬─pickup(6072)
 │              └─qmgr(1123)
 ├─rpc.idmapd(400)
 ├─rpc.statd(778)
 ├─rpcbind(662)
 ├─rsyslogd(387)─┬─{rsyslogd}(394)
 │               ├─{rsyslogd}(395)
 ...
```

```
...
 |                    └─{rsyslogd}(396)
 ├─rwhod(1144)──rwhod(1146)
 ├─sshd(798)──sshd(6107)──sshd(6196)──bash(6197)──pstree(7379)
 ├─systemd-logind(21591)
 ├─systemd-udevd(21445)
 ├─upstart-file-br(465)
 ├─upstart-socket-(906)
 ├─upstart-udev-br(21442)
 ├─vmtoolsd(1202)
 └─ypbind(859)─┬─{ypbind}(860)
               └─{ypbind}(864)
```

The part in **red** is my current logging shell, and its *child* process running ***pstreet***

- For some reason, the connecting lines in the output of this command do not appear properly when running an ***ssh*** client on Windows

Executing a Command

- When you run a command from within a shell, the shell creates a *child process* using a system call; then it *sleeps*, waiting for the child process to finish
  - While sleeping, the parent process is inactive
  - When the child process finishes, it notifies its parent process of its success or failure by returning an *exit status code* … and then it dies
  - When the parent process receives the exit status code, it wakes up and it runs again

Executing a Command

- When you run a command in the background, the shell creates a child process for the job but *does not go to sleep*
- When running a built-in, the shell *does not* create a process because the built-in runs in the same process as the shell
- By default, variables are ***local*** and are ***not*** passed to child processes
- But, global variables are ***inherited*** by all child processes