Introduction to the bash Shell

(Bourne-Again SHell)

Commands, programs, scripts, etc.

Command

A directive to the shell typed at the prompt. It could be a utility, a program, a built-in, or a shell script.

Program

A file containing a sequence of executable instructions. Note that it's not always a binary file but can be text (that is, a script).

Script

A file containing a sequence of text statements to be processed by an interpreter like **bash**, Perl, etc.

Every program or script has a **stdin**, **stdout**, and **stderr** by default, but they may not always be used.

Filter

A command that can take its input from **`stdin`** and send its output to **`stdout`**. It is often used to transform a stream of data through a series of pipes.

Scripts are often written as filters.

Utility

A program/script or set of programs/scripts that provides a service to a user.

Built-in

A command that is built into the shell. That is, it is not a program or script as defined above. It also do not require a new process, unlike those above.

History

A list of previous shell commands that can be recalled, edited if desired, and re-executed.

Token

The smallest unit of parsing; often a word delimited by white space (blanks or spaces, tabs and newlines) or other punctuation (quotes and other special characters).

`stdin`

The standard input file; the keyboard; the file at offset 0 in the file table.

`stdout`

The standard output file; the screen; offset 1 in the file table.

stderr

The standard error file; usually the screen; offset 2 in the file table.

Standard I/O (Numbered 0, 1, and 2, in order)

stdin, **stdout**, and **stderr**

Pipe

To make use a shell service (filters) that connects the **stdout** of one program to the **stdin** of the next; the "**|**" (pipe, or vertical bar) symbol.

Redirect

To use a shell service that replaces **stdin**, **stdout**, or **stderr** with a regular named file.

Process

- A process is what a script or program is called while it's being executed. Some processes (called daemons) never end, as they provide a service to many users, such as crontab services from **crond**.
- Other processes are run by you, the user, from commands you enter at the prompt. These usually run in the foreground, using the screen and keyboard for their standard I/O. You can run them in the background instead, if you wish.
- Each process has a PID (or pid, the process identifier), and a parent process with its own pid, known to the child as a ppid (parent pid). You can look at the running processes with the **ps** command or examine the family relationships with **pstree**.

Child process

- Every process is a child process, with the sole exception of process number 1 – the **`init`** process.
- A child process is forked or spawned from a parent by means of a system call to the kernel services.
- Forking produces an exact copy of the process, so it is then replaced by an exec system call.
- The forked copy also includes the environment variables and the file table of the parent.
- This becomes very useful when redirecting standard I/O, since a child can redirect its own I/O without affecting its parent.
- Each command you type is run as a child of your shell (however, see also the **`source`** command).

Parsing the command line

1. Substitute **`history`**
2. Tokenize command line (break it into "words" based on spaces and other delimiters)
3. Update **`history`**
4. Process quotes
5. Substitute **`aliases`** and defined functions
6. Set up redirection, pipes, and background processes
7. Substitute variables
8. Substitute commands
9. Substitute filenames (called "globbing")
10. Execute (run) the resulting program

History

- The command history is a list of all the previous commands you have executed in this session with this copy of the shell. It's usually set to some large number of entries, often 1000 or more.
- Use **`echo $HISTSIZE`** to see your maximum entries
- You can reach back into this history and pull out a command to edit (if you wish) and re-execute.

Some history examples

- To list the history:

  System prompt> history | less
- To repeat the last command entered:

  System prompt> !!
- To repeat the last ls command:

  System prompt> !ls
- To repeat the command from prompt number 3:

  System prompt> !3
- To scroll up and down the list:

  Use arrow keys
- To edit the command:

  Scroll to the command and edit in place

Redirection

- Three files are open and available immediately upon shell startup: **stdin**, **stdout**, **stderr**
- These can be overridden by various redirection operators
- Following is a list of most of these operators (there are a few others that we will not often use; see **man bash** for details)
- Multiple redirection operators are processed from left to right: **redir-1 redir-2** may not be the same as **redir-2 redir-1**
- If no number is present with **>** or **<**, **0** (**stdin**) is assumed for **<** and **1** (**stdout**) for **>**; to work with **2** (**stderr**) it must be specified, like **2>**

Operator	Behaviour
Individual streams	
< **filename**	Redirects **stdin** from **filename**
> **filename**	Redirects **stdout** to **filename**
>> **filename**	Appends **stdout** onto **filename**
2> **filename**	Redirects **stderr** to **filename**
2>> **filename**	Appends **stderr** onto **filename**
Combined streams	
&> **filename**	Redirects both **stdout** and **stderr** to **filename**
>& **filename**	Same as **&>**, but do not use
&>> **filename**	Appends both **stdout** and **stderr** onto **filename**
>>& **filename**	Not valid; produces an error

Operator	Behaviour
Merged streams	
2>&1	Redirects **stderr** to the same place as **stdout**, which must already be redirected
1>&2	Redirects **stdout** to the same place as **stderr**, which must already be redirected
Special stdin processing ("here" files), mainly for use within scripts	
<< string	Read **stdin** using **string** as the end-of-file indicator
<<- string	Same as **<<**, but remove leading **TAB** characters
<<< string	Read **string** into **stdin**

Command aliases

- To create an alias (no spaces after alias name)

  `alias ll="ls -l"`

- To list all aliases

  `alias  or  alias | less`

- To delete an alias

  `unalias ll`

- Command aliases are normally placed in your **`~/.bashrc`** file (first, make a back-up copy; then use **`vi`** to edit the file)

- If you need something more complex than a simple alias (they have no arguments or options), then write a bash function script.

Filename Globbing: Metacharacters

Metacharacter	Behaviour
`\`	Escape; use next char literally
`&`	Run process in the background
`;`	Separate multiple commands
`$xxx`	Substitute variable **xxx**
`?`	Match any single character
`*`	Match zero or more characters
`[abc]`	Match any one char from list
`[!abc]`	Match any one char **not** in list
`(cmd)`	Run command in a subshell
`{cmd}`	Run in the current shell

Simple Quoting

- No quoting:

```
System Prompt$ echo $SHELL
/bin/bash
```

- Double quote: "

```
System Prompt$ echo "$SHELL"
/bin/bash
```

- Single quote: '

```
System Prompt$ echo '$SHELL'
$SHELL
```

Observations:

Double quotes allow variable substitution;

Single quotes do not allow for substitution.

Escape and Quoting

- Escape alone:

```
Prompt$ echo \$SHELL
$SHELL
```

- Escape inside double quotes:

```
Prompt$ echo "\$SHELL"
$SHELL
```

- Escape inside single quotes:

```
Prompt$ echo '\$SHELL'
\$SHELL
```

Observations:

Escape leaves the next char unchanged;
Double quotes obey escape (process it);
Single quotes don't process it (ignore it)

Filespecs and Quoting

```
System Prompt$ ls
a  b  c
System Prompt$ echo *
a b c
System Prompt$ echo "*"
*
System Prompt$ echo '*'
*
System Prompt$ echo \*
*
```

Observation:

Everything prevents file globs

Backquotes and Quoting

```
System Prompt$ echo $(ls)   # alternate
a b c
System Prompt$ echo `ls`    #   forms
a b c
System Prompt$ echo "`ls`"
a
b
c
System Prompt$ echo '`ls`'
`ls`
```

Observations:

Single quotes prevent command processing

Summary so far

Double quotes allow variable substitution

"$SHELL" becomes **/bin/bash**

Single quotes do not allow for substitution

'$SHELL'** becomes **$SHELL

Escape leaves the next char unchanged

\$SHELL becomes **$SHELL**

Double quotes obey escape (process it);

"\$SHELL" becomes **$SHELL**

Single quotes don't process it (ignore it)

'\$SHELL' becomes **\$SHELL**

Everything prevents file globs

"*" '*' * each become *

Single quotes prevent command processing

'`ls`' becomes **`ls`**

Escaping quotes

```
System Prompt$ echo ab"cd
> "
abcd
System Prompt$ echo ab\"cd
ab"cd
System Prompt$ echo 'ab\"cd'
ab\"cd
System Prompt$ echo "ab"cd"
> "
abcd
```

More quote escapes

```
System Prompt$ echo "ab\"cd"
ab"cd
System Prompt$ echo don't
> '
dont
System Prompt$ echo don\'t
don't
System Prompt$ echo "don't"
don't
System Prompt$ echo 'don't'
> '
dont
```

Observations

Unbalanced quotes are not allowed

- **`don't`** causes an error

Unescaped quotes are removed

- **`"hello"`** becomes **`hello`**

Quoting protects quotes, as does `\` escaping

- **`"don't"`** and **`don\'t`** are the same, and OK

Single quotes are more restrictive than double

```
System Prompt$ echo '$USER' "$USER"
$USER allisor
```

A Short Script

To further examine command-line substitutions, it's of considerable value to have a short script to do the heavy lifting.

What do we need it to do?

It's got to look at each argument on the command line from the first one to the last. Luckily, we can use the special variable **$#** to find the last argument.

The list starts at argument 0, which is the command itself, and each argument can be referenced in the form **$0**, **$1**, **$2**, and so on.

We'll want to count each argument, both to display each number and so we'll know when to stop. Stop what? Ah, we're going to do the same thing over and over, so that means we will need a loop.

Describe the Problem

```
GET the count of the arguments from $#
SET a counter variable to 0
SHOW how many arguments there are
LOOP for each argument (until our count is $#)
    SHOW the argument number
    SHOW the value of the argument
    COUNT how many arguments we have done
END LOOP
SHOW that we're all done
```

That's a sample of PDL, the Problem Description Language that we'll use this semester (and all later semesters) for scripting. Don't worry too much about it just yet. Next slide is the already-written script.

```
System Prompt$ cat tt
#! /bin/bash
# create count with starting value zero
declare count=0
# show how many command arguments there are
echo Number of arguments: $#
# while still command-line arguments, do this
while (( $count <= $# ))
do
   # show the arg number and its |value|
   eval "echo arg $count '|'\${$count}'|'"
   # count each argument processed
   let "count += 1"
# end loop
done
echo all arguments evaluated
System Prompt$
```

What's that `eval` do?

Here's what is in **`man bash`** for **`eval`** (it's a built-in command):

`eval [arg ...]`

> **`The args are read and concatenated together into a single command. This command is then read and executed by the shell, and its exit status is returned as the value of eval. If there are no args, or only null arguments, eval returns 0.`**

Briefly, then, it constructs a command and then executes it.

If we start with this:

```
eval "echo arg $count '|'\${$count}'|'"
```

and if **$count** has the value, say, 2, we next get:

```
eval "echo arg 2 '|'${2}'|'"
```

which **eval** executes as:

```
echo arg 2 '|'${2}'|'
```

giving us (for **./tt aa bb**):

```
arg 2 |bb|
```

as our output line on **stdout**.

So, eval substitutes what it can (**$count** becomes **2**, **\$** becomes **$**) and executes the resulting string (**echo arg 2 '|'${2}'|'**) as a single command.

Some Sample Runs

```
System Prompt$ ./tt
Number of arguments: 0
arg 0 |./tt|
all arguments evaluated

System Prompt$ ./tt a
Number of arguments: 1
arg 0 |./tt|
arg 1 |a|
all arguments evaluated

System Prompt$ ./tt a b
Number of arguments: 2
arg 0 |./tt|
arg 1 |a|
arg 2 |b|
all arguments evaluated
```

```
System Prompt$ ./tt ab cd
Number of arguments: 2
arg 0 |./tt|
arg 1 |ab|
arg 2 |cd|
all arguments evaluated
```

```
System Prompt$ ./tt ab\ cd
Number of arguments: 1
arg 0 |./tt|
arg 1 |ab cd|
all arguments evaluated
```

To run this yourself, copy and paste the source into a **vi** text file (**vi tt**; **i** for insert mode; select Paste from the Edit menu; **ESC**). Save it (**:wq**) and set it to be executable (**chmod +x tt**).

Filespec Args with Variable

```
Prompt$ export s="*"
Prompt$ echo $s
a b c tt
Prompt$ echo '$s'
$s
Prompt$ echo "$s"
*
Prompt$ ./tt s
Number of args: 1
arg 0 |./tt|
arg 1 |s|
all args evaluated
```

```
Prompt$ ./tt $s
Number of args: 4
arg 0 |./tt|
arg 1 |a|
arg 2 |b|
arg 3 |c|
arg 4 |tt|
all args evaluated
```

Other Commands

- You should also examine all the other commands suggested in the lab document "60 Commands" (on the course web site), filters like **`grep`**, **`cut`** and **`tr`**, for example.
- You will need to know the basic behavior of many of these commands for labs and assignments, but many of the administration commands we will address together.
- Feel free to refer to your textbooks and the man pages for more insight. (Translation: that actually means for you to look up what we've discussed, read it over, and do some lab practice with all the commands and services. Or else!)