Basic UNIX

Processes and Shells

Shell

ls
pico
httpd

Kernel

CPU
Disk
NIC
Processes and Shells

Processes

Processes are tasks run by you or the OS.

Processes can be:

- shells
- commands
- programs
- daemons
- scripts
Basic UNIX

Processes and Shells

Shells

Processes operate in the context of a *shell*.

The shell is a command interpreter which:

- Interprets built-in characters, variables and commands
- Passes the results on to the kernel

The *kernel* is the lowest level of software running. It controls access to all hardware in the computer.
Basic UNIX

Processes and Shells

Shells

Types of shells:

- `/bin/sh` – Bourne shell
- `/bin/csh` – C shell
- `/bin/tcsh` – Enhanced C shell
- `/bin/bash` – Bourne “again” shell
- `/bin/zsh` – Z shell
- `/bin/ksh` – Korn shell
Shell Scripts

Shell scripts are files which contain commands to be interpreted and executed by a shell.

A shell is its own programming environment. Shells contain:

- Variables
- Loops
- Conditional statements
- Input and Output
- Built-in commands
- Ability to write functions
Shell Scripts

Specifying the shell to be used:

On the first line of the file:

- Implicitly
  - blank line – Bourne shell
  - # in column 1 – C shell

- Explicitly
  - #!/bin/sh – Bourne shell
  - #!/bin/csh – C shell
Basic UNIX

Processes and Shells

Exercise

Which shell are you using?

# echo $SHELL

(Don't worry about what these mean, we'll come back to them later)
Basic UNIX

An Interlude

How to be “Cool”

All UNIX people pronounce EVERYTHING. If you don't you aren't cool.

Examples:

• ! - bang
• # - pound
• awk – awk as in “awkward”
• grep – grrrrrrrr ep
• chmod – chaaaa mod
• chown – chaa own
• www – wu wu wu
Using `echo` is very useful for debugging scripts. The `echo` command prints the value of an expression (to the screen by default)

```
Hello World!
```

The `-n` option suppresses newlines
Basic UNIX
Processes and Shells

Exercise

Run the following script:

```
#!/bin/sh
i=1
while [ $i < 12 ]
do
  echo -n "."
  sleep 1
  i=`expr $i + 1`
done
```
echo

Echoing in a script prints each shell script line to the screen before it is executed. Use the command

```
set -x
```

to turn this on
Basic UNIX

Processes and Shells

Exercise

Run this script:

# ./echotoggle.sh

#!/bin/sh
set -x
echo
echo Here is a listing of the files
echo
ls -l
Shell Variables

Variables hold strings that can be used later

Two Types of Variables:

- Local (local scope)
- Environment (global scope)
Basic UNIX

Processes and Shells

Shell Variable Assignment

- Local Variables

  \[ variable = \langle value \rangle \]

- Environment

  \[ variable = \langle value \rangle \]
  \[ export \ variable \]
Basic UNIX

Processes and Shells

Accessing Variables

All variables are *dereferenced* by placing a $ in front of the variable name

```
<lister> echo $PATH
```
Basic UNIX
Processes and Shells

Exercise

Run this script:

# ./variables1.sh

#!/bin/sh
GREETING=Hello
export GREETING
there=there
friends='Kevin Lisa Joanne'
echo $GREETING $there $friends
Blanks and other *white space* are ignored by the shell. If you want them included, you must use quotes.

Two types of quotes:

- `'`
- `"`

Each has a different behaviour when using variables.
When a shell interprets each line, it performs variable substitution before executing commands.

If a variable is within double quotes, “ “, it will be substituted.

If a variable is within single quotes, it will not be substituted. It will take on its literal value.
Basic UNIX

Processes and Shells

Exercise

Run this script:

```bash
#!/bin/sh
GREETING=Hello
there=there
friend1=Kevin
friend2=Lisa
friend3=Joanne
friends="$friend1 $friend2 $friend3"
echo $GREETING $there $friends
echo $GREETING "$friends"
```
Basic UNIX

Processes and Shells

Listing Defined Variables For Your Current Shell

To list all variables, use the `set` command with no argument

For environment variables, use `export` with no variables

Exercise

Get listings of the current shell variables

```bash
# set
# export
```
Basic UNIX

Processes and Shells

Some Common Shell Variables

- **PATH** – directory paths to search for commands
- **HOSTNAME** – the name of the computer
- **USER** – the user id of the user running this shell
- **SHELL** – the shell currently being used
- **TERM** – the type of terminal being used
- **PS1** – the prompt to print when then shell is ready for another command
Deassigning Variables

For all variables, use the `unset` command

```bash
unset variable
```
Basic UNIX
Processes and Shells

Command Line Arguments

Powerful feature – passing values to your shell script.

- $1..$9 – first nine arguments
- $0 – name of the file/command
- $* - everything on the command line
- $# returns the number of arguments on the command line
Exercise

Run the following script:

```
#!/bin/sh
echo $#
echo $0
echo $1
```

# ./clargs.sh Hello World
# ./clargs.sh Hello
Basic UNIX

Processes and Shells

The $$ Variable

The $? variable returns the exit value of the most recently called command.

This is useful to detect successful completion of a program before continuing to a program which relies on the output of that command.

0 – usually a sign of success

non-zero – error of some sort
Basic UNIX

Processes and Shells

Special Characters

Filename Wildcards (Globbing)

Wildcard characters allow you to *match* multiple file names

Two wildcard characters:

? - matches a single character

* - matches one or more characters

Historical note: The jargon usage derives from *glob*, the name of a subprogram that expanded wildcards in archaic pre-Bourne versions of the Unix shell.
Basic UNIX

Processes and Shells

Special Characters

Filename Wildcards (Globbing)

Example:

Four files named biffo, boffo, baffa and baffo

b?f?fo matches biffo, boffo and baffo but not baffa

*ff* matches all four
Basic UNIX

Processes and Shells

Special Characters

The \ and # Characters

\ performs two roles:

- It “escapes” characters from substitution

- It signals the continuation of a shell script line to the next line

# before any characters imply that all following characters on the line make up a comment
Basic UNIX

Processes and Shells

I/O Streams and Redirection

Very powerful feature of the shell. Not found in other operating systems.

Think of input and output as streams of data.

Three “standard” streams for a program:

• **Stdin** – input stream
• **Stdout** – output stream
• **Stderr** – stream for error output (on a terminal – same as stdout)
I/O Streams and Redirection

You control the course of the data streams:

- `< file` – direct stdin from file
- `> file` – direct stdout to file
- `>> file` – append stdout to file
- `Command1 | command2` – connects stdout of command1 to stdin of command2 via a pipe
Basic UNIX
Processes and Shells

I/O Streams
File Descriptors

Open files are associated with numeric *file descriptors*

- **0** - stdin
- **1** – stdout
- **2** - stderr

You can direct output to multiple file descriptors simultaneously. The most common is

`1>&2 file` – direct stdout AND stderr to *file*
Basic UNIX

Processes and Shells

Exercise

Run the following script:

```
#!/bin/sh

### Your script here
```
Basic UNIX

Processes and Shells

Command Substitution

Any command contained within a pair of backticks ```` is executed immediately. The output of the command replaces everything in the backticks.

This can be used to assign the output of a command to an array to be used later

```
#!/bin/sh
files=`ls`
echo $files
```
Basic UNIX
Processes and Shells

Exercise

Run the following script:

```
#!/bin/sh
files=`ls`
echo $files
```
Expressions

Expressions are used in *statements* to control the flow of the shell.

Expressions are made up of constants, variables and operators.

Expressions always evaluate to strings. Numeric calculations can be performed but are translated back to strings.

Commands can be executed and variable substitutions can take place before an expression is evaluated.
Expressions

Most common expressions take on the form:

\[ \textit{token operator token} \]

where \textit{token} is usually a variable or a constant.

Types of operators:

• Numeric

• Logical
Numeric Expressions

Numeric expressions are evaluated using the `expr` command:

Numeric operators include `+,-,*,/,%` amongst others

Example:

```bash
#!/bin/sh
i=1
echo $i
i=`expr $i + 2`
echo $i
$i=`expr $i \* 3`
echo $i
```
Basic UNIX

Processes and Shells

Exercise

Run this script:

#!/bin/sh
i=1
echo $i
i=`expr $i + 2`
echo $i
$i=`expr $i \* 3`
echo $i
Logical Expressions

Logical expressions are almost always used with conditional statements.

Logical operators include

- |, &
- =, !=
- <=, >=, <, >
Basic UNIX

Processes and Shells

Logical Operators

- | - Boolean OR
- & - Boolean AND
- = - equivalent
- != - not equivalent
- <=, >=, <, > - lexical or numeric comparisons

Examples:

- expr $i <= 10
- expr “$1” = “dostats”
Logical Expressions

Logical expressions are very often built using the `test` command.

Test allows the shell to test for various conditions

- `test -d file` – true if `file` is a directory
- `test -e file` – true if file exists
- `test $foo -eq $bar` – true if the numbers `$foo` and `$bar` are equal
In fact, `test` is so useful, a shortcut has been created for it

```
[ -e file ]
```

is the same as

```
test -e file
```
Logical expressions can be used with four control statements to direct the flow of execution:

- if..then..elif..then..else..fi
- while..do..done
- for..in..do..done
- case..in..;;..esac
Basic UNIX

Processes and Shells

Control Statements

if statement

if logical expression
then
  .
  .
  .
elif logical expression
then
  .
  .
else
  .
  .
fi
Basic UNIX
Processes and Shells

Control Statements
statement order

Note that the pieces of the if statement needed to be on separate lines. They are each treated as commands.

You can place them on the same line by separating them with a semicolon. This holds for all flow control

```bash
if logical expression ; then
.
elif logical expression ; then
.i
```
Basic UNIX

Processes and Shells

Exercise

Run the following script:

```bash
#!/bin/sh
/bin/sh
st=0
if [ $st -eq 0 ];
then
    echo "Success!"
elif [ $st -eq 1 ];
then
    echo "I'm a failure!"
fi
```

Enter a CTRL-C and then CTRL-D

Then run it again with just CTRL-D
Basic UNIX
Processes and Shells

Control Statements

**case statement**

```plaintext
case string in
    pattern1)
    .
    .
    ;;
    pattern2)
    .
    .
    ;;
    *)
    .
    .
    ;;
esac
```
Basic UNIX

Processes and Shells

Exercise

Run this script:

```
#!/bin/sh

case $1 in
  '-d')
    echo debugging
    set debug
    ;;
  '-c')
    echo compiling
    set compile
    ;;
  *)
    file=$2
    echo $2
esac
```
Basic UNIX

Processes and Shells

Control Statements

*foreach* statement

```
for variable in wordlist
do
.
.
done
```

This statement *loops* over all of the values in *wordlist* and assigns them to *variable* one at a time until all values have been exhausted.
Basic UNIX

Processes and Shells

Exercise

Run this script:

# ./for.sh

#!/bin/sh
files=`ls -a`
for file in $files
do
    echo $file
done
Control Statements

while statement

while logical expression
  do
  .
  .
  end

This statement *loops until* the logical expression is false, that is, it continues to loop *while* the logical expression is true.

Make sure that logical expression can evaluate to false at some point or you will have an *infinite loop*.
Basic UNIX
Processes and Shells

Exercise

Run this script:

# ./while.sh

#!/bin/sh
n=1
echo 'Look. I can count!'
while [ $n <= 10 ]
do
echo -n " $n"
sleep 1
    n=`expr $n + 1`
done
echo " "
exits
Basic UNIX

Processes and Shells

Executing Shell Scripts

There are two ways to execute a shell script:

• *Source* the script – as if you typed in the commands yourself into the current shell

• Make the file executable – a new shell is spawned and the new process is a child of the current (parent) shell
Executing Shell Scripts

Source

.

file  (Note – that is a “dot” and a space)

Each command in the script is interpreted by the current shell.

All variables created are incorporated into the current shell.

All variables modified affect the current shell.

Very useful for start-up scripts
Basic UNIX
Processes and Shells

Executing Shell Scripts

Execute

`chmod 755 file
./file`

A new process is started with a new shell.

Variables created by this child will never be available to the parent.

Variables from the parent, however, are inherited by the child.
Processes and Shells

Processes Encore

Processes can be run in the *background* or the *foreground* of a shell.

Background processes are *batch* processes that must not require terminal input.

Foreground processes run interactively and will block any other input to your current shell until they finish.
Processes and Shells

Processes Encore

By default, commands or scripts started from the terminal start in the foreground.

To background a process, place an ampersand (&) after the command when you run it.

Exercise

Start a clock in the background

# xclock &
Basic UNIX
Processes and Shells

Processes Encore

In the **bash** shell, the **jobs** command will show you the list of background processes associated with the current shell.

To bring a background process to the foreground, use the **fg** command with the **jobid** number given by the **jobs** command:

```
<lister> fg %1
```
Exercise

Bring your clock process back to the foreground and kill it

```
# jobs
# fg %1 (or whatever job number it is)

Enter a CTRL-C
```
Basic UNIX
Processes and Shells

Start-up Scripts

Start-up scripts are useful scripts you can place in all user's home directories to create a common environment.

Typically, a start-up script will call other scripts to create variables:

Excerpt from */etc/profile*

```
for i in /etc/profile.d/* .sh; do
  if [ -r $i ]; then
    . $i
  fi
done
unset i
```
Basic UNIX

Processes and Shells

The **ps** Command

The **ps** command shows processes currently running on your computer. Which processes are shown depends on the options used with the command:

- No options – show only processes associated with the current shell
- `-A` – show all processes
- `-l` – long listing
- `-aux` – the options I use the most