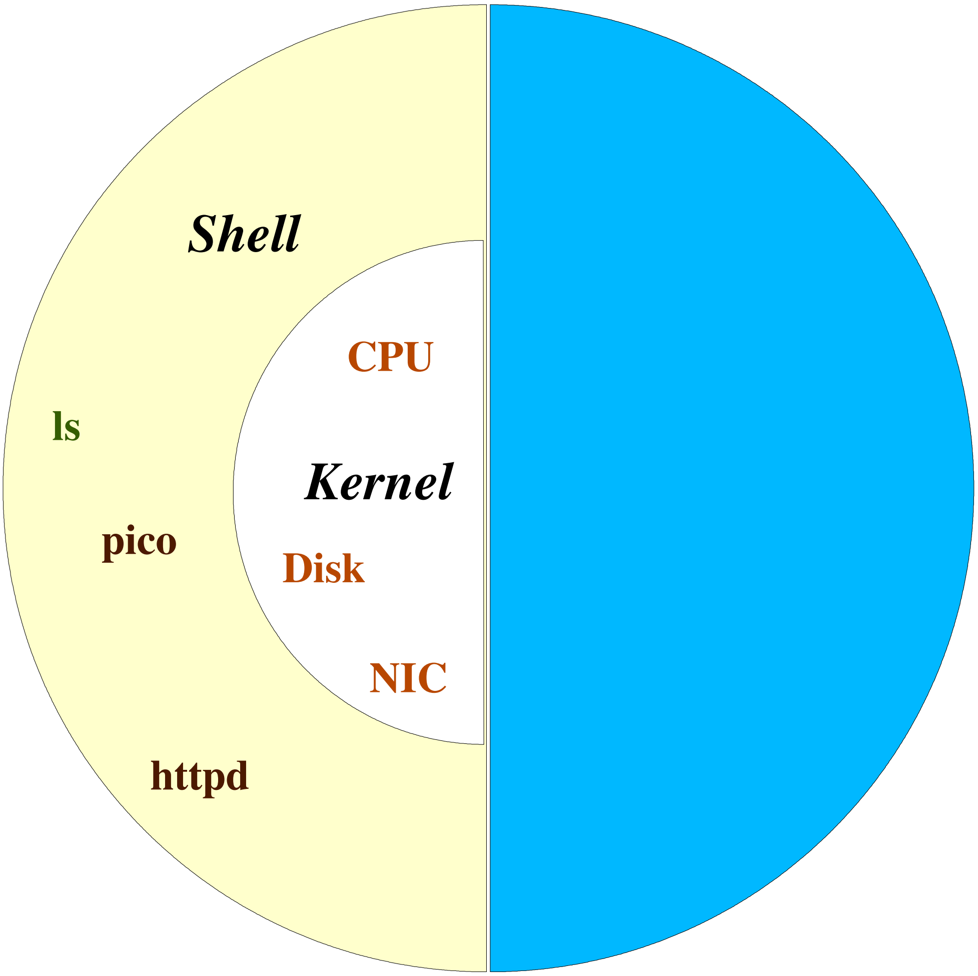
Basic UNIX

Processes and Shells



Basic UNIX

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

Basic UNIX

Processes and Shells

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 – grrrrrrr ep
- chmod – chaaa mod
- chown – chaa own
- www – wu wu wu

The **echo** Command

The **echo** *command* and the **echo** *variable* are very useful for debugging scripts. The e**cho** command prints the value of an expression (to the screen by default)

```
<lister> echo Hello World!
Hello World!
```

The **-n** option suppresses newlines:

Processes and Shells

Exercise

Run the following script:

```
# cd /opt/exercises/Shells
# ./progress.csh
```

```
 #!/bin/csh
 @ i = 1
 while ( $i < 12 )
      echo -n '.'
      sleep 1
     @ i++
end
```

Basic UNIX

Processes and Shells

The **echo** Command

The **echo** *variable* is a *toggle* variable (more on this later) which echos each shell script line to the screen before it is executed

Exercise

Run this script:

```
# ./echotoggle.csh
```

```
#!/bin/csh
set echo
echo
echo Here is a listing of the files
echo
ls -l
```

Basic UNIX

Processes and Shells

Shell Variables

Two Types of Variables:

- Local (local scope)
    - Logical – *toggle* variables which take on true/false values
    - String – contain characters
    - Numeric – contain numbers and may be used as numbers
    - Arrays – indexed collection of string values

- Environment (global scope)

  May only hold string values

Shell Variable Assignment

- Local Variables
  - Logical

    set *variable*
  - String

    set *variable*=<value>
  - Numeric

    @ *variable*=<value>
  - Arrays

    set ***variable*=**(string1...stringn)

- Environment

  setenv *variable* <value>

Accessing Variables

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

```
<lister> echo $PATH
```

Numeric and array variables have exceptions to this

When preceded by an @, numeric variables are treated like numbers

```
<lister> @ counter++
```

Basic UNIX

Processes and Shells

Accessing Variables

For arrays:

- $myarray – returns the full contents of the array “myarray”
- $myarray[2] – returns the second element of the array

Basic UNIX

Processes and Shells

Exercise

Run this script:

```
# ./array.csh
```

```
#!/bin/csh
set array=(bob ted carol alice)
echo $array
echo $array[1]
```

Processes and Shells

Exercise

Run this script:

# ./variables1.csh

```
#!/bin/csh
setenv GREETING Hello
set there=there
set friends=(Kevin Lisa Joanne)
echo $GREETING $there $friends
echo $GREETING $friends[3]
```

Blanks and Quotes

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.

Basic UNIX

Processes and Shells

Quotes and Substitution

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:

```
# ./variables2.csh
```

```
#!/bin/csh
setenv GREETING Hello
set there=there
set friend1=Kevin
set friend2=Lisa
set friend3=Joanne
set friends="$friend1 $friend2 $friend3"
echo $GREETING $there $friends
echo $GREETING '$friends'
```

Basic UNIX

Processes and Shells

Listing Defined Variables For Your Current Shell

For local variables, use the **set** command with no argument

For environement variables, use the **env** and **printenv** variables.

Exercise

Get a listing of the current shell variables

```
# set
# printenv
```

Basic UNIX

Processes and Shells

Some Common Shell Variables

- **PATH** – directory paths to search for commands
- **HOST** – the name of the computer
- **LOGIN** – the user id of the user running this shell
- **SHELL** – the shell currently being used
- **tty** – the pseudo terminal on which you are connected
- **term** – the type of terminal being used
- **prompt** – the prompt to print when then shell is ready for another command

Deassigning Variables

For local variables, use the **unset** command

unset *variable*

For environment variables, use the **unsetenv** command

unsetenv *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
- **$argv** – array of command line arguments
- **$#argv** – number of elements in argv array

(Actually, **$#** returns the number of arguments for any array variable)

Basic UNIX

Processes and Shells

Exercise

Run the following script:

```
# ./clargs.csh Hello World
# ./clargs.csh Hello
```

```
#!/bin/csh
echo $#argv
echo $0
echo $1
echo $argv[2]
```

Basic UNIX

Processes and Shells

The *status* Variable

The *status* 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

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?ffo 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

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)

Basic UNIX

Processes and Shells

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*

- **>&** *file* – direct stdout AND stderr to *file*

- *Command1* **|** *command2* – connects stdout of *command1* to stdin of *command2* via a **pipe**

Processes and Shells

Exercise

Run the following script:

```
# ./redir.csh
```

```
#!/bin/csh
cd /root
ls -a > /tmp/ls
echo < /tmp/ls
cat /tmp/ls | grep csh
```

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/csh
set files=`ls`
echo $#files
echo $files
```

Basic UNIX

Processes and Shells

Exercise

Run the following script:

```
# ./bt.csh
```

```
#!/bin/csh
set files=`ls`
echo $#files
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:

token **operator** *token*

where *token* is usually a variable or a constant.

Types of operators:

- Numeric
- Logical

Basic UNIX

Processes and Shells

Numeric Expressions

Numeric expressions are always signaled with the use of the @:

Numeric operators include +,-,*,/,% and ++ and --

Example:

```
#!/bin/csh
@ i=1
echo $i
@ i+=2
echo $i
@ i=$i + 3
echo $i
@ i++
echo $i
```

Basic UNIX

Processes and Shells

Exercise

Run this script:

```
# ./math.csh
```

```
#!/bin/csh
@ i=1
echo $i
@ i+=2
echo $i
@ i=$i + 3
echo $i
@ i++
echo $i
```

Basic UNIX

Processes and Shells

Logical Expressions

Logical expressions are almost always used with conditional statements.

Logical operators include

- ||, &&, |, &
- ^
- ==, !=, =~, !~
- <=, >=, <, >

Basic UNIX

Processes and Shells

Logical Operators

- || - Booean OR
- && - Boolean AND
- == - equivalent
- != - not equivalent
- =~ - matches
- !~ - does not match
- <=, >=, <, > - numeric comparison

Examples:

- $i <= 10
- $file =~ *pid
- “$1” == “dostats”

Control Statements

Logical expressions can be used with four *control* statements to direct the flow of execution:

- **if..then..else if..then..endif**
- **while..end**
- **foreach..end**
- **switch..case..endsw**

Basic UNIX

Processes and Shells

Control Statements
if statement

```
if (logical expression) then
.
.
.
else if (logical expression) then
.
.
.
else
.
.
.
endif
```

Basic UNIX

Processes and Shells

Exercise

Run the following script:

```
# ./if.csh
```

Enter a CTRL-C and then CTRL-D

Then run it again with just CTRL-D

```
#!/bin/csh
/bin/csh
set st=$status
if ( $st == 0 ) then
   echo "Success!"
else if ( $st == 1 ) then
   echo "I'm a failure!"
endif
```

Basic UNIX

Processes and Shells

Control Statements

switch statment

```
switch (string)
case (str1):
.
.
breaksw
case (str2):
.
.
breaksw
default:
.
breaksw
ensw
```

Basic UNIX

Processes and Shells

Exercise

Run this script:

# ./switch.csh -d

```
#!/bin/csh
@ argn=1
@ argc=$#argv
while ( $argn <= $argc )
   switch ($argv[$argn])
   case '-d':
     echo debugging
     set debug
     breaksw
   case '-c':
     echo compiling
     set compile
     breaksw
   default:
     set file=$argv[$argc]
   endsw
   @ argn++
end
```

Basic UNIX

Processes and Shells

Control Statements

foreach statment

foreach *variable* (*wordlist*)

.

.

.

end

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:

```
# ./foreach.csh
```

```
#!/bin/csh
set files=`ls -a`
echo $#files
foreach file ($files)
   echo $file
end
```

Basic UNIX

Processes and Shells

Control Statements

while statment

```
while (logical expression)
.
.
.
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.csh
```

```
#!/bin/csh
set files=`ls -a`
set numfiles=$#files
@ fnum=1
while ($fnum <= 4)
 echo "$fnum - $files[$fnum]"
 @ fnum++
end
```

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 processs is a *child* of the current (*parent*) shell

Basic UNIX

Processes and Shells

Executing Shell Scripts

Source

source *file*

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.

Basic UNIX

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

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
```

Basic UNIX

Processes and Shells

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/csh.cshrc

```
if ( -d /etc/profile.d ) then
set nonomatch
    foreach i ( /etc/profile.d/*.csh )
        if ( -r $i ) then
        source $i
        endif
    end
    unset i nonomatch
endif
```

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