ELEC 377 – Operating Systems

Week 7 – Class 2

Last Class

- Virtual Memory
◊ Page Replacement Algorithms
◊ Frame Allocation
◊ Thrashing
◊ Working Set

Today

- Shell Programming
  - ◊ Important for Testing
  - ◊ Upper level classes (CISC 327, ELEC 498)
- System Programs
  - ◊ extension of 2nd week.
  - ◊ Shell and related tools...

Shells

- We will be covering the bash shell.
  - ◊ there are other shells... tcsh, zsh, ash
  - ◊ different languages, sort of like C vs Java vs Pascal
  - ◊ located in /bin/bash

- general structure of commands.
  - ◊ *command arg1 arg2 arg3 arg4*
  - ◊ first word on the line is the command
  - ◊ other words are arguments
  - ◊ arguments starting with ‘-’ are flags or switches

Commands

- examples
  - ◊ cd lab3 *change directory to lab3*
  - ◊ ls -a *list everything including hidden files*
  - ◊ echo foo *prints the arguments to the terminal*

  -- output is “foo”
- some commands have multiple behaviours depending on style of flags. Ex Process status
  - ◊ ps auxww *no ‘-’ → bsd behaviour*

    list everyting in wide format
  - ◊ ps -ef adding ‘-’ → system V behaviour

    list everything in different format

Flags

- Some flags are multi character, some are single character
  - ◊ find . -print
  - ◊ ls -l
  - ◊ ls -lt *most commands allow single character flags to be combined (ls -l -t)*

- some flags require more information
  - ◊ find . -name ‘*.x’ -print
    -name requires a pattern

Wild Cards

- arguments for files can contain wild cards.
  - ◊ simple regular expressions
  - ◊ * → any sequence of zero or more characters
  - ◊ ? → any single character
  - ◊ [xyz] → x or y or z
  - ◊ [a-z] → any character from a to z

  - ◊ rm *~ *remove all editor backup files*
  - ◊ rm * ~ *remove all files and the file named '~'*

Quoting

- spaces separate arguments, no space, no arguments
  - ◊ cd/home — *looks for command called ‘cd/home’ not the command ‘cd’ with the argument ‘/home’*
  - How do you put spaces in an argument then?

◊ “quotes”

◊ cd “foo bar” — *change to a directory that has a space in the name*

Variables

- Like most computer languages, the shell language has variables...
  - ◊ destDir=/home/student/trd/lab4/dest
    creates the variable if it does not exist
    value is the string "/home/student/trd/lab4/dest"
  - ◊ echo $destDir
    $ used to access the value of a variable
    -- outputs '/home/student/trd/lab4/dest'
  - ◊ echo destDir
    no $, no variable value
    -- outputs destDir

Environment Variables

- variables are local
  - ◊ variables are not visible to subprocesses
  - ◊ one shell can start another
  - - variables not visible to nested shells (called *subshells*)
  - ◊ export FOO=”bar”

called an environment variable
traditionally named in all CAPS (but not necessary)
variable is visible to all subprocesses (passed through exec system call)

Environment Variables

- special environment variables
  - ◊ PATH *list of directories to find commands separated by ':' e.g. "/usr/bin:/usr/local/bin/:."*
  - ◊ HOME *your home directory*
  - ◊ SHELL *the current shell*
  - ◊ USER *your user name*
  - ◊ PWD *the current directory*
  - ◊ SHLVL *the number of nested shells (subshells).*

Subshells

- starting a subshell

```
student@e377:~$ echo $SHLVL
1
student@e377:~$ bash
student@e377:~$ echo $SHLVL
 2
student@e377:~$ exit
exit
student@e377:~$ echo $SHLVL
1
student@e377:~$
```

Quoting (revisited)

- What if we want to pass a ‘$’ as an argument
- What if we want to pass a * or ? as an argument
- Two different types of quotes -- double and single
  - ◊ foo=bar
  - ◊ echo “$foo” *stuff inside double quotes is evaluated*

bar

◊ echo ‘$foo’ *stuff in single quotes is not evaluated*

$foo

◊ echo ‘?’

?

Quoting (revisited)

- the entire argument does not need to be quoted
  - ◊ foo=bar
  - ◊ echo $foo’$’”$foo”xyzzy
  bar$barxyzzy
  - ◊ echo $fooxyzzy *produces nothing.*
  why?

Variable Manipulation

- {} similar to double quotes
  ◊ foo=bar
  ◊ echo ${foo}xyzzy
  barxyzzy
  ◊ other operations understood in {}
  ◊ ${var:-value} *if var is empty use value*
  ◊ ${foo:-xyzzy}
  bar
  ◊${bat:-xyzzy} *assuming bat is empty*
  xyzzy

Variable Manipulation

- {} similar to double quotes
  - ◊ ${#var} *length of value in var*
  - ◊ ${var/pattern/replacement}
    replace first pattern with replacement
  - ◊ ${var//pattern/replacement}
    replace every pattern with replacement
  - ◊ foo=barr
  - ◊ echo ${#foo}
  4
  - ◊ echo ${foo/r/t}
  batr
  - ◊ echo ${foo//r/t}
  batt

Initialization

- several files contain commands that are executed whenever a new shell starts
  - ◊ .bash_profile (read whenever a login shell starts)
  - ◊ .bashrc (read whenever any other bash shell starts i.e. subshells)

Redirection

- In unix/Linux, all processes have three files open by when they start
  - ◊ stdin (file decscriptor 0) is by default a read only file connected to the terminal window
  - ◊ stdout (file descriptor 1) is by default a write only file connected to the terminal window
  - -- used for the ‘normal’ output of the command
  - ◊ stderr (file descriptor 2) is by default a write only file connected to the terminal window
  - -- used for error and other non normal messages

Redirection

- Can redirect the default files from the terminal window and hook them up to files.
  - ◊ ls > lsout.txt *put the normal output of ls in the file 'lsout.txt'*
  - ◊ mycmd < infile.txt*read input for my cmd from the file 'infile.txt'*
  - ◊ mycmd 2> erout.txt *put the error output of mycmd in the file erout.txt*
  - ◊ mycmd > out.txt 2>&1

    put the output and error output of mycmd in out.txt (2>&1 means point 2 at the same file as 1)

Pipes

- What if we want the output of one program to be the input of another command?
  - ◊ ls > out.txt *put the normal output of ls in the file 'lsout.txt'*
  - ◊ grep 'foo' < out.txt *search for the string foo in the output of ls*
  - ◊ ls | grep 'foo' *hook the output of ls to the input of grep 'foo'*
  - ◊ ls 2>&1 | grep 'foo' *hook the output and error of ls to the input of grep 'foo'*
  - ◊ /dev/null is a file that throws away everything stored in it. Good way to ignore output.

Scripts

- a file containing commands to be executed
  ◊ hello.sh contains:
  #!/bin/bash
  string=”Hello World”
  echo $string
- if file has execution bit set (chmod +x filename) then can be executed as a command...
  ◊ ./hello.sh
  Hello World
- Otherwise, must call shell explicitly
  ◊ bash hello.sh
  Hello World

Shell Scripts

- first line of the script is a magic number line
- *exec* system call looks at first few bytes of any file that is executable that is passed as a command.
  - ◊ if the first four bytes are the magic number 0x7f E L F then the file contains binary code
  - ◊ if the first two bytes are '#!' then the rest of the first line of the file is the path to a command that be executed, and passed the name of the file as the first argument.
  - ◊ for shell scripts, pass the path to the shell you wish to execute (i.e. /bin/bash).
  - ◊ if missing, then assumes the default shell....

Shell Scripting

- we have already talked about assigning variables, reading variables, running programs.
- Other language features
  - ◊ special variables and expressions
  - ◊ control statements (if, while, for, switch, etc)
  - ◊ functions

Shell Variables

- special shell variables
  - ◊ $0 *name of the shell script*
  - ◊ $# *number of arguments*
  - ◊ $1 *the first argument* (same for $2, $3 ...)
  - ◊ $* *all arguments separated by spaces*
  - ◊ $? *exit code of last command executed*
    0 means success, anything else means fail

% myshell a b c d e
$0 = myshell, $#= 5 $1 = a, $2 = b, … $*=”a b c d e”

Control flow - if statement

if *test command* ; **then**
   true commands
elif *another test command* ; **then**
   otherwise true commands
else
   false commands
fi

◊ check the exit status of the test commands to determine if the statements are executed

Control flow - if statement

```
if test -f xxx ; then                    xxx exists and is a file
   cat xxx
elif test -d yyy ; then
   ls yyy
else
   echo “Neither xxx nor yyy exist!!”
fi
```

- test -f *name* returns 0 (success) if file *name* exists, 1 otherwise.
- test -d *name* returns 0 (success) if directory *name* exits, 1 otherwise

Control Flow - Arithmetic

```
if (( $LIFE == 42 )) ; then
   echo “everything”
else
   echo “:-(”
fi
```

- [[7 < 22]] *false - string comparison*
- ((7 < 22)) *true - numeric comparison*
- ((X = 5 + 2)) *assignment allowed, 0 = false, !0 = true*

Control Flow - while

```
((i = 1))
while (( i < 10)) ; do
   echo $i
  (( i++ ))
done
```

- note that ((...)) does not *require* $ in front of variables (also in if statements too!!)
- [[ ]] and commands with exit status can also be used for while condition.

Control Flow - for

for var **in** wordlist ; **do** commands; **done**

for p **in** /proc/[0-9]* ; **do**
  echo -n “$p: “ *-n = no newline, also space in string*
  grep ‘State’ $p/status *find line with State*
done

output:

/proc/1: State: S (sleeping)

....

Control Flow - for

shift shifts the position arguments
$2 → $1, $3 → $2, $4 → $3, etc.

```
while (( $# > 0 ))
do
    echo $1
    shift
done
```

Find Command

- find command
  - ◊ finds files or directories that match a pattern
  - ◊ find /home/student -name ‘*.c’ -or -name ‘*.h” -print

/home/student/trd/lab0/lab0mod.c
/home/student/trd/lab0/lab0user.c
/home/student/trd/lab1/lab1.c
....
/home/student/trd/lab4/common.h
...

For and Find Command - Friends!!

- want to print all c (.c and .h) files in a particular directory
  - ◊ lpr *filename* *prints a file (or stdin if no file)*
  - ◊ a2ps filename *converts file to postscript*

```
for i in `find /home/student -name ‘*.c’ -or -name ‘*.h” -print`
do
    a2ps $i | lpr          create a postscript version and print
done
```

◊ $(command) same as `command`

The path least taken…

- two other useful commands
  - ◊ dirname *path* *directory name of path*
  - ◊ dirname /a/b/c/defg.c
    /a/b/c
  - ◊ dirname defg.c
    .

◊ basename path extension *base name of file*
◊ basename /a/b/c/defg.c .c
defg