CSC275 – Operating Systems Practicum

Lecture 6 – Using The Shell

What is **`bash`**?

- **`bash`** (Bourne Again Shell) is designed as a free replacement of the Bourne shell (the original UNIX shell) to be bundled with Linux.
- It is freely available from the GNU project.
- It is considered a superset of the Bourne shell.

Starting Up **bash**

- When you log in on a Linux system, it starts for chosen shell; if there is none, it starts **bash**.
- It will execute the commands in **.bash_profile**; on Panther, it sources **/etc/profile** (it executes it within the current shell using its environment).

.bash_profile on Panther

```
SIEGFRIE@panther:~$ more .bash_profile
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then #If .bashrc is in $HOME
        . ~/.bashrc     # execute .bashrc
fi
# User specific environment and startup programs

PATH=$PATH:$HOME/bin    #Add $HOME/bin to the path
export PATH             # export PATH so it's
                        # available to subshells
unset USERNAME          # Remove  USERNAME from the
                        #  namespace
```

The Initialization Files

- The initialization files include:
  - **~/.bash_profile** – executed when logging in.
  - **BASH_ENV** or **.bashrc** – contains bash variables and aliases
  - **/etc/bashrc** – System-wide version of **.bashrc**
  - **~/.profile** – System-wide version of .bash_profile
  - **~/.bash-logout** – executed when logging out.
  - **.inputrc** – another default initialization file.

/etc/profile

```
# /etc/profile: system-wide .profile file for the
# Bourne shell (sh(1)) and Bourne compatible shells
# (bash(1), ksh(1),# ash(1), ...).

if [ "$PS1" ]; then
  if [ "$BASH" ] && [ "$BASH" != "/bin/sh" ]; then
    # The file bash.bashrc already sets the default
    # PS1.
    # PS1='\h:\w\$ '
    if [ -f /etc/bash.bashrc ]; then
      . /etc/bash.bashrc
    fi
```

```
else
    if [ "`id -u`" -eq 0 ]; then
      PS1='# '
    else
      PS1='$ '
    fi
  fi
fi

# The default umask is now handled by pam_umask.
# See pam_umask(8) and /etc/login.defs.

if [ -d /etc/profile.d ]; then
```

Command Line Structure

- The simplest command is a single word:

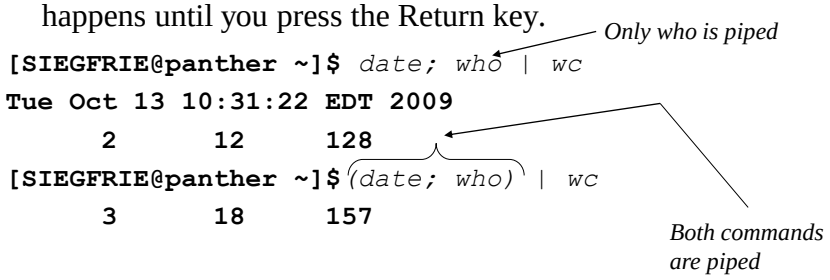
```
[SIEGFRIE@panther ~]$ who
don      pts/0     Oct  6 12:05 (10.80.4.78)
SIEGFRIE pts/2     Oct 13 10:30 (pool-….verizon.net)
```

- A command is terminated with a semi-colon or a newline

```
[SIEGFRIE@panther ~]$ date;
Tue Oct 13 10:31:09 EDT 2009
[SIEGFRIE@panther ~]$ date; who
Tue Oct 13 10:31:14 EDT 2009
don      pts/0     Oct  6 12:05 (10.80.4.78)
SIEGFRIE pts/2     Oct 13 10:30 (pool-….verizon.net)
```

Terminating Commands

- Even if you terminate a command with a semi-colon, nothing happens until you press the Return key.



"Tapping" The Pipe

- The data going into the pipe can be "tapped" using the tee command. It saves intermediate results in a file while still passing them on.

```
[SIEGFRIE@panther ~]$ (date; who) | tee save | wc
      3      18     157
[SIEGFRIE@panther ~]$ cat save
Tue Oct 13 10:47:45 EDT 2009
don      pts/0     Oct  6 12:05 (10.80.4.78)
SIEGFRIE pts/2     Oct 13 10:30 (pool-….verizon.net)
[SIEGFRIE@panther ~]$ wc < save
  3  18 157
[SIEGFRIE@panther ~]$
```

&

- **&** is also a command terminator. It is used for running long-running processes in the background.

```
$ long-running-command &
5273  ← pid
$
```

- This gives us new ways to use background processes:

```
[SIEGFRIE@panther ~]$ sleep 5
[SIEGFRIE@panther ~]$
[SIEGFRIE@panther ~]$ (sleep 5; date)& date
[1] 31257
Tue Oct 13 11:12:45 EDT 2009
[SIEGFRIE@panther ~]$ Tue Oct 13 11:12:50 EDT 2009
```

UNIX Commands and Arguments

- Most UNIX commands will accept arguments, which are variously words separated by white space. These string may be interpreted in any manner that the program sees fit.
- Example (*How are these arguments used?*)

```
pr file
echo Hello junk
echo Hello > junk
```

< > | ;

- **< > | ;** and **&** are special characters and can appear anywhere on a line.
- Example

```
echo Hello > junk
echo > junk Hello
> junk echo Hello
```

all work

Metacharacters

- The shell gives special recognition to certain characters.
- **echo *** - displays the name of every file in the working directory.
- It doesn't include the files whose name begins with a period to avoid including . and .. – if want to print these as well, we have to add

```
ls .*
```

Metacharacters – An Example

```
[SIEGFRIE@panther junk]$ ls -ld .*
drwxr-xr-x  2 SIEGFRIE users 4096 Oct 20 08:36 .
drwx--x--x 17 SIEGFRIE users 4096 Oct 20 08:34 ..
-rw-r--r--  1 SIEGFRIE users   36 Oct 20 08:36 .mybad
[SIEGFRIE@panther junk]$ echo *
cookie temp
[SIEGFRIE@panther junk]$ echo .*
. .. .mybad
[SIEGFRIE@panther junk]$
```

Metacharacters As Ordinary Characters

- There has to be some way of using metacharacters as regular characters; there are just too many to be completely successful in avoiding them.
- Example

```
[SIEGFRIE@panther junk]$ echo '***'
***
[SIEGFRIE@panther junk]$
```

"

- You can use double quotation marks ("), but these will be checked for **$** **`**…**`** and ****. (These will be discussed later).
- We can put a backslash (\) in front of each metacharacter:

  echo *** *In shell terminology it's still a word.*

Quotes – Some More Examples

```
[SIEGFRIE@panther ~]$ echo "Don't do that"
Don't do that
[SIEGFRIE@panther ~]$ echo x'*'y
x*y
[SIEGFRIE@panther ~]$ echo '*'A'?'
*A?
[SIEGFRIE@panther ~]$ echo 'hello,
> world'
hello,
world
[SIEGFRIE@panther ~]$
```

secondary prompt

*

- In the command

  `echo x*y`

  x*y is replaced by all the filenames beginning with x and ending with y.
- This substitution is performed by the shell – **`echo`** is not involved in this.

* - An Example

```
[SIEGFRIE@panther junk]$ ls x*y
ls: x*y: No such file or directory
[SIEGFRIE@panther junk]$ cat >xyzzy
The rain in Spain stays mainly in the Plain
[SIEGFRIE@panther junk]$ ls x*y
xyzzy
[SIEGFRIE@panther junk]$ ls 'x*y'
ls: x*y: No such file or directory
[SIEGFRIE@panther junk]$
```

\ and

- A backslash (\) at the end of a line causes the line to be continued:

```
[SIEGFRIE@panther junk]$ echo abc\
> def\
> ghi
abcdefghi
[SIEGFRIE@panther junk]$
```

- # is used for comments in a command:

```
[SIEGFRIE@panther junk]$ echo hello #there
hello
[SIEGFRIE@panther junk]$ echo hello#there
hello#there
[SIEGFRIE@panther junk]$
```

begins a shell "word"

doesn't begin a shell "word"

Shell Metacharacters

>	*prog* **>** *file* direct standard output to *file*
>>	*prog* **>>** *file* append standard output to *file*
<	*prog* **<** *file* take standard input to *file*
|	p_1 **|** p_2 connect standard output of p_1 to standard input of p_2
< *str*	*here document:* standard input follows, up to next *str* on a line by itself.
*****	match any string of zero or more characters in filenames.
?	match any single character in filenames.
[*ccc***]**	match any character from ccc in filenames; ranges from **0-9** or **a-z** are legal.
;	command terminator: p_1; p_2 does p_1, then p_2
&	like ; but doesn't wait for p_1 to finish

Shell Metacharacters (continued)

`` `…` ``	run command(s) in …; output replaces `` `…` ``
(…**)**	run command(s) in …; in a subshell
{…**}**	run command(s) in …; in current shell (rarely used)
$1, $2, *etc.*	**$0..$9** replaced by arguments to shell file
$var	value of shell variable *var*
${*var***}**	value of shell variable *var* avoids confusion when concatenating text
****	take character *c* literally, *newline* discarded
'…**'**	take … literally
"…**"**	take … literally after **$**, **`…`** and **** are interpreted,
#	if **#** starts a word, rest of line is a comment

Shell Metacharacters (continued)

*var***=***value*	assign *value* to variable *var*
p_1 **&&** p_2	run p_1 if successful run p_2
p_1 **||** p_2	run p_1 if unsuccessful run p_2

echo

- There are 2 different versions of **echo**, one with a newline at the end and one without a newline at the end.

```
[SIEGFRIE@panther junk]$ echo Enter a command:
Enter a command:
[SIEGFRIE@panther junk]$ echo -n Enter a command:
Enter a command:[SIEGFRIE@panther junk]$
```

Creating New Commands

- Let's create our own commands, to include a series of UNIX commands in combination:

```
[SIEGFRIE@panther junk]$ who | wc -l
4
[SIEGFRIE@panther junk]$
```

- Let's create a file containing this command line:

```
[SIEGFRIE@panther junk]$ echo 'who | wc -l' > nu
[SIEGFRIE@panther junk]$
```

(What would **nu** *contain if we omitted the "?)*

Using A New Command

```
[SIEGFRIE@panther junk]$ who
CHAYS    pts/0  Oct 20 08:07 (10.84.10.219)
SIEGFRIE pts/2  Oct 20 11:16 (pool-… .verizon.net)
HOBSON   pts/4  Oct 20 09:59 (ool-… optonline.net)
MOHAMMED pts/5  Oct 20 13:06 (10.2.19.249)
[SIEGFRIE@panther junk]$ cat nu
who | wc -l
```

- **sh** is the shell – we can run it and redirect it input:

```
[SIEGFRIE@panther junk]$ sh <nu
4
```

- **sh**, like any other program can take input from command-line parameters:

```
[SIEGFRIE@panther junk]$ sh nu
4
```

- This will work for **bash** as well:

```
[SIEGFRIE@panther junk]$ sh <nu
4
```

Using A New Command

- Just creating the file doesn't mean that it is executable

```
[SIEGFRIE@panther junk]$ nu
-bash: nu: command not found
[SIEGFRIE@panther junk]$ chmod +x nu
[SIEGFRIE@panther junk]$ nu
-bash: nu: command not found
```

- You have to give it execute permission AND move it into the bin directory.

```
[SIEGFRIE@panther junk]$ mv nu ../bin
[SIEGFRIE@panther junk]$ nu
4
[SIEGFRIE@panther junk]$
```

Command Arguments and Parameters

- What if we want to a shorthand for

  `chmod +x` <*whatever*>

  (After all, how many times will we change nu's permissions?)
- **`$1`** is the first argument, **`$2`** is the second argument, etc.
- Now we can write

  `chmod +x $1`

Creating A Command – An Example

```
[SIEGFRIE@panther junk]$  echo 'chmod +x $1' > cx
[SIEGFRIE@panther junk]$ sh cx cx
[SIEGFRIE@panther junk]$ echo echo Hi there > hello
[SIEGFRIE@panther junk]$ ls -l hello
-rw-r--r--  1 SIEGFRIE users 14 Oct 20 14:23 hello
[SIEGFRIE@panther junk]$ mv cx ../bin
[SIEGFRIE@panther junk]$ cx hello
[SIEGFRIE@panther junk]$ ls -l hello
-rwxr-xr-x  1 SIEGFRIE users 14 Oct 20 14:23 hello
[SIEGFRIE@panther junk]$ hello
-bash: hello: command not found
[SIEGFRIE@panther junk]$ mv hello ../bin
[SIEGFRIE@panther junk]$ hello
Hi there
[SIEGFRIE@panther junk]$
```

$*

- What if we wanted to make several shell scripts executable at once?

  chmod +x $1 $2 $3 $4

  is very clumsy. And there are no more than 9 arguments explicitly numbered that are allowed. (**$10** is **$1** with a **0** appended to it.)
- Try

  chmod +x $* #all files in directory

$* - An Example

```
[SIEGFRIE@panther junk]$ chmod +x lc
[SIEGFRIE@panther junk]$ mv lc ../bin
[SIEGFRIE@panther junk]$ cat ../bin/lc
# lc: Count number of lines in files
wc -l $*
[SIEGFRIE@panther junk]$ lc *
  1 cookie
  2 temp
  1 xyzzy
  4 total
[SIEGFRIE@panther junk]$
```

A Small Phone Book Project

- Let's put together a small telephone book; our data is:

```
dial-a-joke     212-976-3838
dial-a-prayer   212-976-4200
dial santa 212-976-3636
dow jones report    212-976-4141
```

- We can use grep to search the phonebook file.

The Phone Book Shell Script

```
[SIEGFRIE@panther junk]$ echo \
  'grep $* $HOME/junk/phone-book' >411
[SIEGFRIE@panther junk]$ cx 411
[SIEGFRIE@panther junk]$ mv 411 ../bin
[SIEGFRIE@panther junk]$ 411 joke
dial-a-joke     212-976-3838
[SIEGFRIE@panther junk]$ 411 prayer
dial-a-prayer   212-976-4200
[SIEGFRIE@panther junk]$ 411 santa
dial santa      212-976-3636
[SIEGFRIE@panther junk]$ 411 dow jones
grep: jones: No such file or directory
/home/siegfried/junk/phone-book:dow jones report
  212-976-4141
```

The Phone Book Shell Script

- The operating system sees Dow Jones as two parameters unless they are in quotes:

```
[SIEGFRIE@panther junk]$ echo \
  'grep "$*" $HOME/junk/phone-book'>411
[SIEGFRIE@panther junk]$ cx 411
[SIEGFRIE@panther junk]$ mv 411 ../bin
[SIEGFRIE@panther junk]$ 411 dow jones
dow jones report        212-976-4141
[SIEGFRIE@panther junk]$
```

- Using **`grep -y`** makes the program case-insensitive

$0

- **`$0`** is a special command parameter – a reference to the command's file itself.

```
[SIEGFRIE@panther junk]$ cat >echoecho
echo $0
[SIEGFRIE@panther junk]$ cx echoecho
[SIEGFRIE@panther junk]$ mv echoecho ../bin
[SIEGFRIE@panther junk]$ echoecho
/home/siegfried/bin/echoecho
[SIEGFRIE@panther junk]$
```

`  `
`…`

- Any text appearing inside `…` is interpreted as a command and the output of the command becomes part of the text.
- Example

```
[SIEGFRIE@panther junk]$ echo At the tone, the\
  time will be `date`
At the tone, the time will be Tue Dec 1 11:07:14
  EST 2009
[SIEGFRIE@panther junk]$
```

More About `…`

- This is interpreted correctly ***EVEN*** inside "…"

```
[SIEGFRIE@panther junk]$ echo "At the tone
> the time will be `date`"
At the tone
the time will be Tue Dec  1 11:12:00 EST 2009
[SIEGFRIE@panther junk]$
```

Shell Variables

- **$1**, **$2**, **$3**, .. are positional variables, where the digit indicates the position on the command line.
- Other shell variables indicate other things
  - **HOME** – home (or login) directory
  - **PATH** – collection of directories searched for an executable file.

Example of Shell Variables - **PATH**

```
[SIEGFRIE@panther junk]$ echo $HOME
/home/siegfried
[SIEGFRIE@panther junk]$ echo $PATH
/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/
  X11R6/bin
[SIEGFRIE@panther junk]$ set \
  PATH=$PATH:/home/siegfrie/bin
[SIEGFRIE@panther junk]$ echo $PATH
/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/
  X11R6/bin/home/siegfrie/bin
```

- There cannot be blanks on either side of the **=** and there cannot be blanks in the assigned string. Place quotation marks if necessary.

Other Shell Variables

- There are other variables that are not special to the shell:

```
[SIEGFRIE@panther junk]$ di=`pwd`
[SIEGFRIE@panther junk]$ set
BASH=/bin/bash
… …
di=/home/siegfried/junk
[SIEGFRIE@panther junk]$ mkdir morejunk
[SIEGFRIE@panther junk]$ pwd
/home/siegfried/junk
[SIEGFRIE@panther junk]$ cd ../java
[SIEGFRIE@panther java]$ ls $di
411  cookie  morejunk  oreo  phone-book  temp
  xyzzy
[SIEGFRIE@panther java]$ cd $di
[SIEGFRIE@panther junk]$ pwd
/home/siegfried/junk
[SIEGFRIE@panther junk]$
```

`set`

- `set` displays the values of the various system variables:

```
[SIEGFRIE@panther junk]$ set
BASH=/bin/bash
…
HOME=/home/siegfried
HOSTNAME=panther.adelphi.edu
HOSTTYPE=i686
…
PATH=/usr/kerberos/bin:/usr/local/bin:/bin:/usr/b
  in:/usr/X11R6/bin:/home/siegfried/bin
…
PS1='[\u@\h \W]\$ '
PS2='> '
PS4='+ '
PWD=/home/siegfried/junk
…
di=/home/siegfried/junk
```

Shell Variables and Child Processes

- Variable values are associated with the shell that created it and are not passed automatically to child processes:

```
[SIEGFRIE@panther junk]$ x=Hello
[SIEGFRIE@panther junk]$ echo $x
Hello
[SIEGFRIE@panther junk]$ sh
sh-3.00$ x=Goodbye
sh-3.00$ echo $x
Goodbye
sh-3.00$ ^d
[SIEGFRIE@panther junk]$ echo $x
Hello
[SIEGFRIE@panther junk]$
```

Shell Variables and Subshells

- You cannot normally modify or use shell variables in subshells, including shell scripts (files containing commands).
- You can use the command . to tell the script to use the parent process's variables.
  - NB – This does not allow the use of command-line parameters **`$1`**, **`$2`**, etc.

. – An Example

```
[SIEGFRIE@panther junk]$ cat >ddi
echo $di
[SIEGFRIE@panther junk]$ cx ddi
[SIEGFRIE@panther junk]$ mv ddi $HOME/bin
[SIEGFRIE@panther junk]$ ddi

[SIEGFRIE@panther junk]$ echo $di
/home/siegfried/junk
[SIEGFRIE@panther junk]$ . ddi
/home/siegfried/junk
[SIEGFRIE@panther junk]$
```

`export`

- The **`export`** command marks the listed variables for automatic export to subprocesses (the child processes get to use the values).
- Example

```
[SIEGFRIE@panther junk]$ export di
[SIEGFRIE@panther junk]$ ddi
/home/siegfried/junk
[SIEGFRIE@panther junk]$
```

More On Redirection

- Standard error is separate from standard output:

```
[SIEGFRIE@panther junk]$ cat >file1
This is a test of the emergency programming system.
[SIEGFRIE@panther junk]$ cat > file2
If this were a real emergency, you would be toast!
[SIEGFRIE@panther junk]$ diff file1 fiel2 \
  > diff.out
diff: fiel2: No such file or directory
[SIEGFRIE@panther junk]$
```

- **<u>This is important</u>** – error messages do not show up together with output in cases like this.

Standard I/O and File Descriptors

- All files are opened using ***<u>file descriptors</u>***, which are entries in the process's open-file table.
- All processes have 3 files open:
  - stdin – standard input – usually the keyboard
  - stdout – standard output – usually the monitor
  - stderr – standard error – the usally the monitor.

Writing to **stderr**

```
[SIEGFRIE@panther c]$ cat sss.c
#include        <stdio.h>

int     main(void)
{
  fprintf(stderr, "This is a test\n");
  return(0);
}
```

Redirecting **stderr**

```
[SIEGFRIE@panther c]$ sss
This is a test
[SIEGFRIE@panther c]$ sss > sss.out
This is a test
[SIEGFRIE@panther c]$ sss 2>sss.err
[SIEGFRIE@panther c]$ more sss.err
This is a test
[SIEGFRIE@panther c]$ sss 2>&1
This is a test
```

cpp.c – A simple copy program

```
#include        <stdio.h>

#define BUFSIZE         512
int     main(void)
{
  char    buf[BUFSIZE];
  int     n;

  while ((n = read(0, buf, BUFSIZE)) > 0)
      write(1, buf, n);

  return(0);
}
```

A More Complicated **copy.c**

```
#include     <stdio.h>

#define      BUFSIZE         512
#define      PMODE           0644

void   error(char s1[], char s2[]);

int    main(int argc, char *argv[])
{
  char       buf[BUFSIZE];
  int        fd1, fd2, bufp, n;

  if (argc != 3)  {
      error("Usage: copy <fromfile> <tofile>\n", NULL);
      exit(1);
  }
```

```
   if ((fd1 = open(argv[1], 0)) == -1)
       error("Cannot open %s", argv[1]);

   if ((fd2 = creat(argv[2], PMODE)) == -1)
       error("Cannot open %s", argv[2]);

   while ((n = read(fd1, buf, BUFSIZE)) > 0)
       if (write(fd2, buf, n) != n)
              error("copy: write error", NULL);

   return(0);
}

void    error(char s1[], char s2[])
{
   printf(s1, s2);
   printf("\n");
}
```

Shell I/O Redirection

> *file*	direct **stdout** to *file*
>> *file*	append **stdout** to *file*
< *file*	take **stdin** to *file*
p_1 **|** p_2	connect **stdout** from p_1 to **stdin** for p_2.
n **>** *file*	redirect output from file descriptor *n* to *file*
n **>>** *file*	append output from file descriptor *n* to *file*
n **>** **&***m*	merge output from file descriptor *n* with file descriptor *m*
n **<** **&***m*	merge input from file descriptor *n* with file descriptor *m*
<< *s*	Here document: take **stdin** until next *s* at beginning of a line – substitute for **$**, **`...`**, and ****
<< *****s*	here document with no substitution
<<'*s***'**	here document with no substitution