CS2042 - Unix Tools

Fall 2010

Lecture 11

Hussam Abu-Libdeh

based on slides by David Slater

October 1, 2010

Intro To Screen

The `screen` command

`screen` - a screen manager with terminal emulation

Generally `screen` can be used just as you would normally use a terminal window. However, special commands can be used to allow you to save your session, create extra shells, or split the window into multiple independent panes.

Passing Commands to screen

Each `screen` commands consists of a CTRL-a (hereafter referred to as C-a) followed by another character.

Using Screen

Attach a screen

`screen [options]`

- Opens a new screen for use
- -a : include all capabilities

Resume a screen

`screen -r [pid.tty.host]`

- Resumes a detached screen session

`screen -x [pid.tty.host]`

- Attach to a non-detached screen session

If you only have one screen, the [pid.tty.host] string is unnecessary.

Identifying Screen Sessions

Screen Listing

`screen -ls` or `screen -list`

- Lists your screen sessions and their statuses

These screen sessions are the [pid.tty.host] strings required for resuming

Resuming a screen

If `screen -ls` returns `15829.pts-9.rumman (Detached)`

- `screen -r 15829.pts-9.rumman` to resume the screen

Note: You only need to specify the full "name" of the session if you have multiple sessions open. If you just have one session, just use `screen -r`

Creating More Shells

Creates a New Shell Window

`C-a c`

- Creates a new shell in a new window and switches to it
- Useful for opening multiple shells in a single terminal
- Similar to tabbed browsing/tabbed IMs

But how do we switch between windows? (hint: every window is numbered by order of creation)

Window Selection

`C-a 1` - switches to window 1 `C-a 9` - switches to window 9

Splitting Screen

Split Screen Computing

`C-a S` - splits your terminal area into multiple panes
`C-a tab` - changes the input focus to the next pane

- The 'S' is case-sensitive
- Each split results in a blank pane
- Use `C-a c` to create a new shell in a pane
- Use `C-a <num>` to move an existing window to a pane

Note:

When you reattach a split `screen`, the split view ill be gone. Just re-split the view, then switch between panes and reopen the other windows in each with `C-a <num>`

Now lets put this together to do something useful

Suppose you are doing some serious scientific computing and want to run it on a remote server. We can put together what we have learned to do this efficiently:

- `ssh` into the remote machine

```
ssh slater@boom.cam.cornell.edu
```

- start `screen`

```
screen
```

- start mathematica

```
math < BatchJob.m
```

- renice the math kernel so other uses can use the machine

```
renice -20 PID
```

- Detach the screen, logout, and come back 8 hours later when it is done

The Other Way

If you have a **noninteractive** batch job, you can also allow it to continue to run after you logout by using `nohup`

`nohup`

`nohup command`

- command will continue to run after you logout
- output is sent to `nohup.out` if not otherwise redirected
- can be combined with `nice`

Example:

```
nohup nice -15 math < BatchJob.m &
```

Back to scripting

Some Review

What does this do?

```
#!  /bin/bash
gawk '$1 = "'$1'" {count++ ; print $2}
END { print count}' infile
```

Some Review

What does this do?

```
#!  /bin/bash
gawk '$1 = "'$1'" {count++ ; print $2}
END { print count}' infile
```

Prints the second field whenever the first matches the first argument and then prints the total number of matched lines.

Arithmetic

A little arithmetic can be useful and BASH can perform all the standard operators

Arithmetic

- a++, a– : Post-increment/decrement
- ++a, –a : Pre-increment/decrement
- a+b, a-b : Addition/subtraction
- a*b, a/b : Multiplication/division
- a%b : Modulu
- a**b : Exponential
- a>b, a<b : Greater than, less than
- a==b, a!=b : Equality/inequality
- =, +=, -= : Assignments

Using Arithmetic Expressions

We have already seen one way to do arithmetic:

Example:

```
echo $((2+5))
7
```

We can also use it as part of a larger command:

The "Let" Built-In

```
VAR1=2
let VAR2=$VAR1+15
let VAR2++
echo $VAR2
18
```

- `let` evaluates all expressions following the equal sign

The Difference

There are two major differences:

- all characters between the ((and)) are treated as quoted (no shell expansion)
- The let statement requires there be no spaces **anywhere** (so need to quote)

Example:

```
let "i=i + 1"
i=$((i + 1))
```

The `while` loop

```
while cmd
do
    cmd1
    cmd2
done
```

Executes `cmd1, cmd2` as long as `cmd` is successful (i.e. its exit code is 0).

While loop example

```
i="1"
while [ $i -le 10 ]
do
    echo "$i"
    i=$(($i+1))
done
```

This loop prints all numbers 1 to 10.

Until loop

```
until cmd
do
    cmd1
    cmd2
done
```

Executes `cmd1, cmd2` as long as `cmd` is unsuccessful (i.e. its exit code is not 0).

Until loop example

```
i="1"
until [ $i -ge 11 ]
do
    echo i is $i
    i=$(($i+1))
done
```

for loop

```
for var in string1 string2 ...  stringn
do
    cmd1
    cmd2
done
```

The for loop actually has a variety of syntax it can accept. We will look at each in turn.

for loop example

```
#!  /bin/bash
# lcountgood.sh
i="0"
for f in "$@"
do
     j=‘wc -l < $f‘
     i=$(($i+$j))
done
echo $i
```

Recall that $@ expands to all arguments individually quoted ("arg1" "arg2" etc).

This script counts lines in a collection of files. For instance to count the number of lines of all the files in your current directory just run `./lcountgood.sh *`

for loop example

What happens if we change $@ to $*? Recall that $* expands to all arguments quoted together ("arg1 arg2 arg3")

```
#!  /bin/bash
# lcountbad.sh
i="0"
for f in "$*"
do
      j=`wc -l < $f`
      i=$(($i+$j))
done
echo $i
```

This does not work! Lets look at why.

Why we don't like $*

Consider

```
#!  /bin/bash
# explaingood.sh
j=0
for i in "$@"
do
j=$(($j+1))
echo $i
done

echo $j
```

This simply echos all the files you pass to the script and how many.

```
$ ./explaingood.sh *
explainbad.sh
explaingood.sh
lcountright.sh
lcountwrong.sh

4
```

Why we don't like $*

But if we change to $*

```
#!  /bin/bash
# explainbad.sh
j=0
for i in "$*"
do
j=$(($j+1))
echo $i
done

echo $j
```

This simply echos all the files at once and the number 1:

```
$ ./explaingood.sh *
explainbad.sh explaingood.sh lcountright.sh lcountwrong.sh

1
```

other for loop syntax

We can also do things like:

```
for i in {1..10}
do
    echo $i
done
```

To print 1 to 10.

other for loop syntax

We can also do things like:

```
for i in $(seq 1 2 20)
do
     echo $i

done
```

```
1
3
5
7
9
11
13
15
17
19
```

even **more** for loop syntax!

We can also do something more traditional:

```
for (( c=1; c<=5; c++))
do
     echo $c

done
```

To print 1 to 5 (spaces around c=1 etc do not matter)

An infinite loop

We can now create infinite for loops if we want

```
for (( ; ; ))
do
     echo "infinite loop [hit CTRL+C to stop]"

done
```

can't catch a break

We can use `break` to exit `for`, `while` and `until` loops early

```
for i in some set
do
      cmd1
      cmd2
      if (disaster-condition)
      then
            break
      fi
      cmd3
done
```

continue

We can use `continue` to skip to the next iteration of a `for`, `while` or `until` loop.

```
for i in some set
do
      cmd1
      cmd2
      if (i don't like cmd3-condition)
            continue
      fi
      cmd3
done
```

Reading in input from the user

You can ask the user for input by using the `read` command

read

`read varname`

- Asks the user for input
- By default stores the input in $REPLY
- Can read in multiple variables `read x y z`
- -p option allows you to print some text

Example:

```
read -p "How many apples do you have?  " apples
How many apples do you have?  5
$ echo $apples
5
```

Other uses for read

`read` can also be used to go line by line through a file or any other kind of input:

Example:

```
cat /etc/passwd | while read LINE ; do echo $LINE done
```

Other uses for read

`read` can also be used to go line by line through a file or any other kind of input:

Example:

```
cat /etc/passwd | while read LINE ; do echo $LINE done
```

- Prints the contents of /etc/passwd line by line

```
ls *.txt | while read LINE ; do newname=$(echo $LINE |\
sed 's/txt/text/' ); mv -v "$LINE" "$(newname)" ; done
```

Other uses for read

`read` can also be used to go line by line through a file or any other kind of input:

Example:

```
cat /etc/passwd | while read LINE ; do echo $LINE done
```

- Prints the contents of /etc/passwd line by line

```
ls *.txt | while read LINE ; do newname=$(echo $LINE |\
sed 's/txt/text/' ); mv -v "$LINE" "$(newname)" ; done
```

- Renames all .txt files in the current directory as .text files.

case

`case` allows you to execute a sequence of if else if statements in a more concise way:

```
case expression in

     pattern1 )
          statements ;;
     pattern2 )
          statements ;;
     ...

esac
```

Here the patterns are expanded using **shell expansion**. We can use match one of several patterns by separated by a pipe `|`.

superficial example

```
$ type=short
$ case $type in
tall)
echo "yay tall"
;;
short | petite)
echo "your height is most likely not that great"
;;
hid*)
echo "variable type starts with hid..."
;;
*)
echo "none of the cases matched :("
;;
esac
```

```
your height is most likely not that great
```

- the case statement stops the first time a pattern is matched
- the case `*)` is a catchall for whatever did not match.