Lecture 2: Advanced UNIX, shell scripts

Programming Tools And Environments
Objective

Objective: To introduce students to the concept of a shell, effective use of the shell, the shell as programming language, and shell scripts. We will use the bash shell for the examples and details provided.

- Shell syntax and commands
- history and command completion
- job control
- variables
- programming constructs
- scripts
What does the shell do?

- In Unix, separate from the OS (change look and feel)
- reads and executes commands
  - some handled by the shell itself (pwd, echo,…)
  - some are programs stored in some directory (look in directories in PATH). Start a subshell to execute these
- Provides support for better interaction with the computer/OS (command history, editing, configuration)
- Supports scripting (is a programming language)
Executing a Command

After reading a command, the shell may do some processing (see wildcards, etc in the syntax description that follows), then it must find a program to execute the command.

Some commands are executed directly by the shell. Other commands are executed by separate programs. These are found by looking in a list of directories for programs with the appropriate name. The shell searches directories in the PATH variable. A hash table is used to make the search fast. You can add new commands simply by adding new programs (a program can be any executable file including scripts – review Unix permissions) to directories in the PATH. You can modify/add directories in the PATH.
Finding out about Commands

- **type** – tells you if a command is a built-in, an alias, or a program (external to the shell)
- **which** – tells in which directory a utility is located
- **help** – displays information about built-in commands (it is a built-in itself)
- **info bash** – a good place to read about the BASH shell
Example

[jjohnson@ws44 cs265]$ which echo
/bin/echo

[jjohnson@ws44 cs265]$ type -a echo
echo is a shell builtin
echo is /bin/echo

[jjohnson@ws44 cs265]$ help echo  # help provides information on builtin commands
echo: echo [-neE] [arg ...]
    Output the ARGs. If -n is specified, the trailing newline is
    suppressed. If the -e option is given, interpretation of the
    following backslash-escaped characters is turned on:
    \a     alert (bell)
    \b     backspace
    \c     suppress trailing newline
    \E     escape character
    \f     form feed
    \n     new line
    \r     carriage return
    \t     horizontal tab
    \v     vertical tab
    \\    backslash
    \num    the character whose ASCII code is NUM (octal).

You can explicitly turn off the interpretation of the above characters
with the -E option.
Initialisation Files

- commands & variables read when shell is started
- if variables are placed in system-wide init files, available to every shell
- customizations must be exported to be available
- commands & aliases cannot be exported so must be placed in user-specific init
/etc/profile - System-Wide Initialization File

Read first by shell

Tasks accomplished:

- Sets and exports variables:
  - PATH, TERM, PS1, etc.
- Displays contents of /etc/motd (msg of the day)
- Sets default file creation permissions (umask)
If the shell is a login shell, it looks for one of the following (in order)

- ~/.bash_profile
- ~/.bash_login
- ~/.profile

If a non-login interactive shell, reads

- ~/.bashrc
Other customizations

- **PS1** – prompt
- **PATH** – add to the search path
- **Set shell options**
  - noclobber
  - ignoreeof
  - command-line editing option (vi or emacs)

**See** ~kschmidt/Public/ .bash_profile and .bashrc for examples

(Feel free to take them, as you wish)
Shell Variables

- The shell keeps track of a set of parameter names and values.
- Some of these parameters determine the behavior of the shell.
- We can access these variables:
  - set new values for some to customize the shell.
  - find out the value of some to help accomplish a task.
Example Shell Variables

sh / ksh / bash

PWD  current working directory
PATH list of places to look for commands
HOME  home directory of user
MAIL where your email is stored
TERM what kind of terminal you have
HISTFILE where your command history is saved
Displaying Shell Variables

- Prefix the name of a shell variable with "$" to dereference
- The `echo` command will do:
  ```
  echo $HOME
  echo $PATH
  ```
- You can use these variables on any command line:
  ```
  ls -al $HOME
  ```
Setting Shell Variables

You can change the value of a shell variable with an assignment command (this is a shell *builtin* command):

```bash
HOME=/etc
PATH=/usr/bin:/usr/etc:/sbin
NEWVAR="blah blah blah blah"
```

Note the lack of spaces around the '='
export

- export a variable
- make it available to subshells
  - value passed in
  - changes made to a value in the subshell do not persist in the caller

Subshell is created whenever a script or a program is run
  - Inherits parent shell's exported variables
The **PATH**

- Each time you give the shell a command line it does the following:
  - Checks to see if the command is a shell built-in.
  - If not - tries to find a program whose name (the filename) is the same as the command.

- The **PATH** variable tells the shell where to look for programs (non built-in commands).
echo $PATH

~ echo $PATH
/home/openwin/bin:/usr/bin:/bin:/usr/local/bin:/usr/sbin:/usr/bin/X11:/usr/games:/usr/local/packages/netscape

◆ The PATH is a list of ":" delimited directories.
◆ The PATH is a list and a search order.

◆ You can add stuff to your PATH by changing the shell startup file.
Notes about PATH

- If you do not have . in your PATH, commands in your current directory will not be found. You may or may not want to add . to your PATH. If you do not and want to execute a command in the current directory
  - ./command

- The PATH is searched sequentially, the first matching program is executed. Beware of naming your executables test as there is another program called test and this may be executed when you enter
  - test
PATH

[jjjohnson@ws44 software]$ echo $PATH
/usr/local/FrameMaker/bin:/home/jjjohnson/bin:/usr/local/gpl/mpich-1.2.4/bin:
   /usr/local/bin:/usr/sbin:/sbin:/usr/openwin/bin:/opt/SUNWspro/bin:/usr/ccs/bin:/usr/ucb:/usr/sbin:/usr/bin:/etc:/usr/etc:/usr/UTILS/publisher/bin:/usr/bin/X11:/usr/remote/alg_soft/linda2.5.2sol2.3/bin:

[jjjohnson@ws44 software]$ PATH=${PATH}:/home/jjjohnson/software

[jjjohnson@ws44 software]$ echo $PATH
/usr/local/FrameMaker/bin:/home/jjjohnson/bin:/usr/local/gpl/mpich-1.2.4/bin:
   /usr/local/bin:/usr/sbin:/sbin:/usr/openwin/bin:/opt/SUNWspro/bin:/usr/ccs/bin:/usr/ucb:/usr/sbin:/usr/bin:/etc:/usr/etc:/usr/UTILS/publisher/bin:/usr/bin/X11:/usr/remote/alg_soft/linda2.5.2sol2.3/bin::/home/jjjohnson/software
The `set` command with no parameters will print out a list of all the shell variables.

Some common options:

- `noclobber` – Keeps `mv`, `cp`, redirection from deleting an existing file
- `-o vi` (or `-o emacs`) – sets command-line editing mode
- `ignoreeof` – ctrl-D won't exit the shell
The `PS1` shell variable is your command line prompt.
The `PS2` shell variable is used as a prompt when the shell needs more input (in the middle of processing a command).
By changing PS1 and/or PS2 you can change the prompt.
Fancy bash prompts

Bash supports some fancy stuff in the prompt string:
\t is replace by the current time
\w is replaced by the current directory
\h is replaced by the hostname
\u is replaced by the username
\n is replaced by a newline
Example *bash* prompt

```
~ echo $PS1
======  [\h]  -  \t  ======
```

You can change your prompt by changing PS1:

```
PS1="Yes Master? 
```
Making Changes Stick

If you want to tell the shell (`bash`) to always use the prompt "Yes Master ?", you need to store the change in a `shell startup file`.

For `bash` - change the file `~/.bashrc`. 
SHELL

The SHELL variable holds the path of the login shell.
HOME

The HOME variable contains the PATH to your home directory.

When you use `cd` command with no arguments, the command uses the value of the HOME variable as the argument.

echo $HOME

/home/vzaychik
Metacharacters

Shell metacharacters are characters that are handled specially by the shell.

Below is an (incomplete) list of shell metacharacters

- >, >>, <, <<, |  Command redirection, pipe
- *, [], ?, {},  File specification
- ;, &, ||, &&, ()  Command Sequencing
Metacharacters (cont)

- "", '
  - Grouping Text
  - Double quotes and single quotes indicated that the enclosed text is to be treated as a unit, not as a group of words.

- #
  - Commenting
  - Rest of line after these characters is ignored

- \ (backslash) – the escape character
  - Indicates that the character immediately following the backslash is to be treated literally.
  - To remove a file named "#bogus" you can use the backslash
    
    `rm \\#bogus`
Aliases

Commands can be aliased (renamed) to alter their behavior. A common used of aliases is to add options to the default behavior of certain commands (e.g. ls, rm, mv, …)

It is common practice to alias rm to prompt the user to make sure that the specified files should be removed. This is especially important since you can remove all files with (see wildcards)

- rm *

To see current aliases, use the alias command, this can also be used to set new aliases.
Create an alias using the builtin command

```
alias name[=value]
```

- You should usually enclose `value` in quotes.
- no spaces around the '='

- Without a value, `alias` prints out the alias associated with `name`

```
alias dir="ls"
alias ls="ls -CF"
dir
```

`Output same as for "ls -CF"`
Variable Substitution

- Shells support the use of variables
- A variable is a name that is bound to a value.
- To set a variable (in Bash) just enter
  \[ \text{name=value} \]
  - No spaces around '='
- To see the settings of all variables just enter
  "set".
- To kill a variable:
  \[ \text{unset name} \]
Variable Substitution (cont)

- When processing a command, variable substitution occurs.
- A variable in a command is flagged by a dollar sign prefix "$\$

```bash
$ echo My shell is $SHELL
My shell is /bin/bash
```

- `echo` Writes out its arguments after substitution is performed.
Variable Substitution (cont)

Variable substitution will occur within double quotes

```
$ echo "My shell is $SHELL"
My shell is /usr/local/bin/tcsh
```

Substitution does not occur within single quotes.

```
$ echo 'My shell is $SHELL'
My shell is $SHELL
```
Variable Substitution (cont)

When the usage of a variable name is not clear, enclose it within braces \$\{name\}\$

```bash
$ prefix=cs265
$ suffix=.pdf
$ echo $prefix03$suffix
    .pdf
$ echo $\{prefix\}03$\{suffix\}
    cs26503.pdf
```

This occurs when constructing file names in script files.
Variable Substitution (cont)

Many programs use shell variables (also called environmental variables) to get configuration information.

Examples

- PRINTER is used by printing commands to determine the default printer.
- TERM is used by programs (e.g., vi, pine) to determine what type of terminal is being used.
- VISUAL is used by network news programs, etc., to determine what editor to use.
Command Substitution

Command substitution allows the output (stdout) of a command to replace the command name. There are two forms:

- $(command)
- `command`
Command-substitution - backquotes

If you surround a string with backquotes the string is replaced with the result of running the command in backquotes:

```
$ echo `ls`
foo fee file?

$ echo The date/time is `date`
The date/time is Tue Jan 25 00:32:04 EST 2000
```
Strong quoting – Single quotes

*Inhibits all* substitution, and the special meaning of metacharacters:

```bash
$ echo '$USER is $USER'
$USER is $USER

$ echo 'today is `date`'
today is `date`

$ echo 'No background&'
No background&

$ echo 'I said, "radio!"'
I said, "radio"
```
Weak quoting – double quotes

- Allows command and variable substitution
- Inhibits special meaning of all other metacharacters

```bash
$ echo "My name is $USER &"
My name is kschimidt &

$ echo "\$2.00 says `date`"
$2.00 says Sun Jan 15 01:43:32 EST 200
Command Execution

- Sometimes we need to combine several commands.
- There are four formats for combining commands into one line: sequenced, grouped, chained, and conditional.
- A sequence of commands can be entered on one line. Each command must be separated from its predecessor by semicolon.
- There is no direct relationship between the commands.

```none
command1; command2; command3
```
Grouped Commands

If we apply the same operation to the group, we can group commands.

Commands are grouped by placing them into parentheses.

Example:

```
echo "Month" > file; cal 10 2000 >> file
```

```
(echo "Month" ; cal 10 2000 ) > file
```
Conditional Commands

We can combine two or more commands using conditional relationships AND (&&) and OR (||).

If we AND two commands, the second is executed only if the first is successful.
If we OR two commands, the second is executed only if the first fails.

```
  cp file1 file2 && echo "Copy successful"
```

```
  cp file1 file2 || echo "Copy failed"
```
Shell Syntax

Comments
- # This is a comment
- ls # list the files in the current directory

Line continuation
- echo A long \n- > line

; #Command separator – you can list more than one command per line separated by ;
- ls ; who

/ #Pathname separator
- cd /home/jjohnson
Shell Syntax –
wildcards (globbing)

Wildcards, and pathname expansion
- * # match any string (including empty)
- ? # match any single character
- [set] # match characters listed in set (can be range)
- ![set] # match any character not given in set

Examples
- ls *.c
- ls *.
- ls *.[Hh][Tt][Ll]
- ls [a-z]
Shell Syntax – redirection and pipes

File redirection and pipes

- `<`  # redirect input from specified source
- `>`  # redirect output to specified source
- `>>`  # redirect output and append to specified source
- `|`  # pipe the output from one command to the input to the next

Examples

- `grep word < /usr/dict/words`
- `ls > listing`
- `ls >> listing`
- `ls -l| wc -l`
Shell Syntax - stderr

- Note file redirection of standard output [stdout] does not include error messages, which go to standard error [stderr] (when you are in a terminal session, both stdout and stderr go to the screen; however, when you redirect stdout to a file, stderr still goes to the screen).

- stdout is designated by the file descriptor 1 and stderr by 2 (standard input is 0)

  - To redirect standard error use 2>
  - ls filenothere > listing 2> error
  - ls filenothere 2>&1 > listing # both stdout and stderr redirected to listing
Shell Syntax – bg jobs

Background jobs

- `&` # run command in the background
- `grep ‘we.*’< /usr/word/dict > wewords &`
- This runs the grep command in the background – you immediately get a new prompt and can continue your work while the command is run. Note that the output was redirected otherwise you would lose it.
- To find out the jobs that are running in the back use the command `jobs` (also see the command `ps`)
- To stop a job use the `kill` command (you need to know the process id of the job [use –l option to jobs or ps]
What Is A Script?

A script is a small program that is executed by the shell.

The script is a text file which will contain:
- Shell commands you normally use.
- Shell flow control constructs (e.g., if-then-else, etc.)
- A heavier use of variables than you normally would use from the command line.
Why Write Scripts?

- The Unix shells are very powerful, but there are some things that they do not do well.
  - For example, renaming all files of the form `cs265l*.ppt` to `cs265l*n.ppt` requires a `mv` command for each file.

- Sequences of operations which you perform often can be placed into a script file and then executed like a single command.

- Any task you do (by hand) more than twice should probably be wrapped into a script.
Shell scripting - Why Not?

- resource-intensive tasks, especially where speed is a factor
- complex applications, where structured programming is a necessity
- mission-critical applications upon which you are betting the ranch, or the future of the company
- situations where security is important, where you need to protect against hacking
Why Not? (cont.)

Project consists of subcomponents with interlocking dependencies.

Extensive file operations required (Bash is limited to serial file access, and that only in a particularly clumsy and inefficient line-by-line fashion).

Need to generate or manipulate graphics or GUIs.
Why Not? (cont.)

Need direct access to system hardware
Need port or socket I/O
Need to use libraries or interface with legacy code
Very Simple Script

#!/bin/sh

echo Hello World
Example: Simple Script

Let’s create a shell script to give us information about the system.

We create the script using a text editor.

- Let’s call the file “status”

```bash
#!/bin/bash
uptime
users
```

Be sure to insert an “enter” after the last line.

- Exit the editor
Running a Script

To execute the shell we can do

```
$ bash status
10:37 up 23 days, 23:54, 14 users, load average ...
afjhj billc ...
```

We can also execute the file as a command if the appropriate execute access is granted.

```
$ status
bash: ./status: Permission denied
$ chmod +x status
$ status # Works correctly.
```
#! – "sha-bang"

- Not needed if bash kicks off the script, but...
- Shells look for "#!" at the very beginning of the file
- It is followed by the program (interpreter) that will read this file:
  - #!/bin/bash
  - #!/usr/bin/Python
    # this is a Python program. Bash doesn't read it. Gives it right to the Python interpreter
Conditional Expressions - `test`

- To perform *ifs* and *whiles* we need to be able to construct conditional expressions.
  - A conditional expression is one that evaluates to true or false depending on its operands (e.g., `x<y`).
- For Bash, this involves using the `test` command.
  
  ```bash
  test expression
  ```
  
  Or

  ```bash
  [ expression ]
  ```

  *Separate expression from brackets spaces*
Conditional Expressions (cont)

- *test* returns an exit status of zero (success) if the expression evaluates to true.

- *test* uses a variety of operators
  - Unary file operators can test various file properties
    - "op file"
      - `-e` True if file exists
      - `-w` True if file exists and is writable
      - `-d` True if file is a directory
      - `-O` True if I own the file
  - E.g.
    ```bash
    if [ -e ~kschmidt/public_html ]; then
echo "Kurt has a public web directory"
fi
    ```
[] – file and string operators

- **Binary file operators** “*file1 op file2*”
  - `-nt` True is *file1* is newer than *file2*
  - `-ot` True is *file1* is older than *file2*

- **Unary string operators** “*op string*”
  - `-z` True if string is of zero length
  - `-n` True if string is not of zero length
  - `-l` *Returns length of string*

- **E.g.**

  ```bash
  if [ -z "$myVar" ] ; then
      echo "\$myVar has null length"
  fi
  ```
Operators are case sensitive:

- ==   !=  <   >   <=  >=

E.g.

```bash
if [ "$myVar" == "duh" ] ; then
    echo "I don't have a clue" ; fi
fi
if [ "abc" < "ABC" ] ; then
    echo ""abc" comes before "ABC""" ; fi
fi
```
[] – arithmetic operators

- Binary operators:
  - `lt`  `gt`  `le`  `ge`  `eq`  `ne`
- E.g.

```bash
if [ 2 -le 3 ] ; then ; echo "cool!" ; fi
x=5
if [ "$x" -ne 12 ] ; then
    echo "Still cool" ; fi
```
Logical Operators

Logical expression tools

- ! *expression* Logical not (i.e., changes sense of expression)
- e1 -o e2 True if e1 or e2 is true.
- e1 -a e2 True if both expressions are true.
- ( *expression*) Works like normal parentheses for expressions; use spaces around the expression.

Examples:

```
test -e bin -a -d /bin is true
[ -e ~/.bashrc -a ! -d ~/.bashrc ] is true.
```
Bash added `[][]` for more C-like usage:

```bash
if [[ -e ~/.bashrc && ! -d ~/.bashrc ]]
then
    echo "Let's parse that puppy"
fi

if [[ -z "$myFile" || ! -r $myFile ]]
...

Why sometimes quote $myFile, sometimes not (it's usually a good idea to do so)?
Arithmetic Expressions

- Bash usually treats variables as strings.
- You can change that by using the arithmetic expansion syntax: `(( arithmeticExpr ))`
- `(( ))` shorthand for the `let` builtin statement

```bash
$ x=1
$ x=x+1  # "x+1" is just a string
echo $x
x+1
```
Arithmetic Expression (cont)

```bash
$ x=1
$ x=${x}+1 # still just a string
$ echo $x
1+1

❖ Closer, but still not right.

$ x=1
$ (( x=x+1 ))
$ echo $x
2

❖ Finally!
```
Sample: Basic conditional example if .. then

#!/bin/bash
if [ "$1" = "foo" ] ; then
    echo expression evaluated as true
fi
Sample: Basic conditional example if .. then ... else

#!/bin/bash
if [ "$1" = "foo" ]
then
    echo 'First argument is "foo"
else
    echo 'First arg is not "foo"
fi
Sample: Conditionals with variables

#!/bin/bash
T1="foo"
T2="bar"
if [ "$T1" = "$T2" ] ; then
echo expression evaluated as true
else
echo expression evaluated as false
fi
Checking return value of a command

```bash
if diff "${fileA}" "${fileB}" > /dev/null then
   echo "Files are identical"
else
   echo "Files are different"
fi
```
Case Statement

case $opt in
  a  ) echo "option a";;
  b  ) echo "option b";;
  c  ) echo "option c";;
  \? ) echo \\
       "usage: alice [-a] [-b] [-c] args..."
       exit 1;;
esac
Special Variables

- `$#`  the number of arguments
- `$*`  all arguments
- `$@`  all arguments (quoted individually)
- `$?`  return value of last command executed
- `$$`  process id of shell
- `$HOME, $IFS, $PATH, $PS1, $PS2`
Scripts and Arguments

- Scripts can be started with parameters, just like commands
  
  ```
aScript arg1 arg2 ...
  ```

- The scripts can access these arguments through shell variables:
  - "$n" Is the value of the n\textsuperscript{th} parameter.
    - The command is parameter zero
  - "#$" Is the number of parameters entered.
  - "$*" Expands as a list of all the parameters entered except the command.
Let’s quickly write a script to see this:

- (this first line is a quick and dirty way to write a file)

```
$ cat > xx # cat reads from stdin
      if no file specified
echo $0
echo $#
echo $1 $2
echo $*
C-d # Control-D is the end of file character.
$ chmod +x xx
```

- The file xx is now an executable shell script.
Unspecified parameters expand as empty strings (i.e., as nothing)
Loops for, while and until

The for loop is a little bit different from other programming languages. Basically, it lets you iterate over a series of 'words' within a string.

The while executes a piece of code if the control expression is true, and only stops when it is false (or an explicit break is found within the executed code).

The until loop is almost equivalent to the while loop, except that the code is executed while the control expression evaluates to false.
For samples

**Example**

```bash
$ for x in 1 2 3;
  do
  > echo $x
  > done
```

```bash
$ for x in *
  do
  > echo $x
  > done
```

1
2
3

bin
mail
public_html
...

**Example**

```bash
$ for x in 1 2 3;
  do
  > echo $x
  > done
```
For samples

#!/bin/bash
for i in $(cat list.txt) ; do
echo item: $i
done

#!/bin/bash
for (( i=0; i<10; ++i )) ; do
echo item: $i
done
While sample

COUNTER=0
while [ $COUNTER -lt 10 ] ; do
  echo The counter is $COUNTER
  let COUNTER=COUNTER+1
done

COUNTER=0
while (( COUNTER < 10 )) ; do
  echo The counter is $COUNTER
  (( COUNTER = COUNTER+1 ))
done
#!/bin/bash

COUNTER=20

until [ $COUNTER -lt 10 ]
do
    echo COUNTER $COUNTER
    let COUNTER--=1
done
Example: Renaming

- **Goal:** Rename all the files of the form cs265l*.ppt to cs265l*.exe where * should be 02, 03 … 10.

- **Solution:** *for loop*
  - for n in 02 03 04 05 06 07 08 09 10; do
    mv cs265l$n.ppt cs265l$n.exe
  done

  - Moves cs265l02.ppt to cs265l02.exe, cs265l03.ppt to cs265l03.exe, etc.
Loop Control

- `break` terminates the loop
- `continue` causes a jump to the next iteration of the loop
Debugging Tip

If you want to watch the commands actually being executed in a script file, insert the line “set -x” in the script.

```
set -x
for n in *; do
echo $n
done
```

- Will display the expanded command before executing it.

```
+ echo bin
bin
+ echo mail
mail
```
Functions

As in almost any programming language, you can use functions to group pieces of code in a more logical way or practice the divine art of recursion.

Declaring a function is just a matter of writing function
my_func { my_code }.

Calling a function is just like calling another program, you just write its name.
Local variables

#!/bin/bash
HELLO=Hello
function hello {
    local HELLO=World
    echo $HELLO
}

$ echo $HELLO
$ hello
$ echo $HELLO

$ echo $HELLO
$ hello
$ echo $HELLO
#!/bin/bash

function quit {
    echo 'Goodbye!'
    exit
}

function hello {
    echo "Hello $1"
}

for name in Vera Kurt ; do
    hello $name
done

quit
Parameter Expansion

- \texttt{${\text{parameter}}:-\text{word}$}
  - Use Default Values.
- \texttt{${\text{parameter}}:=\text{word}$}
  - Assign Default Values.
- \texttt{${\text{parameter}}:?\text{word}$}
  - Display Error if Null or Unset.
- \texttt{${\text{parameter}}:+\text{word}$}
  - Use Alternate Value.
More Parameter Expansion

- We can remove parts of a value:
  - `${param#pattern}`
  - `${param##pattern}`
    - Removes shortest (#) or longest (##) leading pattern, if there's a match
  - `${param#pattern}`
  - `${param##pattern}`
    - Removes shortest(%) or longest (%) trailing pattern, if match

- *pattern* is expanded just as in pathname expansion (globbing) - *, ?, []
Example – Parameter Expansion

```bash
$ foo=j.i.c
$ echo ${foo#*.}
i.c
$ echo ${foo##*.}
c
$ echo ${foo%.*}
j.i
$ echo ${foo%%.*}
j
```
#!/bin/bash

if [ -z "$1" ]; then
    echo usage: $0 directory
    exit
fi

SRCD=$1

TGTD="/tmp/"

OF=home-$ (date +%Y%m%d).tgz

tar -cZf $TGTD$OF $SRCD