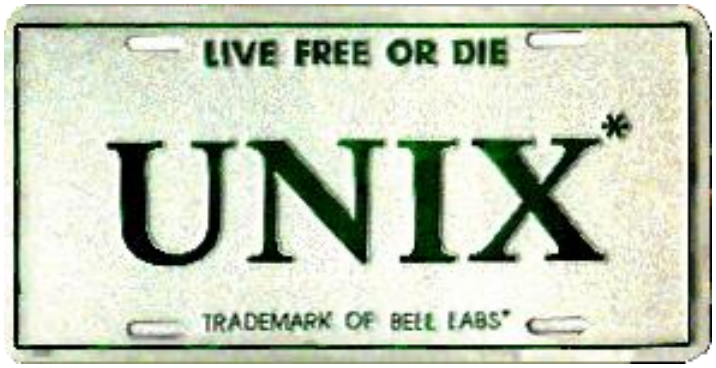


Unix Shells

Slides adapted from Prof. Andrzej (AJ) Bieszczad

Programming or Scripting ?

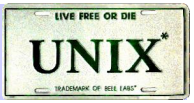
- Shell scripting allows us to automate a lot of tasks that would otherwise require a lot of commands.

  – Programming languages:
    - more powerful and a lot faster than scripting languages
    - compiled into an executable
    - executable is not easily ported into different operating systems

  – Scripting languages:
    - generally slower than compiled programs
    - interpreter reads the instructions and execute
    - source files easily portable to any operating system

The first bash script

```
$ cat > hello.bash
#!/bin/bash
echo “Hello World”
^D

$ chmod 700 hello.bash

$ hello.bash
Hello World
```



The second bash script

- A program that copies all files into a directory, and then deletes the directory along with its contents:

```
$ mkdir ../trash
$ cp –r * ../trash
$ rm -r ../trash
$ echo “Deleted all files!”
```

- Instead of having to type all that interactively on the shell, write a shell program instead:

```
$ cat > trash.bash
#!/bin/bash
mkdir ../trash
cp –r * ../trash
rm –r ../trash
echo “Deleted all files!”
^D
```

- **SHELL PROGRAMS: SCRIPTS**

- The system decides which shell the script is written for by examining the first line of the script.

- Here are the rules that it uses to make this decision:

1) If the first line of the script is just a pound sign(#),
then the script is interpreted by the shell from which you executed this script as a command.

2) If the first line of the script is of the form #! path name,
then the executable program *pathName* is used to interpret the script.

3) If neither rule1 nor rule2 applies,
then the script is interpreted by a Bourne shell (sh).

- **SHELL PROGRAMS: SCRIPTS**

- Here is an example that illustrates the construction and execution of two scripts, one for the Bash shell and the other for the Korn shell.

```
$ cat  >  script1      ---> create  the bash script.
#!/bin/bash
# This is a sample bash script.
echo  -n  the date today is   # in bash,  -n omits new line
date  # output today's date.
^D                          ---> end of input.

$ cat  > script2       ---> create the Korn-shell script.
#!/bin/ksh
#This is a sample Korn shell script.
echo  "the date today is \c"  # in ksh, \c omits the new line
date     # output  today's date.
^D                          ---> end of input
```

- **SHELL PROGRAMS: SCRIPTS**

```
$ chmod +x  script1  script2   ---> make the scripts executable.

$ ls  -lF   script1    script2                ---> look at the attributes of the
                                                  ---> scripts.
-rwxr-xr-x   1   glass      138  Feb  1  19:46  script1*
-rwxr-xr-x   1   glass      142  Feb  1  19:47  script2*

$ script1                          ---> execute the Bash shell script.
The date today is Wed, Aug 24, 2016  7:38:55 AM

$ script2                          ---> execute the Korn-shell script.
The date today is Wed, Aug 24, 2016  7:38:55 AM
$ _
```

- **SUBSHELLS**

1) When a grouped command such as (ls; pwd; date) is executed

If the command is not executed in the background, the parent shell sleeps until the child shell terminates.

2) When a script is executed

If the script is not executed in the background, the parent shell sleeps until the child shell terminates.

3) When a background job is executed

The parent shell continues to run concurrently with the child shell.

UNIX Shells

- A child shell is called a *subshell*.

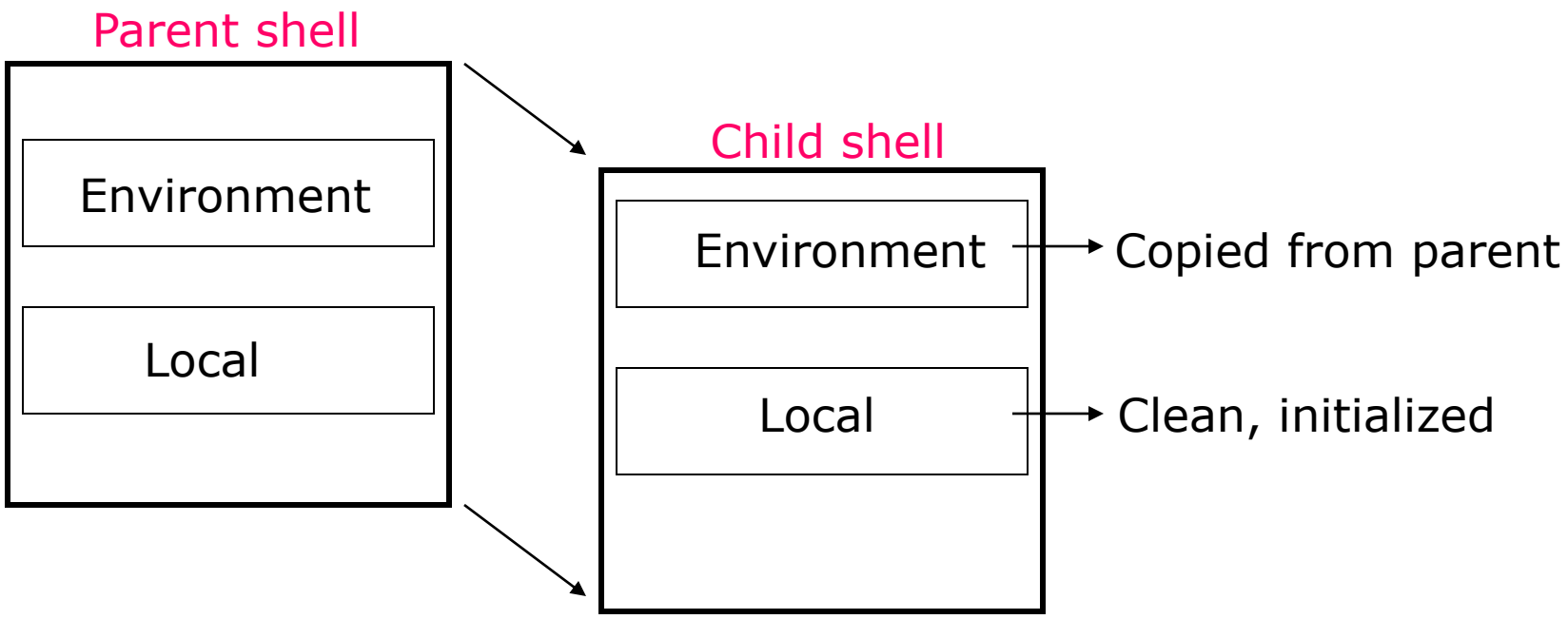
- *cd* commands executed in a subshell do not affect the working directory of the parent shell:

```
$ pwd              ---> display my login shell's current directory.
/home/glass

$ ( cd /; pwd ) ---> the subshell moves and executes pwd.
/                  ---> output comes from the subshell.

$ pwd              ---> my login shell never moved.
/home/glass
$ -
```

UNIX Shells

- Every shell contains two data areas:
  an environment space and a local-variable space.

A child shell inherits a copy of its parent's environment space and a clean local-variable space:



Environment variables are therefore used for transmitting useful information between parent shells and their children.

- **VARIABLES**

- Here is a list of the predefined environment variables that are common to all shells:

Name	Meaning
$HOME	the full pathname of your home directory
$PATH	a list of directories to search for commands
$MAIL	the full pathname of your mailbox
$USER	your username
$SHELL	the full pathname of your login shell
$TERM	the type of your terminal

- **VARIABLES**

- the syntax for assigning a variable is as follows:

variableName=value ---> place no spaces around the value
or
variableName=“ value ” ---> here, spacing doesn’t matter.

```
$ echo $HOME
/home/SRD

$ HOME=UG1

$ echo $HOME
UG1
$
```

- **VARIABLES**

- The next example illustrates the difference between local and environment variables.

  In the following, we assign values to two local variables and then make one of them an environment variable by using the Bourne shell *export* command.

  Note that the value of the environment variable is copied into the child shell, but the value of the local variable is not.

  Finally, we press ***Control-D*** to terminate the child shell and restart the parent shell, and then display the original variables:

UNIX Shells

```
$ firstname=Shiv Ram                    ---> set a local variable.
$ lastname=Dubey                        ---> set another local variable.
$ echo $firstname  $lastname            ---> display their values.
Shiv Ram Dubey

$ export lastname                       ---> make "lastname" an
                                        ---> environment variable.
$ sh                    ---> start a child shell; the parent sleeps.

$ echo  $firstname   $lastname          ---> display values again.
Dubey                   ---> note that firstname was't copied.

$ ^D
$ echo  $firstname  $lastname           ---> they remain unchanged.
Shiv Ram Dubey
$ _
```

Warning !

- The shell programming language does not type-cast its variables. This means that a variable can hold number data or character data.

```
count=0
count=Sunday
```

- Switching the TYPE of a variable can lead to confusion for the writer of the script or someone trying to modify it.

- So it is recommended to use a variable for only a single TYPE of data in a script.

UNIX Shells

- several common built-in variables that have special meanings:

Name	Meaning
$$	The process ID of the shell.
$#	The number of parameters passed.
$0	The name of the shell script(if applicable).
$1..$9	$n refers to the nth command line argument (if applicable).
$*	A list of all the command-line arguments.
$@	Array of words containing all the parameters passed to the script

Example:

```
$ cat  script3        ---> list the script.
 echo the name of this script is $0
 echo the first argument is $1
 echo a list of all the arguments is $*
 echo this script places the date into a temporary file called $1.$$
 date > $1.$$    # redirect the output of date.
 ls   $1.$$      # list the file.

$ script3  paul  ringo  george  john   ---> execute the script.
 the name of this script is script3
 the first argument is paul
 a list of all the arguments is paul ringo george john
 this script places the date into a temporary file called paul.24321
 paul.24321
 $ _
```

- **QUOTING**

- There are often times when you want to inhibit the shell's wildcard-replacement, variable-substitution, and/or command-substitution mechanisms.

The shell's quoting system allows you to do just that.

- Here's the way that it works:

1) Single quotes(') inhibit wildcard replacement, variable substitution, and command substitution.

2) Double quotes(") inhibit wildcard replacement only.

3) When quotes are nested, it's only the outer quotes that have any effect.

- **QUOTING**

-Examples:

```
$ echo  3 * 4 = 12    ---> remember, * is a wildcard.
  3   a.c    b    b.c    c.c    4 = 12
$ echo  "3 * 4 = 12" ---> double quotes inhibit wildcards.
  3 * 4 = 12
$ echo  '3 * 4 = 12'  ---> single quotes inhibit wildcards.
  3 * 4 = 12

$ name=Graham
$ echo  'my name is $name - date is `date`'
  my name is $name - date is 'date'

$ echo  "my name is $name - date is `date`"
  my name is Graham - date is Wed, Aug 24, 2016  7:38:55 AM
$ -
```

Command Substitution

- The backquote “`” is different from the single quote “´”. It is used for command substitution:

```
$ LIST=`ls`
$ echo $LIST
hello.bash read.bash

$ PS1=“`pwd`---->”
/home/SRD---->
```

- We can perform the command substitution by means of $(command)

```
$ LIST=$(ls)
$ echo $LIST
hello.bash read.bash

$ BCKUP=/home/`whoami`/backup-$(date +%d-%m-%y)
$ echo $BCKUP
/home/SRD/backup-30-08-16
```

- **JOB CONTROL**

- Convenient multitasking is one of UNIX's best features, so it's important to be able to obtain a ***listing of the current processes*** and to ***control their behavior***.

1) **ps**, which generates a list of processes and their attributes, including their names, process ID numbers, controlling terminals and owner.

2) **kill**, which allows to terminate a process based on its ID number.

3) **wait**, which allows a shell to wait for one of its child processes to terminate.

Utility : ps -ef

ps generates a listing of process-status information.

By default, the output is limited to processes created by your current shell.

The -e option instructs ps to include all running processes.
The -f option causes ps to generate a full listing.

Utility : sleep seconds

The sleep utility sleeps for the specified number of seconds and then terminates.

UNIX Shells

```
$ ( sleep 10; echo done ) &     ---> delayed echo in background.
27387                           ---> the process ID number.
$ ps
PID    TTY    TIME    CMD
27355  pts/3  0:00  bash          ---> the long shell.
27387  pts/3  0:00  bash          ---> the subshell.
27388  pts/3  0:00  sleep 10      ---> the sleep.
27389  pts/3  0:00  ps            ---> the ps command itself!
$ done                          ---> the output from the background process.
```

The meaning of the common column headings of ps output:

Column	Meaning
UID	the effective user ID of the process
PID	the ID of the process
PPID	the ID of the parent process
TTY	the controlling terminal
CMD	the name of the command
STIME	the time the process was created, or the date, if the process was created before the current day
S (STAT)	the state of the process

- **Process Status: ps**

- The S field encodes the stat of the process as follows:

letter	Meaning
O	running on a processor
R	runable
S	sleeping
T	suspended
Z	zombie process

- **Process Status: ps**

- Here's an example of some user-oriented output from ps:

```
$ ( sleep 10; echo done ) &
27462
$ ps -f                    ---> request user-oriented output.
     UID   PID   PPID  C     STIME     TTY     TIME    CMD
   glass   731   728   0  21:48:46  pts/5     0:01    -ksh
   glass   831   830   1  22:27:06  pts/5     0:00    sleep 10
   glass   830   731   0  22:27:06  pts/5     0:00    -ksh
$ done                     ---> output from previous command
```