Shell Programs (Scrips)

- Shell commands can be listed in text file to be executed
  - Requires changing permissions to *execute*
    - how does one do that?
  - To execute just type the file name, e.g., *./my-script*
  - Which shell is the script written for? How does this affect the execution?

Shell Programs (Scrips)

- System decides which shell the script is written for by examining the first line of the script
  - if first line is just #
    - use the calling shell to execute the script
  - if first line is of form #! *pathName*
    - use executable program specified by *pathName* to execute the script
  - if neither of the above rules applie
    - use Bash

Shell Programs (Scrips)

- Example

```
$ cat > script.csh              ...create the C shell script.

#!/bin/csh
# This is a sample C shell script.
echo -n the date today is       # in csh, -n omits newline
date                            # output today’s date
^D                              ...end-of-input.

$ cat > script.ksh              ...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 nl
date                            # output today's date.
^D                          ...end-of-input.
```

Shell Programs (Scrips)

- output

```
$ chmod +x script.csh script.ksh    ...make them executable.
$ ls -lFG script.csh script.ksh     ...look at attributes.
-rwxr-xr-x  1 glass       138 Feb  1 19:46 script.csh*
-rwxr-xr-x  1 glass       142 Feb  1 19:47 script.ksh*
$ ./script.csh                   ...execute the C shell script.
the date today is Tue Feb  1 19:50:00 CST 2005
$ ./script.ksh                   ...execute the Korn shell script.
the date today is Tue Feb  1 19:50:05 CST 2005
$ _
```

Subshells or Child Shells

- What shell is executing?
  - when you log into system an initial login shell is executed
    - which executes any simple command
    - however sometimes the shell creates a new shell (child process) to perform tasks

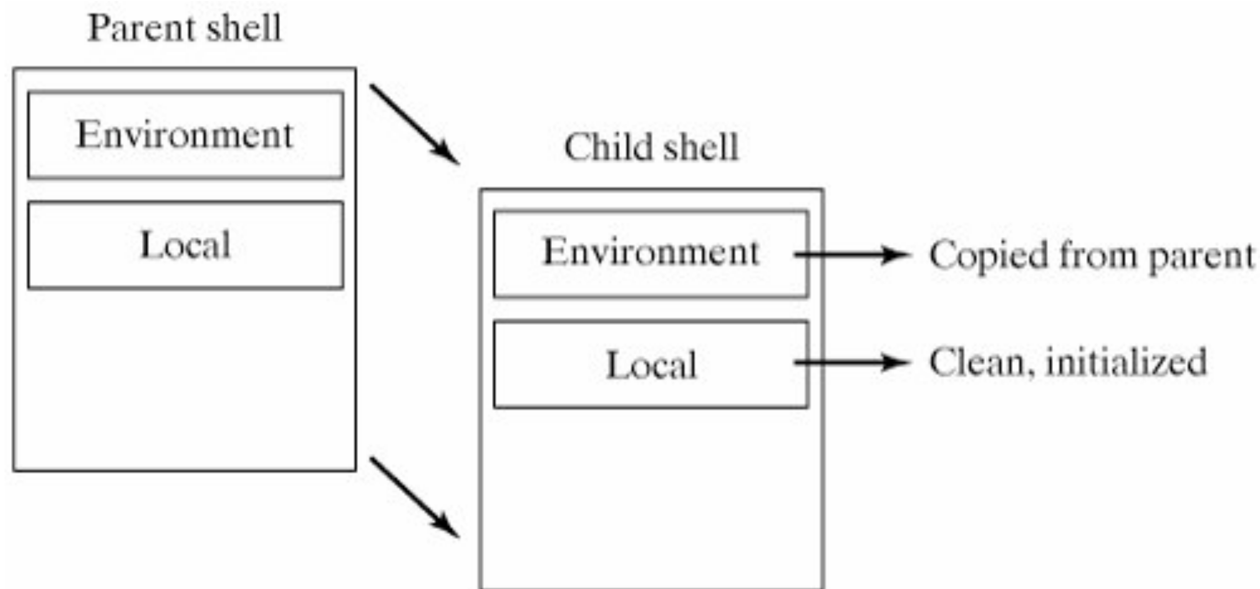
Shell Programs (Scrips)

- Child shells are created when:
  - grouped commands are executed, e.g., *ls; pwd; date*
  - a script is executed. If the script is not executed in background, the parent shell sleeps until child shell terminates
  - when background job is executed, parent shell creates child shell to execute this background command.
    - parent and child shells run concurrently

Shell Programs (Scrips)

- shell contains two data areas:

    1. environment space
    2. local variable space



source: fig 5-10 of textbook

Example

```
$ pwd                  ...display my login shell's current dir.

/home/glass
$ (cd /; pwd)          ...the subshell moves and executes pwd.
/                      ...output comes from the subshell.
$ pwd                  ...my login shell never moved.
/home/glass
$ _
```

Variables

- Two kinds of variables:
  - Local variables
  - Environmental variables
- Both variables hold data in a string format
- What is the difference”
  - Child shell inherits environment variables from parent,
  - but not its data variables

Environment variables

- Predefined environment variables, common to all shells
  - how does on display them?

Figure 5-11. Predefined shell variables.

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.

Environment variables

```
-bash-3.2$ echo HOME = $HOME, PATH = $PATH
HOME = /home/krings, PATH = /usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin
-bash-3.2$
-bash-3.2$ echo MAIL = $MAIL
MAIL = /var/spool/mail/krings
-bash-3.2$
-bash-3.2$ echo USER = $USER, SHELL = $SHELL, TERM=$TERM
USER = krings, SHELL = /bin/bash, TERM=xterm
-bash-3.2$
```

Variables

- Declare local variables, e.g.,

  -bash-3.2$ firstname=Carl

  -bash-3.2$ lastname=Strammsack

  -bash-3.2$ echo $firstname $lastname

  Carl Strammsack

  -bash-3.2$

Variables

- Now export *lastname* to make it an environment variable

  -bash-3.2$ export lastname

  -bash-3.2$ sh

  sh-3.2$ echo $firstname $lastname

  Strammsack

  sh-3.2$ exit

  -bash-3.2$ echo $firstname $lastname

  Carl Strammsack

Variables

- Special built-in shell variables

Figure 5-12. Special built-in shell variables.

Name	Meaning
$$	The process ID of the shell.
$0	The name of the shell script (if applicable).
$1..$9	$*n* refers to the *n*th command-line argument (if applicable).
$*	A list of all the command-line arguments.

Variables

- examples of using common special variables

```
$ cat script.sh                          ...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
echo called $1.$$
date > $1.$$      # redirect the output of date.
ls $1.$$          # list the file.
rm $1.$$          # remove the file.
$ ./script.sh paul ringo george john       ...execute it.
the name of this script is script.sh
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

- Quoting and wildcard-replacement
  - Single quotes (') inhibit wildcard replacement, variable substitution, and command substitution.
  - Double quotes (") inhibit wildcard replacement only.
  - When quotes are nested, only the outer quotes have any effect.

Quoting

- Quoting and wildcard-replacement

  -bash-3.2$ echo my name is $lastname: date is `date`

  my name is Strammsack: date is Wed Sep 22 10:41:11 PDT 2010

  -bash-3.2$ echo 'my name is $lastname: date is `date` '

  may name is $lastname: date is `date`

  -bash-3.2$ echo "my name is $lastname: date is `date` "

  my name is Strammsack: date is Wed Sep 22 10:43:35 PDT 2010

Script example: *here.sh*

- consider the following script:

```
-bash-3.2$ cat here.sh
#! /bin/sh

mail $1 << ENDOFTEXT
Dear $1,
 Please see me regarding some exciting news!
- $USER
ENDOFTEXT
echo mail sent to $1
-bash-3.2$
```

Script example: *here.sh*

- execute the script:

  -bash-3.2$ ./here.sh krings
  mail sent to krings
  -bash-3.2$

Script example: *here.sh*

- Verify what the script did

```
-bash-3.2$ mail
Mail version 8.1 6/6/93.  Type ? for help.
"/var/spool/mail/krings": 1 message 1 new
>N  1 krings@eternium.cs.u  Wed Sep 22 10:48  15/549
&
Message 1:
From krings@eternium.cs.uidaho.edu  Wed Sep 22 10:48:36 2010
X-Original-To: krings
Delivered-To: krings@eternium.cs.uidaho.edu
To: krings@eternium.cs.uidaho.edu
Date: Wed, 22 Sep 2010 10:48:36 -0700 (PDT)
From: krings@eternium.cs.uidaho.edu (Axel Krings)

Dear krings,
 Please see me regarding some exciting news!
- krings
```

Job control

- Processes and control
  - ps
    - generates list of processes and their attributes, including their name, process ID number, controlling terminal, and owner
  - kill
    - allows you to terminate a process based on its ID number
  - wait
    - allows a shell to wait for one of its child processes to terminate

Process Status: ps

Figure 5-13. Description of the ps command.

Utility: **ps** -efl

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. The **-l** option generates a long listing. The meaning of each **ps** column is described in the text that follows.

Process Status: ps

- ps output column meaning (source fig 5-15 of text)

Column	Meaning
S	The process state.
UID	The effective user ID of the process.
PID	The process ID.
PPID	The parent process ID.
C	The percentage of CPU time that the process used in the last minute.
PRI	The process priority.
SZ	The size of the process's data and stack in kilobytes.
STIME	The time the process was created, or the date if created before today.
TTY	The controlling terminal.
TIME	The amount of CPU time used so far (MM:SS).
CMD	The name of the command.

Process Status: ps

- ps process state codes (source: textbook)

Figure 5-16. Process state codes reported by ps.

Letter	Meaning
O	Running on a processor.
R	Runable.
S	Sleeping.
T	Suspended.
Z	Zombie process.

Signaling Processes: kill

- from man page:
  - The command kill sends the specified signal to the specified process or process group.
  - If no signal is specified, the TERM signal is sent. The TERM signal will kill processes which do not catch this signal.
  - For other processes, it may be necessary to use the KILL (9) signal, since this signal cannot be caught.
  - Most modern shells have a builtin kill function, with a usage rather similar to that of the command described here.

Signaling Processes: kill

- Utility/Shell Command:
  - *kill [-signalId] {pid }+*
  - default is -15 = TERM signal
  - for list of legal signal names use *kill -l*
  - *kill -9* unconditional kill

Wait for child: wait

- Shell Command: wait [pid]
  - wait causes the shell to suspend until the child process with the specified process ID number terminates.
  - If no arguments are supplied, the shell waits for all of its child processes.

The $PATH

- PATH indicates the default search paths the system indicates, separated by “:”
  - *echo $PATH*
  - my output is: */usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin*
- Why is the current directory “.” not included?
  - e.g., *a.out* versus *./a.out* ????