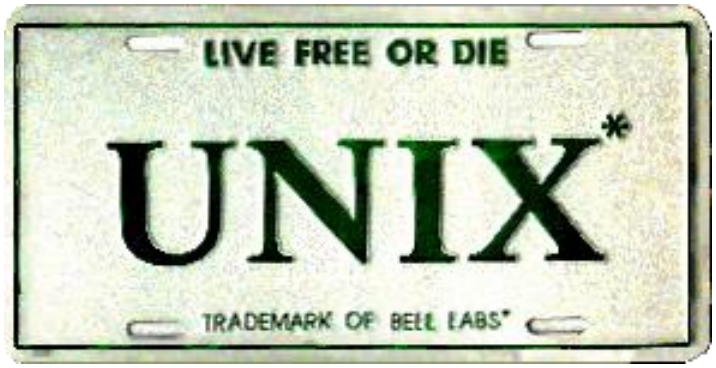

Unix Shells

Slides adapted from Prof. Andrzej (AJ) Bieszczad

Utility : nohup command

- **The Bash, Bourne and Korn shells** automatically terminate background processes when you log out, whereas the C shell allows them to continue.

- The nohup utility make a background process immune to this effect in Bash, Bourne or Korn shell.

- This utility is ideal for ensuring that background processes are not terminated when your login shell is exited.

- The standard output and error channels of command are automatically redirected to a file called "nohup.out".

UNIX Shells

- Executing a command using **nohup**
  Logout, and then log back in again
  The command is not visible in the output of a regular ps.

  To include a list of all of the current processes without control terminals in a **ps** output, use the -x option.

UNIX Shells

- Here's an example of this effect:

```
$ nohup  sleep  10000 &          ---> nohup a background process.
27406
Sending output to 'nohup.out'  ---> message from "nohup".
$ ps                            ---> look at processes.
PID   TT   STAT   TIME   COMMAND
27399  p3  S      0:00    -sh(sh)
27406  p3  S  N   0:00   sleep 10000
27407  p3  R      0:00   ps
$ ^D                            ---> logout.
```

UNIX Shells

```
Login: UG1                          ---> log back in.
Password:                           ---> secret.

$ ps                                ---> the background process is not
                                    ---> listed.
PID    TT   STAT   TIME   COMMAND
27409  p3   S      0:00   -sh(sh)
27411  p3   R      0:00   ps

$ ps   -x        ---> the background process is listed.

PID     TT      STAT    TIME    COMMAND
27406   ?         IN     0:00    sleep  10000
27409   p3        S      0:00    -sh ( sh )
27412   p3        R      0:00    ps  -x
$ _
```

- **Signaling Processes: kill**

- ***kill*** command terminates a process before it completes.

- Bash, Korn and C shells contain Kill as built-in command, whereas Bourne shell invoke the standard utility instead.

- Both versions of kill supports the following functionality:

Utility/Shell Command : kill [-signalId] {pid}
kill -l

- kill sends the signal with code signalId to the list of processes.

- signalId may be the number or name of a signal.

UNIX Shells


```
$ kill  -l          ---> list the signal names.
1) SIGHUP       2) SIGINT       3) SIGQUIT
4) SIGILL       5) SIGTRAP      6) SIGABRT
7) SIGEMT       8) SIGFPE       9) SIGKILL
10) SIGBUS      11) SIGSEGV     12) SIGSYS
13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGUSR1     17) SIGUSR2     18) SIGCHLD
19) SIGPWR      20) SIGVTALRM   21) SIGPROF
22) SIGIO       23) SIGWINCH    24) SIGSTOP
25) SIGTSTP     26) SIGCONT     27) SIGTTIN
28) SIGTTOU     29) SIGURG      30) SIGLOST
32) SIGDIL      33) SIGXCPU     34) SIGXFSZ
35) SIGCANCEL   37) SIGRTMIN    38) SIGRTMIN+1
39) SIGRTMIN+2  40) SIGRTMIN+3  41) SIGRTMAX-3
42) SIGRTMAX-2  43) SIGRTMAX-1  44) SIGRTMAX
```

UNIX Shells

- By default, ***kill*** sends a TERM signal (number 15), which causes the receiving processes to terminate.

- To send a signal to a process, you must either own it or be a super-user.

- To ensure a kill (forcefully), send signal number 9.

- The kill utility (as opposed to the shell built-in commands) allows to specify 0 as the pid to terminate all of the processes associated with the shell.

UNIX Shells

- Example: Create a background process and then kill it.

```
$ ( sleep 10; echo done ) & ---> create background process
27390                        ---> process ID number.

$ kill 27390                 ---> kill the process.

$ ps                         ---> it's gone!
PID   TT   STAT    TIME    COMMAND
27355   p3    S          0:00   -sh(sh)
27394   p3    R          0:00   ps


$ (sleep 10;  echo  done) &
27490                        ---> process ID number.
$ kill  -KILL  27490 ---> kill the process with signal #9.
or
$ kill  -9  27490
```

UNIX Shells

- Finally, here's an example of the kill utility's ability to kill all of the processes associated with the current shell:

```
$ sleep 30 & sleep 30 & sleep 30 & ---> create three processes.
27429
27430
27431

$ kill  0                                  ---> kill them all.
27431  Terminated
27430  Terminated
27429  Terminated
```

PID = -1 means ALL processes belonging to the user
(ABSOLUTE ALL if used by the super user)

- **Waiting For Child Processes: wait**

Built-in 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.

UNIX Shells

-Example:

```
$ ( sleep 30; echo done 1 ) &        ---> create a child process.
24193

$ ( sleep 30; echo done 2 ) &        ---> create a child process.
24195

$ echo done 3; wait; echo done 4  ---> wait for children.
done 3
done 1       ---> output from first child.
done 2       ---> output from second child.
done 4
$ _
```

- **OVERLOADING STANDARD UTILITIES**

```
$ cat > ls          ---> create a script called "ls".
echo my ls
^D                  ---> end of input.
$ chmod +x  ls  ---> make it executable.

$ echo $PATH        ---> look at the current PATH setting.
/bin:/usr/bin:/usr/sbin
$ echo $HOME        ---> get pathname of my home directory.
/home/UG1

$ PATH=/home/UG1:$PATH    ---> update.
$ ls        --->  call "ls".
my ls     ---> my own version overrides "/bin/ls".
$ _
```

Note that only this shell and its child shells would be affected by the change to PATH; all other shells would be unaffected.

- TERMINATION AND EXIT CODES

In the Bash, Bourne and Korn shells, the special shell variable $? always contains the value of the previous command's exit code.

In the C shell, the $status variable holds the exit code.

UNIX Shells

-In the following example, the date utility succeeded, whereas the cc and awk utilities failed:

```
$ date              ---> date succeeds.
Mon  29  8  22:13:38  2016
$ echo $?           ---> display its exit value.
0                   ---> indicates success.

$ cc prog.c         ---> compile a nonexistent program.
cpp: Unable to open source file 'prog.c'.
$ echo $?           ---> display its exit value.
1                   ---> indicates failure.

$ awk               ---> use awk illegally.
awk: Usage: awk [-Fc] [-f source | 'cmds'] [files]
$ echo $?           ---> display its exit value.
1                   ---> indicates failure.
$ _
```

UNIX Shells

- Any script that user write should always explicitly return an exit code.

To terminate a script, use the built-in *exit* command;

Shell Command: exit number

exit terminates the script and returns the exit value number to its parent process.

If number is omitted, the exit value of the previous command is used.

If a script doesn't include an explicit exit statement, the exit value of the last command is returned by default.

-Example:

```
$ cat  script.bash    ---> look at the script.
echo  this script returns an exit code of 3
exit 3

$ script.bash         ---> execute the script.
this script returns an exit code of 3

$ echo $?             ---> look at the exit value.
3
$ _
```

- ***Eval BUILT-IN COMMAND***

The *eval* shell command executes the output of a command as a regular shell command.

It is useful for processing the output of utilities that generate shell commands.

-Example: execute the result of *echo* command:

```
$ echo  x=5                    ---> first execute an echo directly.
x=5

$ echo $x

$ eval  `echo x=5`             ---> execute the result of the echo.

$ echo $x
5
$ _
```