Job Control Commands

Certain of the following job control commands take a *job identifier* as an argument. See the table at end of the chapter.

jobs

Lists the jobs running in the background, giving the *job number*. Not as useful as ps.

It is all too easy to confuse *jobs* and *processes*. Certain builtins, such as**kill**, **disown**, and **wait** accept either a job number or a process number as an argument. The fg, bg and **jobs** commands accept only a job number.

```
bash$ sleep 100 &
[1] 1384

bash $ jobs
[1]+  Running                 sleep 100 &
```

"1" is the job number (jobs are maintained by the current shell). "1384" is the PID or *process ID number* (processes are maintained by the system). To kill this job/process, either a **kill %1** or a **kill 1384** works.

Thanks, S.C.

disown

Remove job(s) from the shell's table of active jobs.

fg, **bg**

The **fg** command switches a job running in the background into the foreground. The **bg**command restarts a suspended job, and runs it in the background. If no job number is specified, then the **fg** or **bg** command acts upon the currently running job.

wait

Suspend script execution until all jobs running in background have terminated, or until the job number or process ID specified as an option terminates. Returns the exit status of waited-for command.

You may use the **wait** command to prevent a script from exiting before a background job finishes executing (this would create a dreaded orphan process).

Example 15-26. Waiting for a process to finish before proceeding

```
#!/bin/bash

ROOT_UID=0   # Only users with $UID 0 have root privileges.
E_NOTROOT=65
E_NOPARAMS=66

if [ "$UID" -ne "$ROOT_UID" ]
```

```
then
  echo "Must be root to run this script."
  # "Run along kid, it's past your bedtime."
  exit $E_NOTROOT
fi

if [ -z "$1" ]
then
  echo "Usage: `basename $0` find-string"
  exit $E_NOPARAMS
fi


echo "Updating 'locate' database..."
echo "This may take a while."
updatedb /usr &     # Must be run as root.

wait
# Don't run the rest of the script until 'updatedb' finished.
# You want the the database updated before looking up the file name.

locate $1

#  Without the 'wait' command, in the worse case scenario,
#+ the script would exit while 'updatedb' was still running,
#+ leaving it as an orphan process.

exit 0
```

Optionally, **wait** can take a *job identifier* as an argument, for example, *wait%1* or *wait $PPID*. [1] See the job id table.

Within a script, running a command in the background with an ampersand (&) may cause the script to hang until **ENTER** is hit. This seems to occur with commands that write to `stdout`. It can be a major annoyance.

```
#!/bin/bash
# test.sh

ls -l &
echo "Done."
```

```
bash$ ./test.sh
Done.
 [bozo@localhost test-scripts]$ total 1
 -rwxr-xr-x    1 bozo     bozo           34 Oct 11 15:09 test.sh
 _
```

As Walter Brameld IV explains it:

As far as I can tell, such scripts don't actually hang. It just seems that they do because the background command writes text to the console after the prompt. The user gets the impression that the prompt was never displayed. Here's the sequence of events:

1. Script launches background command.
2. Script exits.
3. Shell displays the prompt.

4. Background command continues running and writing text to the console.
5. Background command finishes.
6. User doesn't see a prompt at the bottom of the output, thinks script is hanging.

Placing a **wait** after the background command seems to remedy this.

```
#!/bin/bash
# test.sh

ls -l &
echo "Done."
wait
```

```
bash$ ./test.sh
Done.
 [bozo@localhost test-scripts]$ total 1
 -rwxr-xr-x    1 bozo     bozo           34 Oct 11 15:09 test.sh
```

Redirecting the output of the command to a file or even to `/dev/null` also takes care of this problem.

suspend

This has a similar effect to **Control-Z**, but it suspends the shell (the shell's parent process should resume it at an appropriate time).

logout

Exit a login shell, optionally specifying an exit status.

times

Gives statistics on the system time elapsed when executing commands, in the following form:

```
0m0.020s 0m0.020s
```

This capability is of relatively limited value, since it is not common to profile and benchmark shell scripts.

kill

Forcibly terminate a process by sending it an appropriate *terminate* signal (see Example 17-6).

Example 15-27. A script that kills itself

```
#!/bin/bash
# self-destruct.sh

kill $$  # Script kills its own process here.
         # Recall that "$$" is the script's PID.

echo "This line will not echo."
# Instead, the shell sends a "Terminated" message to stdout.
```

```
exit 0   # Normal exit? No!

#  After this script terminates prematurely,
#+ what exit status does it return?
#
# sh self-destruct.sh
# echo $?
# 143
#
# 143 = 128 + 15
#             TERM signal
```

kill -l lists all the signals (as does the file /usr/include/asm/signal.h). A **kill -9** is a *sure kill*, which will usually terminate a process that stubbornly refuses to die with a plain **kill**. Sometimes, a **kill -15** works. A *zombie* process, that is, a child process that has terminated, but that the parent process has not (yet) killed, cannot be killed by a logged-on user -- you can't kill something that is already dead -- but **init** will generally clean it up sooner or later.

killall

The **killall** command kills a running process by *name*, rather than by process ID. If there are multiple instances of a particular command running, then doing a *killall* on that command will terminate them *all*.

This refers to the **killall** command in /usr/bin, *not* the killall script in /etc/rc.d/init.d.

command

The **command** directive disables aliases and functions for the command immediately following it.

```
bash$ command ls
```

This is one of three shell directives that effect script command processing. The others are builtin and enable.

builtin

Invoking **builtin BUILTIN_COMMAND** runs the command *BUILTIN_COMMAND* as a shell builtin, temporarily disabling both functions and external system commands with the same name.

enable

This either enables or disables a shell builtin command. As an example, *enable -n kill*disables the shell builtin kill, so that when Bash subsequently encounters *kill*, it invokes the external command /bin/kill.

The -a option to *enable* lists all the shell builtins, indicating whether or not they are enabled. The -f filename option lets *enable* load a builtin as a shared library (DLL) module from a properly compiled object file. [2].

autoload

This is a port to Bash of the *ksh* autoloader. With **autoload** in place, a function with an *autoload* declaration will load from an external file at its first invocation. [3] This saves system resources.

Note that *autoload* is not a part of the core Bash installation. It needs to be loaded in with`enable -f` (see above).

Table 15-1. Job identifiers

Notation	Meaning
`%N`	Job number [N]
`%S`	Invocation (command-line) of job begins with string *S*
`%?S`	Invocation (command-line) of job contains within it string *S*
`%%`	"current" job (last job stopped in foreground or started in background)
`%+`	"current" job (last job stopped in foreground or started in background)
`%-`	Last job
`$!`	Last background process

Notes

[1] This only applies to *child processes*, of course.

[2] The C source for a number of loadable builtins is typically found in the `/usr/share/doc/bash-?.??/functions` directory.

Note that the `-f` option to **enable** is not portable to all systems.

[3] The same effect as **autoload** can be achieved with typeset -fu.