Lab 4: Exam 1 practice, job control (1.2.16)

Goal: In this lab we will review selected problems and learn about job control.

At the beginning, TA will discuss problems 4, 5 and 6 from In-Sem 1 of previous year. After that, do the Exercise below. For checkoff, show your solution to Exercise 2 of lab 3 to TA. If anything is not clear, please ask the TA for help.

Exercise 1

Job control is a feature provided by many shells (Bash and Tcsh included) which allows you to control multiple running commands, or jobs, at once. The command ps displays a list of currently running processes. Here's an example:

```
[user@host ~]  ps


   PID TT        TIME        COMMAND
  1482  tty1     00:00:00    (bash)
  1538  tty1     00:00:00    ps

[user@host ~]
```

The PID listed in the first column is the process ID, a unique number given to every running process. The last column, COMMAND, is the name of the running command. Here. we re only looking at the processes which Issstudent is currently running. These are bash and the ps command itself. As you can see, bash is running concurrently with the ps command. bash executed ps when issstudent typed the command ps. After ps is finished running (after the table of processes is displayed), control is returned to the bash process, which displays the prompt, ready for another command.

A running process is known as a *job* to the shell. The terms *process* and *job* are interchangeable. However, a process is usually referred to as a "job" when used in conjunction with job control—a feature of the shell which allows you to switch between several independent jobs.

There are many other processes running on the system as well "ps -aux" lists them all.

In most cases users are only running a single job at a time-that being whatever command they last typed to the shell. However, using job control, you can run several jobs at once, switching between them as needed. How might this be useful? Let's say that you're editing a text file and need to suddenly interrupt your editing and do something else. With job control, you can temporarily suspend the editor, and back at the shell prompt start to work on something else. When you're done, you can start the editor back up, and be back where you started, as if you never left the editor. This is just one example.

Foreground and background

Jobs can either be in the foreground or in the background. There can only be one job in the foreground at any one time. The foreground job is the job which you interact with—it receives input from the keyboard and sends output to your screen. On the other hand, jobs in the background do not receive input from the terminal, in general, they run along quietly without need for interaction.

Some jobs take a long time to finish, and don't do anything interesting while they are running. Compiling programs is one such job, as is compressing a large file. There's no reason why you should sit around being bored while these jobs complete their tasks: you can just run them in the background. While the jobs are running in the background, you are free to run other programs.

Jobs may also be suspended. A suspended job is a job that is not currently running, but is temporarily stopped. After you suspend a job, you can tell the job to continue in the foreground or the background as needed. Resuming a suspended job will not change the state of the job in any way—the job will continue to run where it left off.

Note that suspending a job is not equal to *interrupting* a job. When you interrupt a running process (by hitting your interrupt key. which is usually ctrl-C), it kills the process, for good. Once the job is killed, there's no hope of resuming it: you'll have to re-run the command. Also note that some programs trap the interrupt, so that hitting ctrl-C won't immediately kill the job. This is to allow the program to perform any necessary cleanup operations before exiting. In fact, some programs simply don't allow you to kill them with an interrupt at all.

Backgrounding and killing jobs

Let's begin with a simple example. The command yes is a seemingly useless command, which sends an endless stream of y's to standard output. (This is actually useful. If youpiped the output of yes to another command which asked a series of yes and no questions, the stream of y's would confirm all of the questions.)

Try it out.

```
[user@host ~] yes

y
y
y
y
y
```

The y's will continue *ad infinitum.* You can kill the process by hitting your interrupt key, which is usually ctrl-C . So that we don't have to put up with the annoying stream of ys, let's redirect the standard output of yes to /dev/null. As you may of the term bit bucket. /dev/null acts as a "black hole" for data. Any data sent to it will disappear. This is a very effective method of quieting an otherwise verbose program.

```
[user@host ~] yes > /dev/nullI
```

Ah, much better. Nothing is printed, but the shell prompt doesn't come back. This is because yes is still running, and is sending those inane y's to /dev/null. Again, to kill the job, hit the interrupt key.

Let’s suppose that we wanted the yes command to continue to run, but wanted to get our shell prompt back to work on other things. We can put yes into the background, which will allow it to run, but without need for interaction.

One way to put a process in the background is to append an “&“ character to the end of the command.

```
[user@host ~] yes > /dev/nuII &
[1] 1543

[user@host ~]
```

As you can see, we have our shell prompt back. But what is this “[1] 1543”? And is the yes command really running?

The “[1]” represents the job number for the yes process. The shell assigns a job number to every running job. Because yes is the one and only job that we’re currently running. it is assigned job number 1. The “1543” is the process ID, or PID, number given by the system to the job. Either number may be used to refer to the job, as we’ll see later.

You now have the yes process running in the background, continuously sending a stream of y’s to /dev/null. To check on the status of this process, use the shell internal command jobs.

```
[user@host ~] jobs
[1]+ Running                 yes > /dev/null &

[user@host ~]
```

Sure enough, there it is. You could also use the ps command as demonstrated above to check on the status of the job.

To terminate the job, use the command kill. This command takes either a job number or a process ID number as an argument. This was job number 1, so using the command

```
[user@host ~] kill %1
```

will kill the job. When identifying the job with the job number, you must prefix the number with a percent (“%“) character.

Now that we’ve killed the job, we can use jobs again to check on it:

```
[user@host ~] jobs

[11+ Terminated                 yes >/dev/null

[user@host ~]
```

The job is in fact dead, and if we use the jobs command again nothing should be printed.

You can also kill the job using the process ID (PID) number, which is printed along with the job ID when you start the job. In our example, the process ID is 1543, so the command

```
[user@host ~] kill 1543
```

is equivalent to

```
[user@host ~] kill %1
```

You don't need to use the "%", when referring to a job by its process ID.

Stopping and restarting jobs

There is another way to put a job into the background. You can start the job normally (in the foreground), stop the job, and then restart it in the background.

First, start the yes process in the foreground, as you normally would:

```
[user@host ~] yes > /dev/null
```

Again, because yes is running in the foreground, you shouldn't get your shell prompt back.

Now, instead of interrupting the job with ctrl-C . we'll *suspend* the job. Suspending a job doesn't kill it: it only temporarily stops the job until you restart it. To do this, you hit the suspend key, which is usually ctrl-Z

```
[student@it215 ~] yes > /dev/null
ctrl-Z

[1]+ Stopped                 yes >/dev/null

[user@host ~]
```

While the job is suspended, it's simply not running. No CPU time is used for the job. However, you can restart the job, which will cause the job to run again as if nothing ever happened. It will continue to run where it left off.

To restart the job in the foreground, use the command fg (for "foreground").

```
[user@host ~] fg

yes >/dev/null
```

The shell prints the name of the command again so you're aware of which job you just put into the foreground. Stop the job again, with ctrl-Z . This time, use the command bg to put the job into the background. This will cause the command to run just as if you started the command with “&“ as in the last section.

```
[user@host ~] bg

[11+ yes >/dev/null &

[user@host ~]
```

And we have our prompt back, jobs should report that yes is indeed running, and we can kill the job with kill as we did before.

How can we stop the job again? Using ctrl-Z won’t work, because the job is in the background. The answer is to put the job in the foreground,with fg, and then stop it. As it turns out you can use fg on either stopped jobs or jobs in the background.

There is a big difference between a job in the background and a job which is stopped. A stopped job is not running-it’s not using any CPU time, and it’s not doing any work (the job still occupies system memory, although it may be swapped out to disk). A job in the background is running, and using memory, as well as completing some task while you do other work. However, a job in the background may try to display text on to your terminal, which can be annoying if you’re trying to work on something else. For example. if you used the command

```
[user@host ~] yes &
```

without redirecting stdout to /dev/null, a stream of y’s would be printed to your screen, without any way of interrupting it (you can’t use ctrl-C to interrupt jobs in the background). In order to stop the endless y’s, you’d have to use the fg command, to bring the job to the foreground, and then use ctl-C to kill it.

Another note. The fg and bg commands normally foreground or background the job which was last stopped (indicated by a “+“ next to the job number when you use the command jobs). If you are running multiple jobs at once, you can foreground or background a specific job by giving the job ID as an argument to fg or bg, as in

```
[user@host ~] fg %2
```

(to foreground job number 2), or

```
[user@host ~] bg %3
```

(to background job number 3). You can’t use process ID numbers with fg or bg.

Furthermore, using the job number alone, as in

```
[user@host ~] 2
```

is equivalent to

```
[user@host ~]fg %2
```

Just remember that using job control is a feature of the shell. The commands fg, bg and jobs are internal to the shell. If for some reason you use a shell which, does not support job control, don’t expect to find these commands available.

In addition, there are some aspects of job control which differ between Bash and Tcsh (do some research on this). In fact, some shells don’t provide job control at all-however, most shells available for Linux support job control.