Linux Fu: Walk, Chew Gum

Originalartikel

Backup

<html> <p>If you ever think about it, computers are exceedingly stupid. Even the most powerful CPU can&#8217;t do very much. However, it can do what it does very rapidly and repeatably. Computers are so fast, they can appear to do a lot of things at once, too and modern computers have multiple CPUs to further enhance their multitasking abilities. However, we often don&#8217;t write programs or shell scripts to take advantage of this. However, there&#8217;s no reason for this, as you&#8217;ll see.</p><h2>Bash Support</h2><p>It is surprisingly easy to get multiple processes running under Bash. For one thing, processes on either side of a pipe run together, and that&#8217;s probably the most common way shell scripts using multiprogramming. In other words, think about what happens if you write

```
ls | more
```

.</p><p>Under the old MSDOS system, the first program would run to completion, spooling its output to a temporary file. Then the second program will run, reading input from the same file. With Linux and most other modern operating systems, both programs will run together with the input of the second program connected to the first program&#8217;s output.</p><p>That&#8217;s easy because the programs synchronize themselves over the input and output channel. However, it is just as easy to start multiple programs independently. The key is using &#8220;&amp;&#8221; to separate programs or end the script line.</p><p>It is easy enough to convert a script like:</p><pre class=„brush: bash; title: ; notranslate“ title=„“>find /mnt1/usr -name '*.so' &gt;/tmp/libs1find /mnt2/usr -name '*.so' &gt; /tmp/libs2# tofind /mnt1/usr -name '*.so' &gt;/tmp/libs1 &amp;find /mnt2/usr -name '*.so' &gt; /tmp/libs2 &amp;</pre><p>In the first case, the searches occur one at a time, and only proceeds after the last find has run. In the second case, both commands run at the same time and the script will continue even while both commands are still going.</p><h2>The Problem</h2><p>There&#8217;s only one problem. While you can spin off multiple programs to run together, it is rare that these programs don&#8217;t need to coordinate with each other.</p><p>Not that it would be useful to really run these in parallel, but take a look at the output of this command:</p><pre class=„brush: bash; title: ; notranslate“ title=„“>alw@Enterprise:~$ banner hello &amp; banner goodbye[1] 173# # ###### # # ######## #### #### ##### ##### # # ####### # # # # # ## # # # # # # # # # # # ####### ##### # # # ## # # # # # # ##### # ###### # # # # # ## ### # # # # # # # # # # ## # # # # # # ## # # # # # # # # # # ## # ###### ###### ###### ######## #### #### ##### ##### # ######[1]+ Done banner hello</pre><p>Not what you expected. Depending on your system, the first program may (or may not) get all of its output done in one go. Interspersing output isn&#8217;t really what you want.</p><h2>Control</h2><p>There are other problems. You might want to find out the PID of the new process. You could use <a href=„https://www.gnu.org/software/bash/manual/html_node/Job-Control-Basics.html“ target=„_blank“>bash&#8217;s job control</a>. However, you don&#8217;t really need to go that far.</p><p>The

```
jobs
```

command will show you what you have running in the background from the current shell. If you add

```
-l
```

or

```
-p
```

you can also learn the PID.</p><p>An easier way to learn the PID of a command is using

```
$!
```

. Of course, if you are running from the shell&#8217;s command prompt, it also tells you the PID of the last run command. For example:</p><pre class=„brush: bash; title: ; notranslate“ title=„“>ls / &amp; echo PID=$![2] 178PID=178</pre><p>Of course, your PID number will almost certainly be different.</p><h2>Then What?</h2><p>Armed the PID, what can you do with it? There are two things you&#8217;ll find most useful. First, you can use

```
wait
```

to understand when a process (or processes) are complete.</p><p>The other thing you can do is use

```
kill
```

to send <a href=„https://hackaday.com/2019/08/26/linux-fu-its-a-trap/“>signals</a> to the background program. That&#8217;s beyond the scope of this post, but you can use signals to create complex communication between programs or to invoke default behaviors like termination.</p><h2>An Example</h2><p>Consider a case where you have a bunch of

```
jpeg
```

files and you want to convert them to

```
png
```

files for a website using

```
ImageMagick
```

. You might try this simple script:</p><pre class=„brush: bash; title: ; notranslate“ title=„“>#!/bin/bashfor I in *.jpgdo convert „$I“ „png/$(basename „$I“ .jpg).png“doneecho Copying files now...ls png</pre><p>This will do the job. Presumably, the last line will have some file copy command like an

```
sftp
```

following it, but I used a directory listing just for an example.</p><p>Instead, you could launch all the conversions at once, taking advantage of multiple processors, and wait for them all to finish up. Without the wait command, the simulated copy would start before the conversions were complete unless there were very few conversions to do.</p><pre class=„brush: bash; title: ; notranslate“

title=„“>#!/bin/bashfor I in *.jpgdo convert „$I“ „png/$(basename „$I“ .jpg).png“ &amp;donewaitecho Copying files now...ls png</pre><h2>Still a Problem</h2><p>There is still one problem. The wait command will wait for any subprocesses active in the shell. That might not be what you want, although in this case, it is probably OK. However, let&#8217;s fix it:</p><pre class=„brush: bash; title: ; notranslate“ title=„“>#!/bin/bashPIDs=„„for I in *.jpgdo convert „$I“ „png/$(basename „$I“ .jpg).png“ &amp;PIDs+=„$! „donewait $PIDsecho Copying files...ls png</pre><p>If you run the timing with a good number of files, you&#8217;ll see a big difference. On my laptop with a handful of pictures, the straight versions took about 40 seconds. It took just over 10 seconds with the final version.</p><h2>Wrap Up</h2><p>It is easy to forget that you can do more than one thing at a time pretty easily. Of course, this also opens up a whole new realm of problems. If you need to protect your programs from each other, check out our <a href=„https://hackaday.com/2020/08/18/linux-fu-one-at-a-time-please-critical-sections-in-bash-scripts/“>earlier post about critical sections</a>. Not everyone thinks

```
bash
```

is a great programming language, but it is <a href=„https://hackaday.com/2017/07/21/linux-fu-better-bash-scripting/“>surprisingly capable</a> and while it might not be good for everything, it is great for some tasks.</p> </html>