Queuing tasks for batch execution with Task Spooler

Originalartikel

Backup

<html> The Task Spooler project allows you to queue up tasks from the shell for batch execution. Task Spooler is simple to use and requires no configuration. You can view and edit queued commands, and you can view the output of queued commands at any time. Task Spooler has some similarities with other delayed and batch execution projects, such as „at.“ While both Task Spooler and at handle multiple queues and allow the execution of commands at a later point, the at project handles output from commands by emailing the results to the user who queued the command, while Task Spooler allows you to get at the results from the command line instead. Another major difference is that Task Spooler is not aimed at executing commands at a specific time, but rather at simply adding to and executing commands from queues. The main repositories for Fedora, openSUSE, and Ubuntu do not contain packages for Task Spooler. There are packages for some versions of Debian, Ubuntu, and openSUSE 10.x available along with the source code on the project's homepage. In this article I'll use a 64-bit Fedora 9 machine and install version 0.6 of Task Spooler from source. Task Spooler does not use autotools to build, so to install it, simply run

```
make; sudo make install
```

. This will install the main Task Spooler command

```
ts
```

and its manual page into /usr/local. A simple interaction with Task Spooler is shown below. First I add a new job to the queue and check the status. As the command is a very simple one, it is likely to have been executed immediately. Executing ts by itself with no arguments shows the executing queue, including tasks that have completed. I then use

```
ts -c
```

to get at the stdout of the executed command. The

```
-c
```

option uses

```
cat
```

to display the output file for a task. Using

```
ts -i
```

shows you information about the job. To clear finished jobs from the queue, use the

```
ts -C
```

command, not shown in the example. $ ts echo „hello world“
6 $ ts
ID State Output E-Level Times(r/u/s) Command [run=0/1]
6 finished /tmp/ts-out.QoKfo9 0 0.00/0.00/0.00 echo hello world $ ts -c 6
hello world $ ts -i 6
Command: echo hello world
Enqueue time: Tue Jul 22 14:42:22 2008
Start time: Tue Jul 22 14:42:22 2008
End time: Tue Jul 22 14:42:22 2008
Time run: 0.003336s   The

```
-t
```

option operates like

```
tail -f
```

, showing you the last few lines of output and continuing to show you any new output from the task. If you would like to be notified when a task has completed, you can use the

```
-m
```

option to have the results mailed to you, or you can queue another command to be executed that just performs the notification. For example, I might add a tar command and want to know when it has completed. The below commands will create a tarball and use libnotify commands to create an inobtrusive popup window on my desktop when the tarball creation is complete. The popup will be dismissed automatically after a timeout. $ ts tar czvf /tmp/mytarball.tar.gz liberror-2.1.80011
11
$ ts notify-send „tarball creation“ „the long running tar creation process is complete.“
12
$ ts
ID State Output E-Level Times(r/u/s) Command [run=0/1]
11 finished /tmp/ts-out.O6epsS 0 4.64/4.31/0.29 tar czvf /tmp/mytarball.tar.gz liberror-2.1.80011
12 finished /tmp/ts-out.4KbPSE 0 0.05/0.00/0.02 notify-send tarball creation the long… is complete. Notice in the output above, toward the far right of the header information, the

```
run=0/1
```

line. This tells you that Task Spooler is executing nothing, and can possibly execute one task. Task spooler allows you to execute multiple tasks at once from your task queue to take advantage of multicore CPUs. The

```
-S
```

option allows you to set how many tasks can be executed in parallel from the queue, as shown below. $ ts -S 2
$ ts
ID State Output E-Level Times(r/u/s) Command [run=0/2]
6 finished /tmp/ts-out.QoKfo9 0 0.00/0.00/0.00 echo hello world If you have two tasks that you want to execute with Task Spooler but one depends on the other having already been executed (and perhaps that the previous job has succeeded too) you can handle this by having one task wait for the other to complete before executing. This becomes more important on a quad core machine when you might have told Task Spooler that it can execute three tasks in parallel. The commands shown below create an explicit dependency, making sure that the second command is

executed only if the first has completed successfully, even when the queue allows multiple tasks to be executed. The first command is queued normally using

```
ts
```

. I use a subshell to execute the commands by having

```
ts
```

explicitly start a new bash shell. The second command uses the

```
-d
```

option, which tells

```
ts
```

to execute the command only after the successful completion of the last command that was appended to the queue. When I first inspect the queue I can see that the first command (28) is executing. The second command is queued but has not been added to the list of executing tasks because Task Spooler is aware that it cannot execute until task 28 is complete. The second time I view the queue, both tasks have completed. $ ts bash -c „sleep 10; echo hi“
28
$ ts -d echo there
29
$ ts
ID State Output E-Level Times(r/u/s) Command [run=1/2]
28 running /tmp/ts-out.hKqDva bash -c sleep 10; echo hi
29 queued (file) && echo there
$ ts
ID State Output E-Level Times(r/u/s) Command [run=0/2]
28 finished /tmp/ts-out.hKqDva 0 10.01/0.00/0.01 bash -c sleep 10; echo hi
29 finished /tmp/ts-out.VDtVp7 0 0.00/0.00/0.00 && echo there
$ cat /tmp/ts-out.hKqDva
hi
$ cat /tmp/ts-out.VDtVp7
there You can also explicitly set dependencies on other tasks as shown below. Because the

```
ts
```

command prints the ID of a new task to the console, the first command puts that ID into a shell variable for use in the second command. The second command passes the task ID of the first task to ts, telling it to wait for the task with that ID to complete before returning. Because this is joined with the command we wish to execute with the

```
&amp;amp;&amp;amp;
```

operation, the second command will execute only if the first one has finished and succeeded. The first time we view the queue you can see that both tasks are running. The first task will be in the

```
sleep
```

command that we used explicitly to slow down its execution. The second command will be executing

```
ts
```

, which will be waiting for the first task to complete. One downside of tracking dependencies this way

is that the second command is added to the running queue even though it cannot do anything until the first task is complete. $ FIRST_TASKID=`ts bash -c „sleep 10; echo hi“`
$ ts sh -c „ts -w $FIRST_TASKID & echo there“
25
$ ts
ID State Output E-Level Times(r/u/s) Command [run=2/2]
24 running /tmp/ts-out.La9Gmz bash -c sleep 10; echo hi
25 running /tmp/ts-out.Zr2n5u sh -c ts -w 24 & echo there
$ ts
ID State Output E-Level Times(r/u/s) Command [run=0/2]
24 finished /tmp/ts-out.La9Gmz 0 10.01/0.00/0.00 bash -c sleep 10; echo hi
25 finished /tmp/ts-out.Zr2n5u 0 9.47/0.00/0.01 sh -c ts -w 24 & echo there
$ ts -c 24
hi
$ ts -c 25
there <h4>Wrap-up</h4> Task Spooler allows you to convert a shell command to a queued command by simply prepending

```
ts
```

to the command line. One major advantage of using ts over something like the

```
at
```

command is that you can effectively run

```
tail -f
```

on the output of a running task and also get at the output of completed tasks from the command line. The utility's ability to execute multiple tasks in parallel is very handy if you are running on a multicore CPU. Because you can explicitly wait for a task, you can set up very complex interactions where you might have several tasks running at once and have jobs that depend on multiple other tasks to complete successfully before they can execute. Because you can make explicitly dependant tasks take up slots in the actively running task queue, you can effectively delay the execution of the queue until a time of your choosing. For example, if you queue up a task that waits for a specific time before returning successfully and have a small group of other tasks that are dependent on this first task to complete, then no tasks in the queue will execute until the first task completes. </html>