Running Jobs Unattended

By Jerry Peek

Linux systems have several ways to run jobs unattended at some later time — either once or many times. Scheduling jobs in the future is handy for system maintenance, for sending yourself reminders, and for making more practical and efficient use of your CPU by running jobs when users are away.

As usual though, there are gotchas: if you want to run multiple commands or run commands that need a particular setup (including environment variables, a tty, or a certain current directory), or want to avoid system overload, you need to know a few tricks. This month, let's dig into job scheduling systems, discuss the potential problems, and find some answers.

Catching some Zs

An easy way to delay execution is with the *sleep* command. *sleep* simply waits for some number of seconds (for instance, `sleep 60` waits sixty seconds) and then exits.

at jobs are great for running jobs irregularly, minutes or months in the future

For example, let's say you're going to lunch and you have three big files named *a*, *b*, and *c* to print on your office laser printer. However, you don't want to tie up the printer constantly. Here's how you could run one *lpr* print job, wait ten minutes, run the next job, and so on:

```
% lpr a; sleep 600; lpr b; sleep 600; lpr c
```

The `;` (semicolon) is the shell's command separator. With it, you can string several commands together at a single prompt. Commands run in the order you type them, and you won't get another shell prompt until the last command (`lpr c`) finishes.

If that's a problem, you can run the command sequence in a separate *xterm* window. Or — especially if you have just one console where you need to run commands while *sleep* is sleeping — run the command in the background with the shell's `&` (ampersand) operator. Be sure to use a subshell (the shell's `( )` operators) to put the entire sequence into the background together.

For instance, to pop open an *xmessage* window four hours from now with the message, "Meet Jim," type:

```
% (sleep 14400; xmessage -near Meet Jim) &
```

The shell silently runs *sleep* for 14,400 seconds, and then runs *xmessage*.

Using *sleep* has advantages over *cron* and *at* (we'll look at *cron* and *at* momentarily). For instance, since *sleep* is typically used in a shell window (like *xterm*), it can open other windows. Other techniques can't open windows because they aren't run from your terminal under X.

at: Doing It at a Later Time

The *at* utility queues one or more commands to run at some time in the future. A daemon (or a special *cron* job) runs jobs submitted by *at*.

Using *at*, you can schedule jobs to run minutes or months into the future. Typically, you put the job time on the command line, and enter the commands to run on *at*'s standard input. Type `man at` for details (there are several versions of *at*, and their command line syntaxes vary.)

Here's a simple example. Let's say you need two things done overnight: copy a huge file with *wget*, and email yourself a note that'll be waiting for you when you come into work. To run a job less than 24 hours in the future, just give the time: for example, `3:04am` or `0304` for 3:04 in the morning. (If your system is down overnight, any jobs missed should be run soon after the system restarts.) Type the command lines from the terminal, which is *at*'s standard input. End input with CTRL-D at the start of a line. (Your *at* may read input silently instead of prompting.)

```
$ at 0304
at> wget -q ftp://z.co.uk/pub/file.tar.gz
at> echo Write the report | mail jpeek
at> ^D
job 6 at 2003-01-16 03:04
$
```

At 3:04 AM, the commands are run by a shell. (*at* often runs a Bourne-type shell like *bash*, but your *at* system may use *csh*. If you want to discover which shell *at* uses, have *at* run a command like `ps $$ | mail yourname` to email the shell's name to yourself.)

To run several commands, or to make a job repeat, put your job in a file. For example, the Bourne shell script *cruncher.at* (the *.at* suffix isn't required), shown in *Listing One*, uses an `if` statement to check the system load average,

taken from the file */proc/loadavg* (however, using the *uptime* command would be more portable). If the one-minute load average is less than 1, a program named *cruncher* is run. Otherwise, *cruncher.at* is re-submitted to run 10 minutes later. Let's submit the job.

Although some versions of *at* accept filenames on the command line, all versions can read jobs from standard input. So (as you see in the *cruncher.at* script file) we'll use the shell's `<` (less than) operator to redirect the file to *at*'s standard input:

```
$ at 0234 < cruncher.at
job 7 at 2003-01-16 02:34
```

The job is scheduled to run at 2:34 AM or a little later (many systems run *at* jobs every five minutes or more). You can check the job queue with *atq* or `at -q`.

If a job produces output, the *at* daemon mails it to you. But, if you expect output, you should probably pipe the output to an email program, redirect it to a file, or discard the output by redirecting it to */dev/null* (the Linux "bit bucket"). That way, you only get email from the *at* daemon if something goes wrong.

Also note that redirecting the output of *at* itself (when you submit the job) won't redirect output of the job as it runs! That's because *at* simply submits the job to a queue. You have to redirect the output of each individual command *in the queued job* (or in Bourne-like shells, use *exec* to redirect all output to a file).

For instance, the first line of our next version of *cruncher.at*, shown in *Listing Two*, appends all output (both standard output and standard error) from all following commands to the file *cruncher.at.log*. The second line outputs a message (written to *cruncher.at.log*) with the current date and time.

Introducing *cron*

at jobs are great for running jobs irregularly or for running jobs (like *cruncher.at*) that resubmit themselves. To run a job periodically — from once a minute to once a year — it's easier to use *cron* (or variant *anacron*, described later).

There are several versions of *cron*. All versions use a daemon to runs jobs, and each includes a master *crontab* file that lists jobs run for root, the superuser. While system administrators maintain root's *crontab*, many versions of *cron* also provide a personal *crontab* file for each user — albeit with a slightly different format.

Here's a simple personal *crontab* entry. At 1:09 AM every morning, run the script *$HOME/bin/ci_crontab:*

```
9 1 * * * bin/ci_crontab
```

Here, the first five entries (`9`, `1`, `*`, `*`, and `*` are *minutes*, *hours*, *day of month*, *month*, and *day of week*, respectively, and the last entry is the command to run.

Each of the first five fields can have a specific value (like `9` for nine minutes past the hour), a set of values separated by a comma (like `11,12` for November and December), or a range of values separated by a hyphen (like `18-23` for 6 PM through 11 PM). An asterisk (`*`) is a placeholder that means "any." (This is the format of *crontab* entries on Red Hat. Run `man crontab` on your system to see if your *crontab* format differs).

Use *cron* to run jobs periodically — from once a minute to once a year

The *ci_crontab* script (available at http://www.linux-magazine.com/downloads/2003-02/power/ci_crontab.txt) compares the current *crontab* file to the previous version stored in an RCS archive. If the file has changed, *ci_crontab* runs *ci* to check it into RCS. (*crontab* files can be easily lost or overwritten; this script makes it easy to recover old *crontab* entries.)

You may be wondering about the relative pathname to the script, `bin/ci_crontab`. That's a good lead-in to the tangly topic of *cron*'s execution environment. (The next section also generally applies to *anacron*.)

LISTING ONE: *cruncher.at:* If the load is low, run a big job, otherwise delay

```
loadavg=`sed 's/\..*//' /proc/loadavg`
if [ "$loadavg" -lt 1 ]
then
  cruncher
else
  at now + 10 min < cruncher.at
fi
```

LISTING TWO: Redirecting the output of an *at* job

```
exec >> cruncher.at.log 2>&1
echo cruncher.at: starting at `date`
loadavg=`sed 's/\..*//' /proc/loadavg`
if [ "$loadavg" -lt 1 ]
then
  cruncher
else
  at now + 10 min < cruncher.at
fi
```

Exploring the Environment

The *cron* daemon, which reads *crontab* files and runs jobs from them, doesn't run from your terminal. So, *cron* doesn't have the same environment as your shell. Environment variables and other settings like the current directory, PATH (a list of directories with program files), and TZ (your timezone), are not defined. There's at least one exception: each *cron* job starts in your home directory, or `$HOME`. So, files you read or write without a pathname will be in your home directory.

This is why our sample entry uses the relative pathname `bin/ci_crontab`: the *ci_crontab* script is in the *$HOME/bin* directory, which isn't in *cron*'s PATH. (You can set environment variables within a *crontab* file, but it may just be simpler to use a pathname.)

cron passes each *crontab* entry to a different instance of (typically) the Bourne shell. Within each entry, you can set its shell's environment — just as you can type a series of environment-setting commands at the shell prompt on a terminal. Separate multiple commands with semicolons (`;`) or conditional operators. Let's look at some examples.

anacron is a better choice if your machine isn't on 24 hours a day, 7 days a week

```
19 16 14 1 *  (env; id; set) > cron-stuff
```

This first example uses a subshell to run three commands on January 14 at 4:19 PM: *env* shows *cron*'s environment variables, *id* displays its UID and GID, and *set* lists the settings of all shell variables. The modifier `"> cron-stuff"` redirects all output to the file *cron-stuff* in the home directory. (This first example is handy to run just once: save its output and then delete the entry from your *crontab*.)

Let's look at a more complex example, shown in *Listing Three*. *Listing Three* (which *must* be entered as a single *crontab* line) runs three commands at 11:03 PM local time, Sunday through Thursday (the day of the week is specified by `0-4`, where `0`=Sunday, `1`=Monday, and so on). Why times like 4:19 PM and 11:03 PM, instead of "normal" times like 4:30 PM or 11:00 PM? See the sidebar, "*Cron* Job Overload."

The first command sets the timezone enviroment variable to *Japan* for the remainder of the *crontab* entry.

The second command uses shell command substitution to set a shell variable named `day` with the current year, month, and date (in Japan), like `20030114`. This version of *cron* requires literal percent signs (`%`) to be escaped (`\%`). We'll soon see why that's necessary.

The third command changes the current directory to */tokyo*. If it succeeds, `cd` returns "true" (a status code of 0), and the shell's conditional operator `&&` executes */tokyo/crunch*. (If */tokyo* weren't accessible, `cd` would fail, and the rest of the command would be skipped.) Because *cron*'s `PATH` doesn't include the current directory (`.`), the pathname must be `./crunch`. The *crunch* output is written to a file like */tokyo/logs/crunch.20030114*.

And one more example:

```
17 4 * * *    ed - /prj/log%1,$g/^NOTE:/d%w
```

This entry is like typing the following on a terminal:

```
$ ed - /prj/log
1,$g/^NOTE:/d
w
```

Here, the entry uses the `%` (percent) operator, which some *crontab* files support, to pass an editing script to the standard input of the *ed* editor. The percent operator also lets you embed newline characters in a *crontab* entry — which is hard to do otherwise because *crontab* entries must be on a single line. Any text after the first `%` is fed to the standard input of the command, with each following `%` replaced by a newline. With techniques like these, you can write entire scripts in single-line *crontab* entries. But you may want to put more-complex entries into a script file, possibly in a library directory full of *cron* scripts.

anacron: Doing It When You Can

Your Linux machine may not be up 24 hours a day, 7 days a week. For example, if your Linux machine is a laptop, you may only turn it on from time to time. Because *cron* jobs are tied to particular times of the day (or week or year), *anacron*, a variant of *cron*, might be a better choice for scheduling jobs on these machines.

Let's take a look at a slightly-edited *anacrontab* file from a Red Hat system:

LISTING THREE: Setting environment variables in *cron* jobs

```
3 23 * * 0-4  export TZ=Japan; day=`date +\%Y\%m\%d`; cd /tokyo && ./crunch > logs/crunch.$day
```

```
PATH=/sbin:/bin:/usr/sbin:/usr/bin
1  5  day     run-parts /etc/cron.daily
7  10 week    run-parts /etc/cron.weekly
30 15 month   run-parts /etc/cron.monthly
```

Each job — here, each entry in the *anacrontab* — has a name and a corresponding timestamp file. When *anacron* starts, it looks for jobs that haven't been executed in the last *n* days, where *n* is the time period listed in the first field of an entry. If a job needs running, *anacron* delays the number of minutes in the second field, passes the command to a shell, and then updates the timestamp named in the third field. Delays can give the system time to start other processes — including, in the case of *anacron*, other *anacrontab* jobs.

In the listing shown above, the first entry sets the search path. The other three entries run jobs. For example, the last entry checks the `month` timestamp. If the job hasn't run in the past 30 days (the `30` field), then *anacron* delays 15 minutes (the `15`) before running the job (`run-parts /etc/cron.monthly`).

In this example, *run-parts* is a simple shell script that looks in a directory like */etc/cron.monthly* for executable programs and runs them all. This is a handy way to avoid cluttering the *anacrontab* with lots of entries. (You can use the same technique in a *crontab*.)

anacron considers each entry in the *anacrontab* unless you specify certain jobs (here, `day`, `week`, or `month`) on its command line. You can use shell-type wildcards to specify jobs.

For instance, your *anacrontab* could have entries named *news.daily*, *news.weekly*, *http.daily*, *http.weekly*, and so on. Running `anacron '*.daily'` (the quotes are necessary to prevent the shell from expanding `*`) would check the jobs *news.daily* and *http.daily*. The command line `anacron 'news.*'` would run jobs whose names start with `news.`.

anacron runs all jobs that it should and then exits. So two places to start it are in your system startup files (*/etc/rc**) or periodically (once a day, or more) from either your personal or system-wide *crontab* file.

CRON JOB OVERLOAD

Do you punch in "even" cook times, like 1:00, on your microwave oven? Maybe this is the same habit that makes a lot of us run all of our *at* and *cron* jobs at the top of the hour — sometimes overloading a system hourly.

Just as you can add some spice to your microwaving by warming your coffee for 57 seconds, you can choose "odd" times like 4:09 AM for your *cron* jobs, unless the job really *needs* to run exactly on the hour. Assigning different times to each job levels the load and reduces the chance that your CPU-intensive job runs at the same time as someone else's.

Use *sleep* directly from the command line to schedule something to run later

Looking Back and Ahead

We've looked at several ways to delay and repeat execution: using *sleep* from a terminal, *at* to run jobs at irregular intervals, *cron* to run jobs periodically, and *anacron* to supplement *cron* on systems that aren't up 24x7.

Next time, we'll explore the dark corners of file transfers.

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for over 20 years. He's happy to hear from readers at jpeek@jpeek.com.

POWER TIP: Remembered Directories

If you repeatedly *cd* between the same few directories, you don't have to type `cd` *`pathname`* over and over. Many shells support the command `cd –`, which means "go back to the previous directory."

To change between several directories over and over, don't use *cd*. Instead, use the shell's *directory stack*, a first-in-first-out list, and the commands *pushd* and *popd*, to "push" (add) and "pop" (remove) directories, respectively.

With an argument, *pushd* adds your current directory to the top of the directory stack and then changes to the directory you specified. With no argument, *pushd* swaps your current directory with the directory on the top of the stack.

The *popd* command removes the top directory from the stack and changes to that directory. (*pushd* and *popd* also accept arguments like *+n* to specify the *n*th directory on the stack.)

Finally, *dirs* shows your current directory and the current state of the stack. Here's a brief example:

```
% pwd
/a/b
% pushd /c/d
% dirs
/c/d /a/b
% pushd /e/f
% dirs
/e/f /c/d /a/b
% popd
/c/d /a/b
% pwd
/c/d
% dirs
/c/d /a/b
```

There's much more to directory stacks! See a good reference about shells — and next month's Power Tip.