Cron: Automate

A *Cron* job a day helps you work, rest and play. **Dr Chris Brown** shows how to configure crontabs and use *Anacron* to have your system running like clockwork.

How's your Greek? Did you know that Χρόνος (chronos) means 'time'? Sure you did! And in Greek mythology, Chronos was the god of the ages, the personification of time. Hence we have words like chronometer and chronology. Given that programmers have never been famous for being that great at spelling, we also have *Cron* – a Linux service that arranges for actions to take place at specific times.

In this tutorial you'll learn how to configure *Cron* to schedule your own jobs, and how to make sense of the *Cron* configuration that comes with your Linux distribution. *Cron* is great for machines that are left running continuously, but in the second half of the tutorial we'll look at its younger sister, *Anacron*, which may be more appropriate for personal computers that spend a lot of their time switched off.

Our expert

Dr Chris Brown is a freelance Linux instructor with a PhD in particle physics and Novell CLP and Red Hat RHCE certification. He has just written a book on SUSE Linux for O'Reilly.

Part 1: Mastering Cron

Cron has been around a long time (though not quite so long as Chronos). Even my bedraggled copy of the manual for Edition Six Unix, dated 1974, has an entry for it. It has been rewritten several times since then, and the version found in Linux distributions today is usually known as *Vixie Cron*, written by Paul Vixie.

Simply, *Cron* is a daemon that arranges for commands to be run at specific times of day, and/or specific days of the week. It works by waking up once every minute and consulting its configuration files – known as crontabs – to see if anything is scheduled to run that minute. If so, it executes the appropriate shell command. Then it goes back to sleep for another minute. *Cron* itself is normally started at boot time.

Some uses for Cron

- » Rotating log files (for more details about this, see the Log File Lumberjack feature on page 56 of **LXF92**, May 2007).
- » Summarising log files using tools such as *Logwatch* and *Webalizer*.
- » Automatically checking distro package repositories for updates.
- » Rebuilding the *slocate* database. (This is a database of all the filenames in the system, used to support the *slocate* command.)
- » Writing backups of the filesystem (just remember to load a tape into the machine before you go home!).
- » Weeding ancient files from directories such as **/tmp**.
- » Running intrusion detection scans using tools such as *Tripwire*.
- » Launching overnight software rebuilds.

We use *Cron* to schedule a variety of activities; some examples are listed in the Some Uses For Cron box, *below*. Most Linux distributions usually provide *Cron* preconfigured to perform some of these, but it is not hard to add your own, once you know how to drive it. Indeed, people get quite inventive about using *Cron*. A colleague of mine once wrote a script that gathered 'busyness' statistics from the machine. Using *Cron*, he ran the script on every machine on the network at 0, 15, 30 and 45 minutes past each hour and copied the results to a central machine. Another script, run by *Cron* at 5, 20, 35 and 50 minutes past each hour on the central machine, massaged the data further to produce HTML-formatted reports that the managers and engineers could look at. He also created bar charts showing how the load varied throughout the day.

Another colleague used *Cron* to download a weather satellite image from the Met Office every hour; he did this for two years then stitched the collection of images together into a movie. I know a manager who used *Cron* to turn off execute permission on all the games at 9:00 each morning and turn it back on again each evening, and a parent who used it to turn off internet access (with *iptables*) late each evening to prevent his kids staying up all night surfing, then turned it back on in the morning. All very god-like.

Cron is told what to do – and when to do it – by a series of configuration files called crontabs. First, there's **/etc/crontab**, sometimes called the system crontab. The anatomy of a single entry in this file is shown in the diagram *opposite*. In each of the five time fields you can put a:

- » *, meaning 'any value'.
- » Single value.

your Linux box

» Comma-separated list of values such as 15,25,40.
» Range such as 9-18.
» Period, usually used following *. For example, */5 in the minutes field means "every five minutes".

Here are a couple of examples:

```
*/10  *    * 5,7,11  *    root command1
 0    9-18  *  *      1-5  root command2
```

These lines will run **command1** every ten minutes during May, July and November, and **command2** once an hour during the working day (09:00 to 18:00) Monday to Friday. You can see that there's a lot of flexibility here! By the way, the sixth field in the crontab, which is root in our examples, specifies the user identity under which the script will run. You nearly always see root here, though a security-savvy administrator might apply the principle of least privilege and choose a less privileged identity unless the activity really does need root permission.

Although it's perfectly permissible to run your commands directly from the system crontab in this way, the current fashion among Linux distros is to interpose an intermediate script, usually called *run-parts*. For example, here's the system crontab from Red Hat and Fedora:

```
01 * * * * root run-parts /etc/cron.hourly
02 4 * * * root run-parts /etc/cron.daily
22 4 * * 0 root run-parts /etc/cron.weekly
42 4 1 * * root run-parts /etc/cron.monthly
```

The *run-parts* script is simple enough: it just runs all the executables in the specified directory. There are, as you see, four such directories, corresponding to four frequencies of execution. This scheme makes life more convenient, in the sense that you only have to drop your new script into the appropriate directory (depending on how often you want it to run) and *Cron* will automatically take care of running it, but you lose the flexibility to run your daily jobs at different times of the day, and everything runs as root. And of course this scheme can't handle operations such as the "disable the games in the morning and re-enable them in the evening" example mentioned earlier.

A glance at **/etc/crontab** in OpenSUSE 10.2 suggests that its authors have almost entirely abandoned the system crontab. There's just one line in **/etc/crontab** that runs the script */usr/lib/cron/run-crons* every 15 minutes. This script, which is undocumented and largely inscrutable, apparently takes over running the scripts in the **cron.hourly**, **cron.daily**, **cron.weekly** and **cron.monthly** directories, though the logic it applies to decide what to run and when is not obvious.

Cron for mortals

In addition to the system-wide **/etc/crontab**, *Vixie Cron* lets each user have their own crontab. These are stored in files named after the user's login under **/var/spool/cron/crontab** (the exact location varies between distros). These files have the same format as the system-wide *Cron* with one exception – there is no user ID column, because individual users are only allowed to run *Cron* jobs as themselves. You are not allowed to edit these files directly; you need to run **crontab -e**, which will open the file for you and drop you into the editor of your choice (actually, the editor specified by the **EDITOR** environment variable) to edit it. Why the **crontab -e** wrapper? Well, for one thing, it checks the syntax of your new entries before installing them, and for another, it jiggles the timestamp on the spool directory to tell *Cron* that one of the *Cron* files has been changed. (Unusually for a Linux daemon, there is no need to signal *Cron* after you've edited the config file, it will automatically notice the next time it wakes up.)

Administrators of multi-user systems can control who is allowed to use cron by placing user names in one of two files, **/etc/cron.allow** and **/etc/cron.deny**. The Can Fred Use Cron example in the box *overpage* shows the logic used to make this decision. This facility was probably more useful back in the days of true multi-user machines with many accounts; it is less relevant on machines that support only one or two users.

A cloudy project

One aspect of *Cron* I've ignored up to now is the environment that *Cron* jobs run in. A regular user's *Cron* jobs run with their current directory set to that user's home directory, and as we've already noted, they run with that user's identity. The number of environment variables defined will be minimal. *Cron* does not read a user's login scripts to establish user-specific settings, and in particular the search path contains only **/usr/bin** and **/bin**, so you need to take care when running commands from your crontab – in fact, it's common practice to refer to commands by their full path name to be sure they will be found.

It's also common practice to redirect the standard output and standard error of a *Cron* job to a file, so you'll often see entries in a crontab along these lines:

```
15 6 * * * /usr/bin/
someprog > /tmp/
someprog.out 2>&1
```

If a *Cron* job generates output that is not redirected, that output will be mailed to the job's owner.

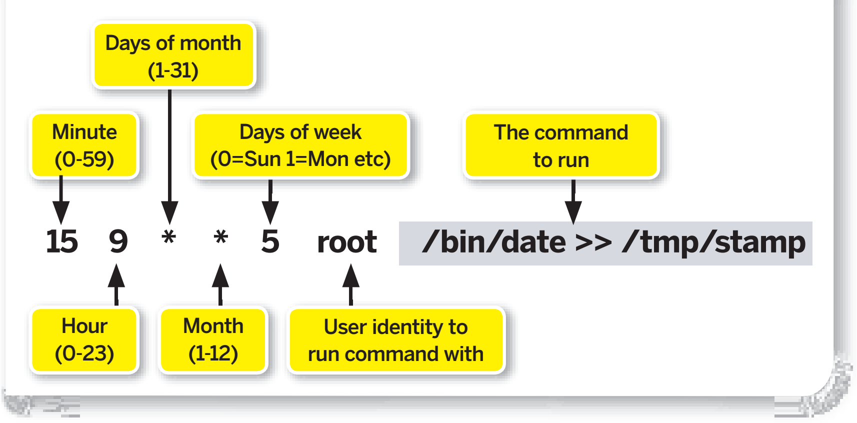
"Cron and Anacron arrange for commands to be run at a specific times."

You can define your own environment variables in your crontab if you want. For example, you could specify which shell will be used to interpret the command, and extend your search path with lines like this:

```
SHELL=/bin/bash
PATH=/usr/bin:/bin:/usr/local/bin
```

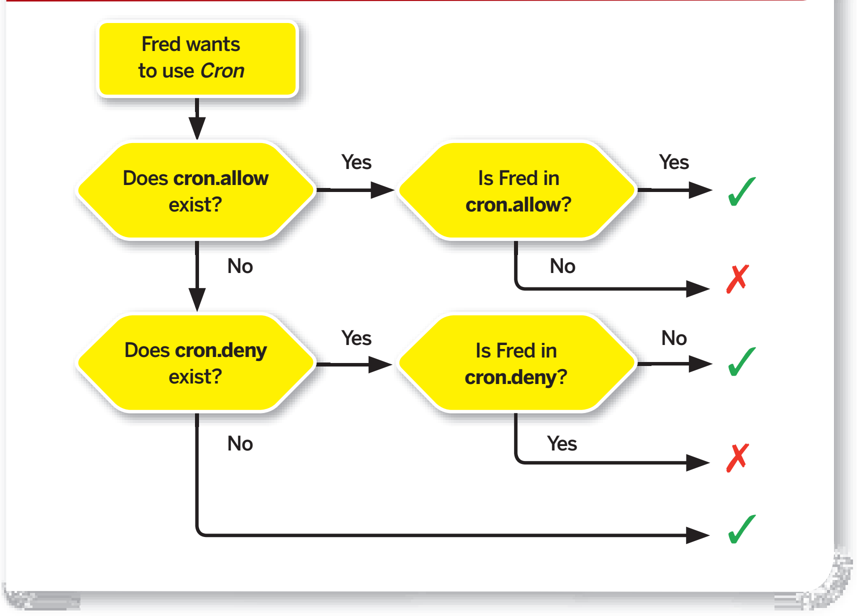
»

The anatomy of a crontab entry



Days of month (1-31)
Minute (0-59)
Days of week (0=Sun 1=Mon etc)
The command to run
15 9 * * 5 root /bin/date >> /tmp/stamp
Hour (0-23)
Month (1-12)
User identity to run command with

Can Fred use Cron?



» If, like me, you learn by experimentation, you will probably want to try out a few *Cron* entries of your own at this stage, but if you need some suggestions, try an experiment something like this:

1 Run **crontab -e**, and in your crontab file place the following:

```
FOO=BAR
PATH=/usr/bin:/bin:/usr/chris/bin
*/5 * * * * env >> env.out
*/10 * * * * echo "hello from a cron job"
30 * * * * wget http://metoffice.gov.uk/satpics/latest_ir.jpg
```

2 Verify that your crontab is correctly installed with the command **crontab -l**.

3 Find something else to do for an hour. Read the rest of the magazine, maybe?

4 At the end of the hour, verify that you have a file called **env.out** in your home directory. It should contain multiple identical copies of the *Cron* job's environment preceded by timestamps at five-minute intervals. Do you see the environment variables that you defined in your crontab?

5 Read your mail (by which I don't mean you to pull mail from your POP3 maildrop with *Evolution*, I mean you to run the *mail* command on the command line to examine your local mail file). Do you see the messages from the *Cron* job? They should appear every ten minutes.

6 Did you get the infrared satellite image? (If necessary, *wget* will automatically append a number to the filename to avoid overwriting the same file each time, so you may get names like **latest_ir.jpg.1**, **latest_ir.jpg.2** and so on.) Verify that England is entirely covered with thick cloud. If it is not, you may have retrieved an image of some other part of the world by mistake.

7 Unless you want to keep your crontab, delete it with the command **crontab -r**.

The first crontab entry in our experiment simply confirms the environment that a crontab job runs with. The second entry shows that if a crontab job generates output that is not redirected to a file, that output is mailed to the user. And the third entry demonstrates how to collect an infrared satellite image every hour. Stitching the image sequence together into a movie is left as an exercise to the reader, as they say in all the best text books.

Quick tip

Don't try to use *Cron* to run jobs at three in the morning if your computer is switched off at night – they won't happen. Use *Anacron* instead.

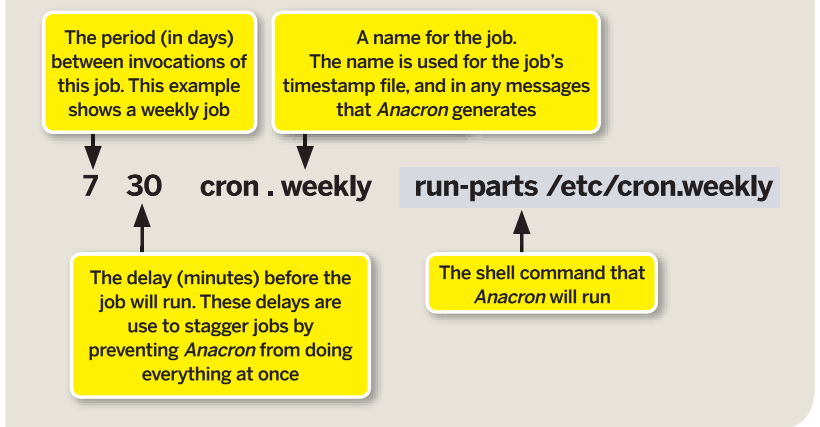
Part 2: Experiment with Anacron

Cron is great, but it has one weakness: it is designed for machines that are left running continuously. If your machine is turned off when a job falls due, it simply won't be run. So, if in the interests of reducing your carbon footprint you are careful to turn your computer off at night, those daily *Cron* jobs (which run at 04:02 on Red Hat/Fedora and 06:25 on Ubuntu) will never happen.

To handle this situation better, a new tool called *Anacron* was written. Returning to our language lesson for a moment, 'anachronistic' means 'chronologically misplaced'. For example, a documentary about Samuel Pepys that showed him writing his diary on a laptop might raise a few eyebrows. Anyway... *Anacron*'s mission is to guarantee that those daily, weekly and monthly activities do actually occur, assuming only that the computer was switched on at some point during the day (or week, or month). *Anacron* is not concerned with ensuring that activities occur at some precise time.

Anacron has a rather simple configuration file, **/etc/anacrontab**, whose syntax is shown in the diagram *below*. You'll see from this figure that the shortest period of time that *Anacron* concerns itself with is a day; consequently, *Anacron* is not a replacement for *Cron*. You cannot tell *Anacron* to run commands at a specific hour or minute, so, to return to one of my earlier *Cron* examples, you couldn't use Anacron to turn off internet access in the evening, and restore it the following morning. To keep track of what has been run, and when, *Anacron* writes timestamps to files in **/var/spool/anacron**. The files are named after the job ID in the anacrontab file. For the example shown in the figure, the timestamp file would be **/var/spool/anacron/cron.weekly**. The timestamp records only the day, month and year that the job was last run. It does not record hours, minutes or seconds.

The anatomy of anacrontab


The period (in days) between invocations of this job. This example shows a weekly job
A name for the job. The name is used for the job's timestamp file, and in any messages that *Anacron* generates
7 30 cron . weekly run-parts /etc/cron.weekly
The delay (minutes) before the job will run. These delays are use to stagger jobs by preventing *Anacron* from doing everything at once
The shell command that *Anacron* will run

Intelligent design

Now, *Anacron* is not a daemon in the usual sense of the word. That is, it does not start at boot time and lurk around waiting for something to do. Each time it is started, *Anacron* figures out what, if anything, needs to be run, waits for the specified delay to run it, then exits. So it is appropriate to ask: "When does *Anacron* actually get run?" A close look at the config files in Red Hat Enterprise Linux (also Fedora) reveals that *Anacron* is run at boot time, via the script */etc/init.d/anacron*. At this time, *Anacron* reads the anacrontab file, checks the timestamp files in **/var/spool/anacron** to see when each of the jobs was last run, and figures out what needs to get run.

For each of the tasks specified in anacrontab, *Anacron* waits for the specified delay, then runs the command. The delays are there to stop *Anacron* trying to do everything at once as soon as the machine has booted, at which time the user is, presumably, looking forward to getting some work done and would prefer not to have their machine fully occupied performing menial updates or, as Douglas Adams might have it, figuring out how to make a nice cup of tea.

Taken by itself, *Anacron* is not a difficult beast to understand. However, it gets more interesting when we try to figure out the interaction between *Anacron* and *Cron*, especially if the same activities are being managed by both. In the Red Hat/Fedora configuration, anacrontab looks like this:

```
1   65  cron.daily     run-parts /etc/cron.daily
7   70  cron.weekly    run-parts /etc/cron.weekly
30  75  cron.monthly   run-parts /etc/cron.monthly
```

From this, we see that *Anacron* is being asked to run precisely the same things as *Cron*; that is, at the appropriate interval (1, 7, or 30 days) it uses the *run-parts* script to run all the scripts in **/etc/cron.daily**, **/etc/cron.weekly** or **/etc/cron.monthly**. You'll notice that *Anacron* does not run the *cron.hourly* scripts. As I've said, it really isn't suitable for that, because it only records the timestamps of the most recently executed tasks to the nearest day.

Don't tread on my toes

When the same activities are being managed by both *Cron* and *Anacron*, there is a danger that *Cron* will run a job that *Anacron* has already run, or vice versa. Different distros use different tricks to prevent this. If you examine the Red Hat/Fedora configuration closely, you'll discover that the **cron.daily** directory includes a script called *0anacron* that runs *Anacron*, but with the **-u** option. So, we see that *Anacron* is run by *Cron* every day. However, the **-u** option tells *Anacron* merely to update its timestamps and not to actually run the commands. Essentially, *Cron* is saying to *Anacron* "please note that I have run these commands today (or this week, or this month ...) so you don't need to run them".

Ubuntu uses a different trick. Its system crontab looks something like this:

```
SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

17 *  * * *   root  run-parts /etc/cron.hourly
25 6  * * *   root  test -x /usr/sbin/anacron || run-parts /etc/cron.
daily
47 6  * * 7   root  test -x /usr/sbin/anacron || run-parts /etc/cron.
weekly
52 6  1 * *   root  test -x /usr/sbin/anacron || run-parts /etc/cron.
monthly
```

The **hourly** entry is straightforward; it looks more or less like Fedora's. The daily, weekly and monthly entries employ a cunning shell programming trick to ensure that *run-parts* is only run if *Anacron* is not present. (See The Magic Run-parts Trick, *above right*, for an explanation of how this works.) We can summarise the difference between the Red Hat and Ubuntu approaches by saying, "In Red Hat, *Cron* wins. In Ubuntu, *Anacron* wins."

The default configurations of *Cron* and *Anacron* provided by the various Linux distributions can be a little confusing, and there is, I suppose, a message to be delivered on top of all of this: you do not have to use *Cron* and *Anacron* in precisely the same way that your Linux distributor has opted to use them. If you understand the operation of these two services, and the syntax of their configuration files, you can make them do whatever you like.

The magic run-parts trick

The shell's logical OR operator, **||**, combines the true or false values of its two operands to yield a true/false result. So for example, **a || b** is true, if *a* is true or *b* is true. (That is to say, if either the command *a* or the command *b* returns zero, exit status.) Now – bear with me here, there is a point to this! – if *a* returns true, the value of **a || b** will be true regardless of the value of *b*. Yes? And in fact, in this case the command *b* won't even get run. This seemingly trivial aspect of the shell's behaviour is actually the whole point of what we see in Ubuntu's crontab. The shell command

```
test -x /usr/sbin/anacron || run-parts /etc/
cron.daily
```

will run two commands, *test* and (maybe) *run-parts*. The *test* command is actually built into the shell; **test -x** returns true if the specified file exists and is executable. So, what this line says is, "if **/usr/sbin/anacron** exists and is executable, then this entire expression is 'true' and there is no need to run *run-parts* to determine this", or to put it another way, "only run *run-parts* if *Anacron* is not present". *Quod erat demonstrandum*, as the Romans said.

Quick tip

Don't feel you have to stick with your distribution's pre-configured crontabs. Once you fully understand the format of these files, it's easy to make *Cron* do whatever you like, whenever you like.

Part 3: One-off tasks with at

We have just enough space to squeeze in a mention of the *at* command. Unlike *Cron* and *Anacron*, which schedule regularly repeated activities, *at* is used to schedule one-off activities in the future. Suppose you would like your computer to rebuild your Linux kernel this evening while you're at the pub. You might proceed like this:

```
$ at 20.30
warning: commands will be executed using /bin/sh
at> cd /usr/src/linux
at> make all
at> <EOT>
job 2 at Wed Feb 14 19:30:00 2007
```

As you'll see from this, *at* collects the commands you want to run from its standard input, up to an **EOT** character (^D by default). Your list of queued jobs can be examined using *atq*:

```
$ atq
2 Wed Feb 14 19:30:00 2007 a chris
$
```

The *at* daemon (*atd*) will start the job at the specified time, leaving you free to do something more interesting. Notice, however, that *at* will not actually arrange to get you to the pub at the appointed hour, nor will it offer to buy a round of drinks. You might, however, show it a little consideration and buy it a packet of crisps.

The details of the jobs you queue are placed in a spool directory (typically **/var/spool/at**). If you queue a job or two then examine the contents of this directory, you'll notice that, unlike *Cron*, *at* goes to some trouble to establish the same environment for the job that you'd have if you had just logged in. The daemon *atd* is responsible for actually running the jobs.

If you change your mind about an *at* job you previously submitted, you can delete a job from the queue using *atrm*:

```
$ atrm 2
$
```

The argument to *atrm* is the job number (**2** in this example).

The thing about *at* that fascinates me most is the wide range of notations it understands for specifying the time at which the command is to be run. The following are all valid:

```
$ at 14.30 tomorrow
$ at noon Friday
$ at 4pm + 3 days
```

My favourite is the use of the keyword **teatime** as a synonym for 4pm:

```
$ at teatime tomorrow
```

Marvellous. LXF