Insider Tips: *Cron, at, anacron*

The Scheduling Game

www.scx.hu

Unix systems need to handle a multitude of repetitive tasks: rotating logfiles, creating backups, updating indices, and managing databases. The traditional approach is to assign these tasks to *cron* and *at*. Another tool called *anacron* takes care of sleepy heads. BY MARC ANDRÉ SELIG

If you take a close look at the list of active processes on a Unix or Linux system, you will probably discover that it includes the *atd* and *cron* daemons (see Figure 1). Both daemons are used to schedule program launches. Admins need to tell *cron* what to do, at what time, and how often, although most Linux distributions provide ready-to-run *cron* jobs.

cron is particularly useful for time-consuming jobs that would stress even the most modern computer temporarily and thus might interfere with a user's everyday work. *locate* is a typical example. This program is used to find files more quickly. To do so it creates an index of all the files it discovers on a system.

If an index is to be useful, it needs to be updated regularly. And it is just as important to keep the *whatis* and *apropos* databases up to date. Depending on the configuration, these jobs can be extremely time-consuming and really interfere with a user's daily chores. So it makes sense for admins to run them during the night.

cron can also take care of important maintenance work that admins tend to forget. For example, admins use *cron* to rotate protocol files at regular intervals.

Fixed Schedule

There are any number of programs that can handle scheduled job execution, but *cron* is the great grandpa of them all, and many of the other scheduling tools are actually derived from the original *cron*.

cron is launched when the system boots and immediately starts looking for jobs to run. The daemon also checks if the schedule has changed. This means you can modify the schedule without relaunching *cron*. Figure 2 shows an example of a schedule, also known as a crontab. In typical Unix style, the lines that start with a pound sign, *#*, are comments.

Each job instruction comprises a time and the program to run. The time is subdivided into five fields for minute, hour, day, month, and weekday. Weekdays are simply numbered, with *0* standing for Sunday, but if you prefer to start the week on a Monday, you can use *7* for Sunday instead.

Don't worry if this is getting you confused, and you can't remember what order the fields are in: *man 5 crontab* will give you details of the crontab syntax. Even experienced sysadmins tend to consult this manpage before setting up a new job.

Instead of configuring static times, you can also specify a scope. For example, an asterisk, *, in a field means any value from the beginning to the end, for example, every minute. A value of * * * * * would mean running a program every minute of every hour every day. *37 * * * ** would launch the program 37 minutes after the full hour.

```
$ ps -ef | egrep '(cron|atd)'
daemon     320     1  0 Sep01 ?        00:00:00 /usr/sbin/atd
root       325     1  0 Sep01 ?        00:00:00 /usr/sbin/cron
$ █
```

Figure 1: The *atd* and *cron* daemons sit in the background waiting to perform scheduled tasks at predefined times.

```
$ cat /etc/crontab
# /etc/crontab: system-wide crontab
# Unlike any other crontab you don't have to run the `crontab'
# command to install the new version when you edit this file.
# This file also has a username field, that none of the other crontabs do.

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

# m h dom mon dow user  command
25 12   * * *   root    test -e /usr/sbin/anacron || run-parts --report /etc/cron.daily
47 12   * * 7   root    test -e /usr/sbin/anacron || run-parts --report /etc/cron.weekly
52 12   1 * *   root    test -e /usr/sbin/anacron || run-parts --report /etc/cron.monthly
#


$ █
```

Figure 2: The */etc/crontab* file contains the master crontab, which is typically provided by the distribution.

Incidentally, you might prefer to use unusual time values, such as 37 or 21, rather than zero. This reduces the risk of jobs running at the same time, and causing unnecessary load. The fields for day and weekday have a special feature: if both fields are set to a value (or range), *cron* will combine the values to create an either/or condition. *37 2 1 * 0* will thus launch a command at 02:37 on the first of the month, and every Sunday.

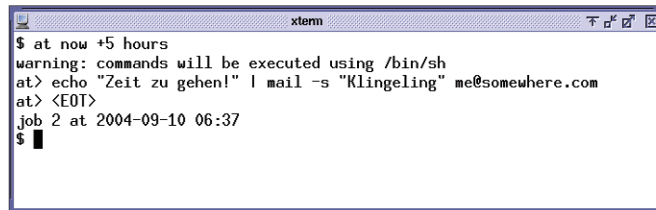
```
$ at now +5 hours
warning: commands will be executed using /bin/sh
at> echo "Zeit zu gehen!" | mail -s "Klingeling" me@somewhere.com
at> <EOT>
job 2 at 2004-09-10 06:37
$
```

Figure 3: The at program performs one-off jobs. In this case, a user wants *at* to send an email reminder in five hours time.

Your Own crontab

In addition to the global */etc/crontab*, every user can create their own crontab. A crontab file is installed onto the system using the *crontab* command. To create a crontab, type *crontab -e* (for "edit"). This will launch an editor and load your crontab in a temporary file.

After modifying the file, quit the editor, and *crontab* will copy the file to the *cron* spool directory. This is the directory that stores the contabs for local users (typically */var/spool/cron/crontabs/*). *crontab -l* outputs a job list on your screen. Users who prefer to store their crontab in their home directory need to point to the file to update the spool directory : *crontab* ~/filename.

Of course, the admin is responsible for updating the global crontab. As Figure 2 shows you, this file is typically quite small, and only calls programs from subdirectories. Most distributions store bash scripts or even binaries in the *cron.daily*, *cron.weekly*, and *cron.monthly* directories below */etc/*.

Assigning *cron* jobs to subdirectories helps to keep the master crontab readable. Each time an admin installs a new package that needs a *cron* job, your distribution's installation system will typically copy the relevant scripts to one of the *cron* directories. This removes the need to edit the master file.

The global crontab has an additional job field that stores the user ID with which *cron* should run the job. The field is directly behind the time specification.

Play it Once with *at*!

Whereas *cron* reads its crontabs to coordinate recurring tasks, *at* takes care of one-off jobs. *at* performs a job and then removes it from the list. This is useful, for example, if you need to send a message to remind someone to do something.

It only makes sense that *at* is an interactive program. The program expects you to specify the job time, although it is not too strict on the format. So to run a program at midnight, you can just say *at midnight*, or in five hours *at now +5 hours*. Of course, this does not mean that you can't be more precise if you need to be: *at 05:45 11 Jun 2005*.

After typing the *at* command, you are shown a prompt where you can enter shell commands. [Ctrl] + [D] takes you back to the normal shell, and tells *at* to save the job. Figure 3 shows an example, where *at* will be sending an email that says "Time to go!" with a subject line of "Ring ring" in five hours from now.

at handles job execution in an intelligent way. It "remembers" what shell the user was using (as this could affect the command syntax!), the current working directory, and the current environmental variables. In other words, users just need to type commands exactly as they would when running them in the shell.

Avoiding Load

Commands that need a graphical interface are exceptions to the rule, as *at* will not store the *$DISPLAY* variable. This said, there is an easy workaround to the display problem: instead of typing *xeyes*, simply type *DISPLAY=:0.0 xeyes*. If the server is running on a different display, you will need to modify the variable to reflect your setup. Mechanisms such as SSH agents will take effect as soon as a user logs out.

The *at* spool directory (to which only root has access) stores the jobs as simple shell scripts. If you do not have root privileges, typing *atq* will give you an overview of the outstanding jobs. *atrm* removes jobs from the queue.

Users wanting to start time-consuming jobs should add an ampersand (*&*) to the *at* prompt to tell the program to run in the background. This said, if 20 users have the same idea, the system load may reach a point where normal work becomes impossible. To handle this kind of scenario, there is a special version of *at* called *batch*.

Like *at*, *batch* accepts one or multiple commands. However, *batch* will only actually run the commands once the system load falls below a certain threshold value. This threshold is hard-coded into the *at* daemon, but admins can change it by launching *atd* with the *-l* value flag.

On legacy Unix systems, *at* jobs are launched by a crontab entry that runs the *atrun* program once a minute, but most current Linux distributions provide a genuine daemon called *atd* (see Figure 1).

anacron for Sleepy Heads

When *cron* and *at* were developed, programmers simply assumed that computers would be running 24x7. These programs date back to the days when Unix typically ran on heavy iron.

Home and office users are quite likely to switch their computers off, and laptops often power down to save batteries. *cron* and *at* jobs are disabled when a computer is down – and if the computer remains switched off for a significant time, important jobs just might never be performed.

anacron allows for more imprecise scheduling. The *anacron* application can handle scheduling such as "round about once a day."

When you boot your computer, *anacron* will check to see if a task needs to be performed and, if so, will launch the task. To prevent the computer taking ages to boot, *anacron* adds a delay, ensuring that each job will wait for the previous job to finish.

If a user who normally switches his or her computer off at night happens to leave it switched on instead, *anacron* will simply run the outstanding jobs at their normal times. And that takes some of the headaches out of system administration. ■