The journal of Apple technology.

Volume Number: 19 (2003)
Issue Number: 9
Column Tag: Programming

Section 7

Time-based Daemons

by Rich Morin

at(1), batch(1), cron(1), etc.

Most daemons (i.e., background processes) are event-based, responding to an incoming packet, the appearance of a file, etc. Some daemons are time-based, however, occurring at a given time (or times).

BSD (and thereby OSX) makes it very easy to set up time-based daemons. In fact, there are several ways to do this, depending on your needs. Let's look at some of the options.

CRON

The cron(8) subsystem (see also crontab(1,5)) runs commands at times which are specified in one or more control files.. It is also, as described below, the basis for time-based services such as at(1) and periodic(8).

Note: The command "man cron" fails in Mac OS X 10.2.6, but you can view the man page by typing "more /usr/share/man/cat8/cron.8.gz".

Originally, there was only one crontab(8) file, located at /etc/crontab. This file was only editable by root, though it could run commands as other users. Later, individual users were given the ability to maintain their own control files, using the crontab(1) command.

The default version of /etc/crontab on Mac OS X looks like:

```
# /etc/crontab
SHELL=/bin/sh
PATH=/etc:/bin:/sbin:/usr/bin:/usr/sbin
HOME=/var/log
#
#minute hour mday month wday who  command
#
#*/5    *    *    *     *    root /usr/libexec/atrun
#
# Run daily/weekly/monthly jobs.
15      3    *    *     *    root periodic daily
30      4    *    *     6    root periodic weekly
30      5    1    *     *    root periodic monthly
```

Environment variables can be set (using Bourne shell syntax) for the scheduled commands. Thus, the commands

listed in this file will have HOME, PATH, and SHELL set for them. Be sure to take these settings into account when writing scripts to be run under cron; if your script asks for a command that isn't found on the PATH, for example, it won't act as desired.

As in the case of most BSD control files (and many scripting languages), "#" can be used to indicate the start of a comment, extending through the end of the current line. This is often used to disable scheduling lines (such as the one for "atrun", in this example).

Each scheduling line has three parts, separated by white space. The first part may be a special string (e.g., @reboot, @daily), but more commonly it will be a set of five fields, also separated by white space. The second part is the username (e.g., root) under which the command will be run. The third part is the command (e.g., "periodic daily").

In the example, "periodic daily" is scheduled to be run (as root) at 3:15 AM every day. Similarly, "periodic weekly" and "periodic monthly" are scheduled for 4:30 AM each Saturday and 5:30 AM on the first day of each month, respectively.

The format of crontab files for individual users is almost identical to that for /etc/crontab. The only difference is that the "who" (username) field is not present. This makes sense; only the root account is able to set the user id under which a command will be run.

Although /etc/crontab can be edited in any desired manner, the individual crontab files must be edited by means of the crontab(1) command. This prevents race conditions, ensures that the cron daemon will notice any changes, etc.

Periodic

If you have a system maintenance command that needs to be run during off hours on a daily, weekly, or monthly basis, the periodic subsystem (periodic(8), periodic.conf(5)) may be exactly what you want.

The periodic(8) command is actually a shell script. I won't discuss it here, but you may wish to give it a look: "more /usr/sbin/periodic". Basically, however, the script runs every executable file found in the specified directory. For example, "periodic daily" runs any commands found in /etc/periodic/daily:

```
% wc -l /etc/periodic/daily/*
      56 /etc/periodic/daily/100.clean-logs
     131 /etc/periodic/daily/500.daily
     187 total
```

I wouldn't suggest modifying any of these files, as Apple may overwrite them in an update, but putting in your own files should be fairly safe. Just stuff an executable file into the appropriate subdirectory, picking the filename to sort into the desired execution order: For example, if you have a script that needs to run after "500.daily", you could name it something like "600.local.fooscript".

You may enjoy looking through these files to see what gets done while you're off snoozing: try "more /etc/periodic/*/*". The configuration files, described in periodic.conf(5), are also worth a look.

At, Batch, etc.

The at(1) and batch(1) commands act in a very similar manner to each other. Both commands schedule a file for execution at a specified time. The difference is that batch(1) also checks the system load level, ensuring that the command doesn't add work to an already-overloaded system.

In order to use either command, however, you'll have to uncomment the "atrun" line in /etc/crontab, causing the

program to be run every five minutes:

```
*/5     *     *     *      *     root /usr/libexec/atrun
```

Actually, there's no particular reason why you couldn't schedule atrun to run every minute, if you wish. On a desktop machine, an occasional process start-up is unlikely to make a noticeable difference. To try this, just edit the line to:

```
*       *     *     *      *     root /usr/libexec/atrun
```

Note: The documentation and configuration of at(1) in OSX 10.2.6 are a bit deficient. Although the at(1) man page says that "Traditional access control to at and batch via the files /var/at/at.allow and /var/at/at.deny is not implemented", the program will fail unless (at least) one of these files is present. The spool directory (/var/at/spool) may also be missing, causing scheduled jobs to silently fail. Fortunately, the fixes are simple:

```
% su
Password:
# touch /var/at/at.deny
# mkdir /var/at/spool
# exit
exit
%
```

Having worked our way past the setup hassles, let's try running some at(1) jobs. Here's a short test script we can use:

```
:
# att - at(1) test script
(date; printenv | sort) > att.$$.out
```

For the shell-challenged, here's a rundown of what's going on here. The initial colon tells the kernel that the script should be interpreted by the Bourne shell. The real work is done by a single line which starts up a subshell (subsidiary copy of the shell), has it run "date" and "printenv | sort", and redirects the (concatenated) output into a file.

Because "$$" evaluates to the process ID of the interpreting shell, the name of the file will look something like "att.12345.out". After you have edited the file, make it executable, run it, and examine the results:

```
% chmod +x att
% att
% more att.*.out
Fri Jul  4 18:46:38 PDT 2003
HOME=/Users/rdm
PATH=/Users/rdm/bin:...
PWD=/Users/rdm/...
SHELL=/bin/tcsh
...
```

The output shows us the date and time that the command was run, as well as the settings for any environment variables. Now, let's try scheduling the script via at(1), waiting for it to get run, and comparing the output with that of our first (manual) run:

```
% at -f att +1 minute
Job a010ce73c.000 will be executed using /bin/sh
```

```
% atq
Date                    Owner   Queue   Job#
19:04:00 07/04/03       rdm     a       a010ce73c.000
...
% atq
% _d_i_f_f_ _a_t_t_._*_._o_u_t_
__1_c_1_
_<_ _F_r_i_ _J_u_l_ _ _4_ _1_9_:_0_0_:_4_8_ _P_D_T_ _2_0_0_3_
_-_-_-_
_>_ _F_r_i_ _J_u_l_ _ _4_ _1_9_:_0_5_:_0_0_ _P_D_T_ _2_0_0_3_
_2_3_,_2_5_c_2_3_
_<_ _S_H_L_V_L_=_2_
_<_ _T_E_R_M_=_v_t_1_0_0_
_<_ _T_E_R_M_C_A_P_='"'"'"_
_-_-_-_
_>_ _S_H_L_V_L_=_1_
```

Not too many changes, really. The time changed, of course, but most of the environment variables stayed the same. SHLVL (the shell level) is lower for the at(1) run, because no interactive shell was involved. The TERM and TERMCAP variables aren't set for the at(1) run, because no terminal is attached to the process.

Rolling your own

If none of these facilities is quite what you need, consider creating your own time-based daemon. Simply putting a process to sleep for a specified period is quite simple; making a process wake up at a specified time is a bit trickier, but still quite possible.

If you take this approach, however, you may want to look at the source code for existing routines that perform similar services. The Darwin source code (www.opendarwin.org) has the source code for the commands described in this column. The CPAN (cpan.perl.org) is a good place to look for relevant Perl modules.

Rich Morin has been using computers since 1970, Unix since 1983, and Mac-based Unix since 1986 (when he helped Apple create A/UX 1.0). When he isn't writing this column, Rich runs Prime Time Freeware (www.ptf.com), a publisher of books and CD-ROMs for the Free and Open Source software community. Feel free to write to Rich at rdm@ptf.com.