Best practices for cron

Cron is a wonderful tool, and a standard part of all sysadmins toolkit. Not only does it allow for precise timing of unattended events, but it has a straightforward syntax, and by default emails all output. What follows are some best practices for writing crontabs I've learned over the years. In the following discussion, "cron" indicates the program itself, "crontab" indicates the file changed by "crontab -e", and "entry" begin a single timed action specified inside the crontab file. Cron best practices:

Version control

This rule is number one for a reason. Always version control everything you do. It provides an instant backup, accountability, easy rollbacks, and a history. Keeping your crontabs in version control is slightly more work than normal files, but all you have to do is pick a standard place for the file, then export it with crontab -l > crontab.postgres.txt. I prefer RCS for quick little version control jobs like this: no setup required, and everything is in one place. Just run: ci -l crontab.postgres.txt and you are done. The name of the file should be something like the example shown, indicating what it is (a crontab file), which one it is (belongs to the user "postgres"), and what format it is in (text).

You can even run another cronjob that compares the current crontab for each user with the version-controlled version and mail an alert and/or check it in automatically on a difference.

Keep it organized

The entries in your crontab should be in some sort of order. What order depends on your preferences and on the nature of your entries, but some options might include:

- Put the most important at the top.
- Put the ones that run more often at the top.
- Order by time they run.
- Order by job groups (e.g. all entries dealing with the mail system).

I generally like to combine the above entries, such that I'll put the entries that run the most often at the top. If two entries happen at the same frequency (e.g. once an hour), then I'll put the one that occurs first in the day (e.g. 00:00) at the top of the list. If all else is still equal, I order them by priority. Whatever you do, put a note at the top of your crontab explaining the system used in the current file.

Always test

It's very important to test out your final product. Cron entries have a nasty habit of working from the command line, but failing when called by cron, usually due to missing environment variables or path problems. Don't wait for the clock to roll around when adding or changing an entry—test it right away by making it fire 1–2 minutes into the future. Of course, this is only after you have tested it by creating a simple shell script and/or running it from the command line.

In addition to testing normal behavior, make sure you test all possible failure and success scenarios as well. If you have an entry that deletes all files older than a day in a certain directory, use the touch command to age some files and verify they get deleted. If your command only performs an action when a rare, hard-to-test criteria is met (such as a disk being 99% full), tweak the parameters so it will pass (such as setting the previous example to 5%).

Once it's all working, set the time to normal and revert any testing tweaks you made. You may want to make the output verbose as a final "live" test, and then make things quiet once it has run successfully.

Use scripts

Don't be afraid to call external scripts. Anything even slightly complex should not be in the crontab itself, but inside of an external script called by the crontab. Make sure you name the script something very descriptive, such as flush_older_iptables_rules.pl. While a script means another separate dependency to keep track of, it offers many advantages:

- The script can be run standalone outside of cron.
- Different crontabs call all share the same script.
- Concurrency and error handling is much easier.
- A script can filter output and write cleaner output to log files.

Use aliases

Use aliases (actually environment variables, but it's easier to call them aliases) at the top of your cron script to store any commands, files, directories, or other things that are used throughout your crontab. Anything that is complex or custom to your site/user/server is a good candidate to make an alias of. This has many advantages:

- The crontab file as a whole is easier to read.
- Entries are easier to read, and allow you to focus on the "meat" of the entry, not the repeated constants.
- Similar aliases grouped together allow for easier spotting of errors.
- Changes only need to be made in one place.
- It is easier to find and make changes.
- Entries can be more easily re-used and cut-n-pasted elsewhere.

Example:

```
PSQL_MASTER='/usr/local/bin/psql -X -A -q -t -d master'
PSQL_SLAVE='/usr/local/bin/psql -X -A -q -t -d master'

*/15 * * * * $PSQL_MASTER -c 'VACUUM pg_listener'
*/5 * * * * $PSQL_SLAVE -c 'VACUUM pg_listener' && $PSQL_SLAVE -c 'VACUUM
pg_class'
```

Forward emails

In addition to using non-root accounts whenever possible, it is also very important to make sure that someone is actively receiving emails for each account that has cronjobs. Email is the first line of defense for things going wrong with cron, but all too often I'll su into an account and find that it has 6000 messages, all of them from cron indicating that the same problem has been occurring over and over for weeks. Don't let this happen to you—learn about the problem the moment it stops happening by making sure the account is either actively checked, or set up a quick forward to one that is. If you don't want to get all the mail for the account, setup a quick filter—the output of cron is very standard and easy to filter.

Document everything

Heavily document your crontab file. The top line should indicate how the entries are organized, and perhaps have a line for $Id: cronrox.html,v 1.1 2008/12/08 13:41:31 greg Exp greg $, if your version control system uses that. Every entry should have at a minimum a comment directly above it explaining how often it runs, what it does, and why it is doing it. A lot of this may seem obvious and duplicated information, but it's invaluable. People not familiar with crontab's format may be reading it. People not as familiar as you with the flags to the "dd" command will appreciate a quick explanation. The goal is to have the crontab in such a state that your CEO (or anyone else on down) can read and understand what each entry is doing.

Avoid root

Whenever possible, use some other account than root for cron entries. Not only is is desirable in general to avoid using root, it should be avoided because:

- The root user probably already gets lots of email, so important cron output is more likely to be missed.
- Entries should belong to the account responsible for that service, so Nagios cleanup jobs should be in the Nagios user's crontab. If rights are needed, consider granting specific sudo permissions.
- Because root is a powerful account, its easier to break things or cause big problems with a simple typo.

Chain things together

When possible, chain items together using the && operator. Not only is this a good precondition test, but it allows you to control concurrency, and creates less processes than separate entries.

Consider these two examples:

```
## Example 1:
30 * * * * $PSQL -c 'VACUUM abc'
32 * * * * $PSQL -c 'ANALYZE abc'
```

```
32 * * * * $PSQL -c 'VACUUM def'
```

```
## Example 2:
30 * * * * $PSQL -c 'VACUUM abc' && $PSQL -c 'VACUUM def' && $PSQL -c
'ANALYZE abc'
```

The first example has many problems. First, it creates three separate cron processes. Second, the ANALYZE on table abc may end up running while the VACUUM is still going on—not a desired behavior. Third, the second VACUUM may start before the previous VACUUM or ANALYZE has finished. Fourth, if the database is down, there are three emailed error reports going out, and three errors in the Postgres logs.

The second example fixes all of these problems. The second VACUUM and the ANALYZE will not run until the previous actions are completed. Only a single cron process is spawned. If the first VACUUM encounters a problem (such as the database being down), the other two commands are not even attempted.

The only drawback is to make sure that you don't stick very important items at the end of a chain, where they may not run if a previous command does not successfully complete (or just takes too long to be useful). A better way around this is to put all the complex interactions into a single script, which can allow you to run later actions even if some of the previous ones failed, with whatever logic you want to control it all.

Avoid redirects to /dev/null

Resist strongly the urge to add 2>/dev/null to the end of your entries. The problem with such a redirect that it is a very crude tool that removes all error output, both the expected (what you are probably trying to filter out) and the unexpected (the stuff you probably do not want filtered out). Turning off the error output negates one of the strongest features of cron—emailing of output.

Rather than using 2>/dev/null or >/dev/null, make the actions quiet by default. Many commands take a -q, –quiet, or –silent option. Use Unix tools to filter out known noise. If all else fails, append the output to a logfile, so you can come back and look at things later when you realize your entry is not working the way you thought it was.

If all else fails, call an external script. It's well worth the extra few minutes to whip up a simple script that parses the error output and filters out the known noise. That way, the unknown noise (e.g. errors) are mailed out, as they should be.

Don't rely on email

Unfortunately, cron emails all output to you by default—both stdout and stderr. This means that the output tends to be overloaded—both informational messages and errors are sent. It's too easy for the error messages to get lost if you tend to to receive many informational cron messages. Even well-intentioned messages tend to cause problems over time, as you grow numb (for example) to the daily message showing you the output of a script that runs at 2 AM. After a while, you stop reading the body of the message, and then you mentally filter them away when you see them—too much mail to

read to look that one over. Unfortunately, that’s when your script fails and cron sends an error message that is not seen.

The best solution is to reserve cron emails for when things go wrong. Thus, an email from cron is a rare event and will very likely be noticed and taken care of. If you still need the output from stdout, you can append it to a logfile somewhere. A better way, but more complex, is to call an external script that can send you an email itself, thus allowing control of the subject line.

Avoid passwords

Don’t put passwords into your crontab. Not only is it a security risk (crontab itself, version control files, ps output), but it decentralizes the information. Use the standard mechanisms when possible. For Postgres connections, this means a .pgpass of pg_service.conf file. For ftp and others, the .netrc file. If all else fails, call a script to perform the action, and have it handle the passwords.

Use full paths

Both for safety and sanity, use the full paths to all commands. This is quite easy to do when using aliases, and allows you to also add standard flags as well (e.g. /usr/bin/psql -q -t -A -X). Of course, you can probably get away with not giving the full path to very standard commands such as “cp”—few sysadmins are that paranoid. :)

Conditionally run

Don’t run a command unless you have to. This also prevents errors from popping up. Generally, you only want to do this when you know there is a chance a command will not need to run, and you don’t care if it doesn’t in that case. For example, on a clustered system, test for a directory indicating that the node in question is active. You also want to account for the possibility that the previous cron entry of the same kind is still running. The simplest way to do this is with a custom PID file, perhaps in /var/run.

Use /etc/cron.* when appropriate

Consider using the system cron directories for what they were designed for: important system-wide items that run at a regular interval (cron.daily cron.hourly cron.monthly cron.weekly). Personally, I don’t use these: for one thing, it’s not possible to put them directly into version control.

Efficiency rarely matters

Don’t go overboard making your commands efficient and/or clever. Cronjobs run at most once a minute, so it’s usually better to be clearer and precise, rather than quick and short.

When in doubt, run more often

Don’t be afraid to run things more often than is strictly needed. Most of the jobs that crontab ends up doing are simple, inexpensive, and mundane. Yet they are also very important and sorely missed when not run. Rather than running something once a day because it only needs to be run once a day, consider running it twice a day. That way, if the job fails for some reason, it still has another chance to meet the minimum once a day criteria. This rule does not apply to all cronjobs, of course.

From:
https://wiki.plecko.hr/ - **Eureka Moment**

Permanent link:
https://wiki.plecko.hr/doku.php?id=linux:misc:cron:best_practices_for_cron

Last update: **2019/10/31 09:14**