Posted in Posted Python (../../../../categories/python/) Raspberry Pi (../../../../categories/raspberry-pi/) on 2013-07-21 13:05

Playing music on a Raspberry Pi using ... Previous (../playing-music-on-a-raspberry-pi-using-upnp-and-dlna-revisited/)

Morse Code Transcriber Next (../../12/morse-code-transcriber/)

Getting a Python script to run in the background (as a service) on boot

For some of my projects I write a simple service in Python and need it to start running in the background when the Raspberry Pi boots. Different Linux distributions use different ways of starting and stopping services (some now use Upstart (http://en.wikipedia.org/wiki/Upstart), some systemd (http://en.wikipedia.org/wiki/Systemd)). I am using the "Wheezy" Debian distribution on my Raspberry Pi, and in this case the proper way to do this is using an "init script". These are stored in the `/etc/init.d` folder. In there you can find scripts that for instance, start the networking system or a print server. Debian Wheezy uses the old Sys V init system which means that these scripts are run according to symbolic links in the `/etc/rc.x` directories. The Debian documentation (http://www.debuntu.org/how-to-managing-services-with-update-rc-d/) explains this.

Anyway, the following init script makes getting a Python script (or e.g. a Perl script) to run when the Raspberry Pi boots fairly painless. Services are supposed to run as "daemons" which is quite complicated in Python and involves forking the process twice and other (http://stackoverflow.com/questions/1603109/how-to-make-a-python-script-run-like-a-service-or-daemon-in-linux) nasty bits (http://code.activestate.com/recipes/278731/). Instead we can make use of the handy `start-stop-daemon` command to run our script in the background and basically deals with everything we need.

python-service/myservice.sh (../../../../listings/python-service/myservice.sh.html) (Source) (../../../../listings/python-service/myservice.sh)

```
1  #!/bin/sh
2
3  ### BEGIN INIT INFO
4  # Provides:          myservice
5  # Required-Start:    $remote_fs $syslog
6  # Required-Stop:     $remote_fs $syslog
7  # Default-Start:     2 3 4 5
8  # Default-Stop:      0 1 6
9  # Short-Description: Put a short description of the service here
10 # Description:       Put a long description of the service here
11 ### END INIT INFO
12
13 # Change the next 3 lines to suit where you install your script and what you want to call it
14 DIR=/usr/local/bin/myservice
15 DAEMON=$DIR/myservice.py
16 DAEMON_NAME=myservice
17
18 # Add any command line options for your daemon here
19 DAEMON_OPTS=""
20
21 # This next line determines what user the script runs as.
22 # Root generally not recommended but necessary if you are using the Raspberry Pi GPIO from Python.
23 DAEMON_USER=root
24
25 # The process ID of the script when it runs is stored here:
26 PIDFILE=/var/run/$DAEMON_NAME.pid
27
28 . /lib/lsb/init-functions
29
30 do_start () {
31     log_daemon_msg "Starting system $DAEMON_NAME daemon"
32     start-stop-daemon --start --background --pidfile $PIDFILE --make-pidfile --user $DAEMON_USER --chuid $DAEMON_USER --startas $DAEMON
33     log_end_msg $?
34 }
35 do_stop () {
36     log_daemon_msg "Stopping system $DAEMON_NAME daemon"
37     start-stop-daemon --stop --pidfile $PIDFILE --retry 10
38     log_end_msg $?
39 }
40
41 case "$1" in
42
43     start|stop)
44         do_${1}
45         ;;
46
47     restart|reload|force-reload)
48         do_stop
49         do_start
50         ;;
51
52     status)
53         status_of_proc "$DAEMON_NAME" "$DAEMON" && exit 0 || exit $?
54         ;;
55
56     *)
57         echo "Usage: /etc/init.d/$DAEMON_NAME {start|stop|restart|status}"
58         exit 1
59         ;;
60
61 esac
62 exit 0
```

Changing the init script

Lines 14 and 15 define where to find the Python script. In this case I have said that there is a folder `/usr/local/bin/myservice` and that the script is called myservice.py inside there. This is so that any additional Python files or other bits that your Python script needs can also be tidily put into that one place (not really how you're supposed to do it, but is easy).

Line 16 defines what we call the service. You should call this script by the same name.

Line 23 sets what user to run the script as. Using `root` is generally not a good idea but might be necessary if you need to access the GPIO pins (which I do). You might want to change this to the "pi" user for instance.

Line 28 loads a some useful functions from a standard file. We later use the logging functions for instance. We then define functions `do_start` and `do_stop` that will be used to start and stop the script.

`start-stop-daemon` needs to be able to identify the process belonging to a service so that (1) it can see it is there and does not start it again, and (2) it can find it and kill it when requested. In the case of a Python script then process name is "python" so this is not a very useful identifier as there may well be other Python processes running and things would get confusing. Instead we get `start-stop-daemon` to store the PID (the or process ID) using the `--pidfile $PIDFILE --make-pidfile` arguments. When told to start the process it looks for the file `$PIDFILE` which is defined in line 26 to be `/var/run/myservice.pid` (which on a Raspberry Pi is actually found at `/run/myservice.pid` thanks to a symbolic link.

Other than that, we use the `--background` flag of `start-stop-daemon` to run our script in the background, `--chuid` to set the user that the script runs as (with `--user` to look for scripts run by that user when we are trying to determine if it is already running) and `--startas` to define what we want to run. The options to `start-stop-daemon`

end with the double-hyphen and then we add on `$DAEMON_OPTS` in case there are any parameters to pass to the daemon itself.

When stopping the daemon the `--retry 10` means that first of all a TERM signal is sent to the process and then 10 seconds later it will check if the process is still there and if it is send a KILL signal (which definitely does the job).

Using the init script

To actually use this script, put your Python script where you want and make sure it is executable (e.g. `chmod 755 myservice.py`) and also starts with the line that tells the computer to use the Python interpreter (e.g. `#!/usr/bin/env python`). Edit the init script accordingly. Copy the init script into `/etc/init.d` using e.g. `sudo cp myservice.sh /etc/init.d`. Make sure the script is executable (`chmod` again) and make sure that it has UNIX line-endings (`dos2unix`).

To make the Raspberry Pi use your init script at the right time, one more step is required: running the command `sudo update-rc.d myservice.sh defaults`. This command adds in symbolic links to the `/etc/rc?.d` directories so that the init script is run at the default times. you can see these links if you do `ls -l /etc/rc?.d/*myservice.sh`

At this point you should be able to start your Python script using the command `sudo /etc/init.d/myservice.sh start`, check its status with the `/etc/init.d/myservice.sh status` argument and stop it with `sudo /etc/init.d/myservice.sh stop`.

An example service

If you run a Python script in this way then you don't get to see any output on the terminal so you need to do proper logging (rather than just `print` statements). The example Python service here shows how to parse command-line arguments and do simple logging to a file.

python-service/myservice.py (../../../../listings/python-service/myservice.py.html) (Source) (../../../../listings/python-service/myservice.py)

```
#!/usr/bin/env python

import logging
import logging.handlers
import argparse
import sys
import time  # this is only being used as part of the example

# Deafults
LOG_FILENAME = "/tmp/myservice.log"
LOG_LEVEL = logging.INFO  # Could be e.g. "DEBUG" or "WARNING"

# Define and parse command line arguments
parser = argparse.ArgumentParser(description="My simple Python service")
parser.add_argument("-l", "--log", help="file to write log to (default '" + LOG_FILENAME + "')")

# If the log file is specified on the command line then override the default
args = parser.parse_args()
if args.log:
        LOG_FILENAME = args.log

# Configure logging to log to a file, making a new file at midnight and keeping the last 3 day's data
# Give the logger a unique name (good practice)
logger = logging.getLogger(__name__)
# Set the log level to LOG_LEVEL
logger.setLevel(LOG_LEVEL)
# Make a handler that writes to a file, making a new file at midnight and keeping 3 backups
handler = logging.handlers.TimedRotatingFileHandler(LOG_FILENAME, when="midnight", backupCount=3)
# Format each log message like this
formatter = logging.Formatter('%(asctime)s %(levelname)-8s %(message)s')
# Attach the formatter to the handler
handler.setFormatter(formatter)
# Attach the handler to the logger
logger.addHandler(handler)

# Make a class we can use to capture stdout and sterr in the log
class MyLogger(object):
        def __init__(self, logger, level):
                """Needs a logger and a logger level."""
                self.logger = logger
                self.level = level

        def write(self, message):
                # Only log if there is a message (not just a new line)
                if message.rstrip() != "":
                        self.logger.log(self.level, message.rstrip())

# Replace stdout with logging to file at INFO level
sys.stdout = MyLogger(logger, logging.INFO)
# Replace stderr with logging to file at ERROR level
sys.stderr = MyLogger(logger, logging.ERROR)

i = 0

# Loop forever, doing something useful hopefully:
while True:
        logger.info("The counter is now " + str(i))
        print "This is a print"
        i += 1
        time.sleep(5)
        if i == 3:
                j = 1/0  # cause an exception to be thrown and the program to exit
```

By default it logs to a file in `/tmp` and at midnight will save the day's log file and start a new one, keeping 3 at most. To change where it logs to you need to use the `--log` or `-l` command line argument, so when running this from the init script you need to set the `$DAEMON_OPTS` variable. For instance, if you run the service as root then you could set `$DAEMON_OPTS="/var/log/myservice.log"` (the normal user cannot write files in there).

The example service above is heavily commented to explain what is going on, but one bit which is a little unusual is line 36-51 which I added to help people debug their services. Those lines set up a class called `MyLogger` which is initialised with the standard `logger` object just constructed along with a log level. The class only defines a `write` method which is all that is needed to emulate a normal stream of the standard output (or "stdout") or standard error (or "stderr") type. Normally when you do e.g. `print "hello"` in Python it actually does `sys.stdout.write("hello")` but line 49 changes this so that the stdout stream is replaced by an instance of `MyLogger` logging at the INFO level. Therefore, later in the example when the statement `print "This is a print"` is executed, that string actually goes into the log file. In the same way the standard error stream is replaced which means that when the program crashes because of the deliberate division by zero, the error and the traceback all appear in the log file at the ERROR level.

Please note that for both of these scripts to work you must make sure that the files have UNIX line-endings (just a LF) not DOS line-endings (CRLF). If you copy and paste from the web page into a Windows text editor and then transfer to a Linux machine (such as a Raspberry Pi) then they may end up with DOS line endings and will not work. If the shell script has DOS line-endings then if you run it using `./myservice.sh` you will see `-bash: ./myservice.sh: /bin/sh^M: bad interpreter: No such file or directory`. If the Python script has DOS line-endings and you run it using `./myservice.py` you will see `: No such file or directory`. You can fix this problem using the `dos2unix` command: just do e.g. `dos2unix myservice.py` (and `sudo apt-get install dos2unix` if you don't have the command). To avoid the problem in the first place you could copy the "raw" link from above and do e.g. `wget https://gist.github.com/scp93ch/cdb15468b84a8b3eb0aa/raw/myservice.py` on the Raspberry Pi to download it directly.

Troubleshooting

There are a lot of things to get right here. Here are some things to check (with thanks to David Selinger):

1. Make sure your Python file can execute stand-alone. Make sure you can run the command `sudo /usr/local/bin/myservice/myservice.py` (or whatever you have called it). If that doesn't work then check it is executable (with `ls -l`) and check it does not assume it is launched from a specific directory.
2. Try the `start-stop-daemon` command outside of the init script. You should be able to do `sudo start-stop-daemon ...` with all the parameters filled in.
3. Try writing a simple init script alternative that strips out most of the configuration and hard-codes the variables such as `$DIR` etc to make sure you have got that part right.
4. Test the complete init script by hand using `sudo /etc/init.d/myservice start` (though note that you must have done the `update-rc.d` command before this).
5. Test if it works when rebooting. If not, then you have a problem with the `update-rc.d` part and the `/etc/rc?.d` links.

Updates

- 2013-09-30: corrected mistake in use of —make-pidfile (thanks to Max Sistemich).
- 2014-06-29: added DAEMON_OPTS, more explanations and an example service with logging and arguments.
- 2014-07-07: added redirection of stdout and stderr.
- 2014-08-01: added instructions on sorting out the line-endings.
- 2016-04-13: update instructions regarding `update-rc.d` and added troubleshooting section.

Python (../../../../categories/python/) Raspberry Pi (../../../../categories/raspberry-pi/)

Playing music on a Raspberry Pi using ... Previous (../playing-music-on-a-raspberry-pi-using-upnp-and-dlna-revisited/)

Morse Code Transcriber Next (../../12/morse-code-transcriber/)

Comments

I am a husband, father and foster carer
a principal research engineer at the IT Innovation Centre
a salsa teacher
a Woodcraft Folk supporter
and in my spare time I write and maintain this web site

Cookies, Privacy and Licences (https://scphillips.com/privacy.html) | Donate (https://scphillips.com/donate.html)

(https://plus.google.com/+StephenPhillips1) (https://scphillips.com/contact.html)

(https://uk.linkedin.com/in/stephencphillips) (https://github.com/scp93ch)