Running Programs Automatically on Your Tiny Computer

Created by Brennen Bearnes

https://learn.adafruit.com/running-programs-automatically-on-your-tiny-computer

Last updated on 2021-11-15 06:31:09 PM EST
# Table of Contents

## Overview
- 3

## An Example Service: mouse.py
- 5

## SysV init: Runlevels
- 7
  - Runlevels and /etc/rc*.d/
  - 7
  - The SSH Init Script
  - 9

## SysV Init: Writing an Init Script for mouse.py
- 10
  - Copy and Modify the Example Init Script
  - 10
  - Test the Script and Install with update-rc.d
  - 12
  - Reboot
  - 13

## systemd: Writing and Enabling a Service
- 14
  - Exploring systemd Basics
  - 14
  - Writing a Unit File
  - 16

## Further Reading
- 17
  - sysvinit
  - 17
  - systemd
  - 17
Overview

For projects using a small GNU/Linux system, it’s often important that certain code always be running when the computer is on. You can always log in and start a script running by hand once you turn on the power, but this isn’t very reliable - after all, the power might flicker, or the system might crash for unrelated reasons. (There’s also a good chance you’ll be running a system without a monitor or keyboard attached, which can make things that much more complicated - it's much easier to have the right stuff run automatically.)

As an example scenario, I have a Raspberry Pi system in charge of logging data from my mousetrap (https://adafruit.it/iRA), and I’m about to go on a trip to another continent. The electricity where I live goes out now and then, and I’d like to make sure that my mouse-counting code comes back up on a reboot.
On an Arduino or similar device, the code you write usually just runs shortly after the power comes on, courtesy of a bit of software called a bootloader. Linux systems have these too, but what runs by default is the kernel, which in turn runs something known as an init system that's in charge of starting all sorts of other software to provide important services.

We'll look at how to use two different init systems which are both common in the wild.

First, we'll check out SysV-style init on a Raspberry Pi 2. With roots dating back to early Unix systems, sysvinit was widely used on most Linux distributions until recently, and is still used by the version of Raspbian released in May of 2015, based on Debian Wheezy. It's also similar to mechanisms still used by the BSD branch of the Unix family tree (https://adafru.it/evC).

Next, we'll look at systemd (https://adafru.it/fYe) on a BeagleBone Black running Debian Wheezy with systemd. systemd is "a suite of basic building blocks for a Linux system", recently adopted by most of the major distributions, including Debian Jessie (the current stable release of the project). Eventually, Raspbian will probably run systemd too.
An Example Service: mouse.py

My goal is to have the init system start a script called `mouse.py` every time the Pi boots to multi-user mode (which hopefully will also mean that the network is available).

If you're following along at home, there's a good chance you don't want to mess around with my entire mouse logging project right now, so I'll write a quick Python script for testing purposes (based on this StackOverflow answer by Nathan Jhaveri (https://adafru.it/fYC)).

You can paste or type the code into `~/mouse.py` using Nano and mark it executable:

```bash
nano /home/pi/mouse.py
```

```python
#!/usr/bin/python
import SocketServer
from BaseHTTPServer import BaseHTTPRequestHandler

class MyHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header("Content-type", "text/plain")
        self.end_headers()
        self.wfile.write('':3)~~~'

httpd = SocketServer.TCPServer('', 80, MyHandler)
httpd.serve_forever()
```

```bash
chmod +x ~/mouse.py
```

All this does is create a very simple web server that, when you visit the Raspberry Pi's network address, returns an ASCII mouse. To test, I'll check my Pi's IP address, start the script, and use `curl` to make sure it's running. (I'm using tmux to open more than one shell (https://adafru.it/jWB).)

Your IP address will likely be different, but in my case this looks like so:
If you're not on a network, you can just use `localhost` instead:

```
pi@raspberrypi ~ $ curl localhost
```

And if you want to test in a web browser, just enter the Pi's address in the URL bar:

Now that we have an example service, let's talk about how to start it.
SysV init: Runlevels

Runlevels and /etc/rc*.d/

The core idea of sysvinit is something called runlevels (https://adafruit.it/fYg), which are essentially just a way of organizing a collection of init scripts which have to run when the system starts or shuts down. Each runlevel corresponds to a directory in /etc/, which in turn contains symlinks to scripts in /etc/init.d/.

For historical reasons, there are a lot of runlevels, but a bunch of them aren't really used for anything special on Debian systems. You can read the full details of this in the Debian manual (https://adafruit.it/fYh), but the short version is that the system normally boots to runlevel 2, which is multi-user mode. Adapted from the manual, here's a partial table of levels:

<table>
<thead>
<tr>
<th>runlevel</th>
<th>directory</th>
<th>usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td></td>
<td>System boot.</td>
</tr>
<tr>
<td>0</td>
<td>/etc/rc0.d/</td>
<td>Halt the system.</td>
</tr>
<tr>
<td>1</td>
<td>/etc/rc1.d/</td>
<td>Single user mode (if you switch from multi-user mode).</td>
</tr>
</tbody>
</table>
Let's get a list of what's in runlevel 2:

```
cd /etc/rc2.d/
ls -l
```

`ls -l` gives a long listing of files, which will helpfully show you if a file is really a link to another file. Again, all of the actual init scripts turn out to live in `/etc/init.d`.

If a file in `/etc/rc*.d/` starts with `K`, it will be invoked by the init system with `[name of script] stop` (K is for kill), and if it starts with `S`, it will be invoked with `[name of script] start`.

Init scripts usually accept (most of) the following command-line parameters:

- **start** - start the service
- **stop** - stop the service
- **status** - check whether the service is running
- **reload** - have the service reload configuration files
- **restart** - stop and start the service
So, for example, you could say `/etc/init.d/ssh restart` to stop and start the SSH service. You often see commands like this in tutorials and HOWTOs.

The SSH Init Script

As an example, load up `/etc/init.d/ssh` in a text editor and give it a look.

```bash
cd /etc/init.d
nano ssh
```

This is actually quite a few lines of code for what seems like a pretty simple task - it just needs to start or stop a program, right? It turns out, though, that there can be a lot of considerations involved in doing that. For example, these rather squirrelly-looking lines...

```bash
test -x /usr/sbin/sshd || exit 0
( /usr/sbin/sshd -? 2>&1 | grep -q OpenSSH ) 2>/dev/null || exit 0
```

...make sure that `sshd` exists, is executable, and at least claims to be the SSH daemon provided by the OpenSSH project. The rest of the script defines functions for all sorts of housekeeping and sanity checks before it gets to the part where it handles the start/stop/etc. commands in a case statement:

```bash
case "$1" in
  start)
    check_privsep_dir
    check_for_no_start
    check_dev_null
    log_daemon_msg "Starting OpenBSD Secure Shell server" "sshd" || true
    if start-stop-daemon --start --quiet --oknodo --pidfile /var/run/sshd.pid --exec /usr/sbin/sshd -- $SSHD_OPTS; then
      log_end_msg 0 || true
    else
      log_end_msg 1 "Starting OpenBSD Secure Shell server" "sshd"
```
For one-off projects on the Pi, you almost certainly don't need to exercise this level of caution, but you will want to handle the common commands.

It would be a pain to write a lot of this stuff out from scratch. Fortunately, we have `/etc/init.d/skeleton` to work with. This just provides the "bones" of an init script, and should work with a few minor changes. Next, we'll look at adapting it to run our example `mouse.py`.

---

### SysV Init: Writing an Init Script for mouse.py

#### Copy and Modify the Example Init Script

Start by copying `/etc/init.d/skeleton` to `/etc/init.d/mouselogger`, and marking it executable:

```bash
cd /etc/init.d
sudo cp skeleton mouselogger
sudo chmod +x mouselogger
```

We need the `sudo` here because (like most things in `/etc`) files in `/etc/init.d` are owned by root. (You can imagine what might happen on a system with lots of users if just anyone could muck around in the startup files.)

Next, open this up in Nano:

```
nano mouselogger
```
I had to change three things. First, the metadata between `BEGIN INIT INFO` and `END INIT INFO` to specifically reference a service called "mouselogger" and provide a description:

```
#!/bin/sh
### BEGIN INIT INFO
# Provides:               mouselogger
# Required-Start:         $remote_fs $syslog
# Required-Stop:          $remote_fs $syslog
# Default-Start:          2 3 4 5
# Default-Stop:           0 1 6
# Short-Description:      Adafruit-MouseLogger
# Description:            Log mice.
### END INIT INFO
```

Second, the environment variables that the rest of the script uses to do its business. I added `/home/pi` to the path, changed the description to "Mouse logger", and changed the name to `mouse.py`, as well as emptying the `DAEMON_ARGS` variable:

```
# PATH should only include /usr/* if it runs after the mountnfs.sh script
PATH=/sbin:/usr/sbin:/bin:/usr/bin:/home/pi
DESC="Mouse logger."
NAME=mouse.py
DAEMON=/home/pi/$NAME
DAEMON_ARGS=""
PIDFILE=/var/run/$NAME.pid
SCRIPTNAME=/etc/init.d/$NAME
```

And lastly, I added `--background` and `--make-pidfile` to the `start-stop-daemon` options in the `do_start()` function:
This ensures that the script is sent to the background instead of running in the foreground after it starts, and creates a file in `/var/run` that contains the process ID of the script.

Here's a copy of the entire file (https://adafru.it/fYi).

**Test the Script and Install with update-rc.d**

You should now be able to run `/etc/init.d/mouselogger` - but first, make sure that `/home/pi/mouse.py` isn't already running. If you have it open in a terminal, you can type Ctrl-c to kill the process. Otherwise, use `ps aux | grep mouse.py` to see if it has a running process:

```
ps aux | grep mouse.py
```

Here, I checked with `ps` and found a copy of the Python interpreter running `mouse.py`, so I killed the corresponding process with `sudo kill 2052`. (2052 is the process id in the second column of the output from `ps`.)

Now you can try starting and stopping the service and testing it out:

```
sudo /etc/init.d/mouselogger start
ps aux | grep mouse.py
curl localhost
```

```
sudo /etc/init.d/mouselogger stop
```

©Adafruit Industries

Page 12 of 17
Once you’re confident that the init script works, you can install it in the default runlevels so that it starts at boot time:

```
sudo update-rc.d mouselogger defaults
```

Reboot

And now, the moment of truth! Reboot and see what happens...

```
sudo reboot
```

This part might take a bit, but once you can log back in, it should be simple to test whether `mouse.py` is running:
Success!

Next up, we’ll look at doing the equivalent thing with systemd on a BeagleBone Black.

systemd: Writing and Enabling a Service

This section uses a BeagleBone Black Rev C, running a Debian Wheezy release from July 2015.

We have a whole category of BeagleBone tutorials (https://adafru.it/jWC). I found the specific image I installed on the elinux BeagleBoneBlack Debian page (https://adafru.it/fYj). Recent editions of the BeagleBone likely come with something very similar installed.

This section assumes that you are logged into your BBB as root.

Exploring systemd Basics

systemd takes a very different approach from the sysvinit scheme of organizing init scripts into directories by runlevel. Instead, it uses unit files to describe services that should run, along with other elements of the system configuration. These are organized into named targets, like multi-user.target and graphical.target.

You can list targets on the current system like so:

```
sudo systemctl list-units --type=target
```

```
root@beaglebone:/etc/systemd/system# systemctl list-units --type=target
UNIT LOAD ACTIVE SUB JOB DESCRIPTION
basic.target loaded active active Basic System
cryptsetup.target loaded active active Encrypted Volumes
getty.target loaded active active Login Prompts
graphical.target loaded active active Graphical Interface
local-fs-pre.target loaded active active Local File Systems (Pre)
local-fs.target loaded active active Local File Systems
multi-user.target loaded active active Multi-User
remote-fs.target loaded active active Remote File Systems
sockets.target loaded active active Sockets
sound.target loaded active active Sound Card
swap.target loaded active active Swap
sshd.target loaded active active System Initialization
syslog.target loaded active active Syslog

LOAD = Reflects whether the unit definition was properly loaded.
ACTIVE = The high-level unit activation state, i.e. generalization of SUB.
SUB = The low-level unit activation state, values depend on unit type.
JOB = Pending job for the unit.

13 units listed. Pass --all to see inactive units, too.
```

©Adafruit Industries
Just running `systemctl` by itself will show you a list of all active units, including configuration for devices, services, filesystem mountpoints, timers, and sockets.

Mostly, systemd units are defined in various files in directories like `/lib/systemd/system`.

If you want to know the status of an existing service, you can use something like `systemctl status foo.service` - for example, let's check up on SSH:

```
sudo systemctl status ssh.service
```

You might notice something a bit odd here - this status report still mentions `/etc/init.d/ssh`. Isn't that a sysvinit thing? Well, yeah. What is happening is that the system we're on is still a hybrid between old-school and new-school init systems. systemd itself is actually capable of acting as a replacement (mostly) for sysvinit.

This all looks a little daunting, and there are a lot of concepts to untangle if you want to understand everything that's going on (your humble narrator is still quite a ways from this), but just configuring a new service is very little work.
Writing a Unit File

Let's reuse our example service, `mouse.py`. Run `sudo nano /root/mouse.py`, and paste the following code (notice that I've changed the port from 80 to 8888, so as not to step on the toes of other services already running on the BeagleBone):

```python
#!/usr/bin/python
import SocketServer
from BaseHTTPServer import BaseHTTPRequestHandler
class MyHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header("Content-type", "text/plain")
        self.end_headers()
        self.wfile.write('&lt;:3)~~~
')
httpd = SocketServer.TCPServer(('', 8888), MyHandler)
httpd.serve_forever()
```

Then make sure this is executable with `sudo chmod +x /root/mouse.py`. Next, create `/lib/systemd/system/mouselogger.service` and use Nano to paste the following:

```
[Unit]
Description=Mouse Logging Service

[Service]
ExecStart=/root/mouse.py
StandardOutput=null

[Install]
WantedBy=multi-user.target
Alias=mouselogger.service
```

Now you can enable your new service:

```
sudo systemctl enable mouselogger.service
sudo systemctl start mouselogger.service
```

Did it work?
curl localhost:8888

raspberry_pi_screencast-2015-08-31-18_20_31.gif

Sure enough! As a last step, you should probably reboot and repeat this test to make sure.

---

**Further Reading**

sysvinit

- [The Debian Manual - The system initialization](https://adafru.it/fYh)
- [Wikipedia: Init](https://adafru.it/fYk)
- [start-stop-daemon: --exec vs. --startas](https://adafru.it/fYl)
- [Getting a Python script to run in the background (as a service) on boot](https://adafru.it/fYq)

systemd

- [Getting Started with systemd on Debian Jessie](https://adafru.it/fYm)
- [systemd for Administrators](https://adafru.it/fYn)
- [How To Use Systemctl to Manage Systemd Services and Units](https://adafru.it/fYo)