Part 15
-
Background, Boot Time, Timed, Daemons & Services

Version: 2020-10-08

Multitasking from the Command Line

Did you know that you aren't limited to working on one thing at a time while on a Linux command line? You can actually "minimize" a program that you are in, get back to the command line, and then return to the program whenever you'd like.
When you run a program or script on the Linux command line (from now on referred to as the shell), you are creating a new job. For those that are used to GUI environments, each of these jobs is somewhat like a window on the desktop. Just as you can have multiple windows and switch between them, the shell is capable of managing multiple jobs and allows you to switch between them.
There is a lot to cover, so let's start simple by describing what states a shell job can be in.

Foreground Jobs
Up to now, you may have only worked with foreground jobs in the shell. Foreground jobs are ones that don't return you to the command prompt until you exit out of the job or the job finishes.
If you run "ping yahoo.com" from the shell, the ping process will become a foreground job and will not return you to the command prompt until you stop it.
Just in case you didn't know, you can stop the ping process by pressing [Ctrl]+c. Unlike in many applications that you may be used to, [Ctrl]+c does not mean "copy" in a Linux shell. Instead of copying text, this signals the foreground process to terminate by sending it the SIGINT (Signal Interrupt) signal.
An example of a foreground job that won't return you to the command prompt until it is finished is the zip command. Imagine that you have a large amount of files inside a folder called "stuff". If I run "zip -r stuff.zip stuff" and don't know about job management, I won't have access to the shell until zip finishes its task and returns me to the prompt.

Background Jobs
Putting a job in the background allows the job to run exactly as if it were in the foreground except that it does not receive user input. If a command produces output and is put in the background, it will still produce the output.
Why would putting a job in the background be helpful then you may wonder? There are a number of reasons, and they will become clear in the examples below.

Stopped Jobs
There is technically a third state that a job can be in. Jobs that aren't in the foreground yet are also not in the background have been stopped.
Like background jobs, stopped jobs do not receive user input. Unlike background jobs, stopped jobs do not produce output. As a matter of fact, they don't do anything. They have essentially been put on hold and won't do any processing until they are resumed or killed.

Multitasking 1: Stopping and Resuming Processes
It may seem like stopped jobs are the odd man out, but they are actually the key to getting started with multitasking on the Linux shell. I use stopped jobs every day, and I couldn't imagine using the shell without them.
Imagine that you are in your favorite shell text editor working on a program or modifying config files, and you can't remember what the IP address you need is but you could easily look it up with dig. You could easily open up another shell connection and run the command, but it would be quicker to simply stop the editor, run the command quickly, copy the text, and bring your editor to the foreground.
The following is a sequence that illustrates how this example would look on your shell.

```
$ sudo vi /etc/httpd/conf/httpd.conf
[sudo] password for chris:
[Ctrl]+Z
[1]+  Stopped                 sudo vi /etc/httpd/conf/httpd.conf
$ dig +short chrisjean.com
96.125.165.12
$ fg %1
```

After running these commands, you will be back in your editor.
The key to stopping a running job is the `[Ctrl]+z` key combination. Again, some of you may be used to `[Ctrl]+z` as the shortcut to undo, but in the Linux shell, `[Ctrl]+z` sends the `SIGTSTP` (Signal Tty SToP) signal to the foreground job. When you press this key combination, the running program will be stopped and you will be returned to the command prompt.
You may notice the `%1` after `fg`. The `1` is the job number that is to be brought to the foreground. The job number is listed in brackets when the job was stopped. These numbers are crucial when you have multiple jobs in stopped states. At any time, you can run `jobs` to get a list of the current jobs.

Multitasking 2: Running Jobs in the Background
Stopping jobs is great, but what if we want to be returned to the command prompt while still allowing the job to run? Sending a job to the background will do exactly that.
Assume that the updatedb command takes a very long time to run (more than an hour).
I'll use the example of the updatedb command to show how you can take a foreground process and put it in the background so that you can free up the shell.

```
$ sudo updatedb
[sudo] password for chris:
[Ctrl]+z
[1]+  Stopped                 sudo updatedb
$ bg %1
[1]+ sudo updatedb &
```

There are a few items of note here.

1. I'm back at the command prompt and can do anything I'd like while the job works merrily away in the background.
2. Notice how I used `%1` with bg just as I did with `fg`. Both commands accept a job number to tell it which job is to be sent to the foreground/background.
3. After running `bg`, a line similar to when I pressed `[Ctrl]+z` was printed showing the command and job number. However, there is also the addition of the `&` at the end. We'll get into the reason for that in the next section.

When a process that is running in background finishes, you will receive a message the next time your command prompt refreshes. The prompt refreshes when you run another command or simply hit `Enter` to get a new prompt.
Continuing from the previous example, I receive output similar to the following when the updatedb command finishes:

```
[1]+  Done                    sudo updatedb
```

As with all the other job-related output, you will be notified of the job number and the command.

Multitasking 3: Starting Jobs in the Background
Now that you see the value of having jobs run in the background, you might wonder if you can just start the command in the background to begin with. In fact, you can!
Remember that odd ampersand `&` after the command in the output of the `bg` command? The ampersand means that the job is running in the background. You can run a command with the ampersand at the end to automatically put the job in the background.
Going back to the updatedb example again, we could have simply run the following to have the job in the background and get our command prompt back immediately:

```
$ sudo -v
[sudo] password for chris:
$ sudo updatedb &
[1] 15231
```

You might wonder what that completely unnecessary `sudo -v` command is all about. Remember that when you send a job to the background, it cannot receive input from the user. Since `sudo` prompts for the password when it hasn't been run recently, I ran that command before running the background `sudo` command to ensure that the password wouldn't be prompted for. If you do send a sudo command to the background and `sudo` prompts for a password, you need to bring it to the foreground first to enter the password and then send it to the background again.
The output when sending the job directly to the background is different than running `bg`. Instead of telling you the command, it tells you the actual process ID. Every program that runs in Linux is a process and has a process ID. You can use these process IDs to kill a specific process, change the process' execution priority, and isolate the resource utilization by that process.
Even though you've put a task in the background, you can still bring it to the foreground using the `fg` command. In the previous example, I can pull the background process to the foreground by running `fg %1`.

Multitasking 4: Killing and Prioritizing Jobs
Now that you know the fundamentals of managing jobs, it's time to get more advanced and tell you how to terminate jobs and modify their execution priority.

Terminating Jobs
Sometimes a process goes rogue and either won't respond to input, is consuming massive amounts of resources, or both. At times like this, it's very important to know how to kill a job which will cease the program's execution and free up its used resources.
The command used to kill processes is appropriately called kill. The kill command can accept either process IDs or job numbers.
If my updatedb job from before fails to respond, the job has a job number of `1`, and I want to tell the job to stop, I could run the following command:

```
$ sudo kill %1
[sudo] password for chris:
```

Notice that there isn't any confirmation that the `kill` command was successful. You can check to ensure that the job has been terminated by running the `jobs` command and looking for that job's listing.
By default, the `kill` command sends a `SIGTERM` signal to the process; however, sometimes this is not enough to stop the execution of an out of control process. To shut down these processes, we need something more powerful.

```
$ sudo kill -9 %1
[sudo] password for chris:
```

This may not look much different, but the addition of the `-9` option tells the command that things are serious now. This option causes a `SIGKILL` (Signal Kill) signal to be sent to the process. A `kill` signal cannot be intercepted by a process and causes the process to

immediately terminate without allowing it to clean up after itself. It's for this reason that you should only use `SIGKILL` after first sending the process a `SIGTERM`.
As mentioned before, you can also run a `kill` command on a process ID rather than a job number. If we wished to kill a process with a process ID of 12267, we can run the following command:

```
$ sudo kill 12267
```

Note that you don't have to always run the `kill` command with `sudo`, you just have to ensure that whatever privilege level you are running the `kill` command at has the ability to control that process. In other words, if you are trying to `kill` a process run with sudo, you'll need to use sudo to `kill` it as well.

Prioritizing Jobs
Linux does a great job of prioritizing tasks automatically; however, what Linux thinks is best isn't always what you want. Basically, Linux always wants to make a process finish as soon as possible, which means that you get a nice bit of lag when you try to do your work as a CPU or disk hungry process churns away. Fortunately, there are tools that will let you tell Linux what you want.
First, let's deal with managing a process' use of CPU time. If you are going to run a command that will eat up large amounts of CPU cycles and you want to make sure that this process doesn't keep you from doing your work, you can run the command with nice to give the command a lower-than-default priority.

```
$ sudo -v
[sudo] password for chris:
$ sudo nice updatedb &
[1] 16312
```

As before, a junk `sudo` command was run first to take care of the password prompt. Running the command in nice lowers the processes CPU priority and will result in smoother CPU performance for your other applications.
What if you already ran a command whose process is now greedily chewing away at your CPU resources? That's where the renice command comes into play.
With renice, you can change the CPU priority of an existing process.

```
$ sudo -v
[sudo] password for chris:
$ sudo updatedb &
[1] 16324
$ sudo renice 10 16324
16324: old priority 0, new priority 10
```

The renice command accepts a priority as the first argument and the process ID as the second argument. The priority argument has a range from -20 to 19 where -20 is the highest priority and 19 is the lowest. In other words, a higher numeric value priority results in lower CPU priority.
That takes care of the CPU, but admittedly, the updatedb command does more damage to the IO or disk traffic than it does to the processor. How can we reign in a process' use of the disk so that performance for the rest of the system improves? For that, we use the ionice command.
With the ionice command, you can set the IO priority for a process to one of three classes: Idle (3), Best Effort (2), and Real Time (1). The Idle class means that the process will only be able to read and write to the disk when all other processes are not using the disk. The Best Effort class is the default and has eight different priority levels from 0 (top priority) to 7 (lowest priority). The Real Time class results in the process having first access to the disk irregardless of other process and should never be used unless you know what you are doing.

If we wish to run the updatedb process in the background with an Idle IO class priority, we can run the following:

```
$ sudo -v
[sudo] password for chris:
$ sudo updatedb &
[1] 16324
$ sudo ionice -c3 -p16324
```

If we'd rather just lower the Best Effort class priority (defaults to 4) for the command so the process isn't limited to idle IO periods, we can run the following:

```
$ sudo -v
[sudo] password for chris:
$ sudo updatedb &
[1] 16324
$ sudo ionice -c2 -n7 -p16324
```

Again, the Real Time class should not be used as it can prevent you from being able to interact with your system. You may wonder where you can get the process ID if you don't know it, can't remember it, or didn't start the process (an automated script may have launched it). You can find process IDs with the ps command. For example, if I had an updatedb program running in the background, and I wanted to find its process ID, I can run the following:

```
$ ps -C updatedb
PID TTY          TIME CMD
4234 ?        00:00:42 updatedb
```

This tells me that the process' process ID (PID) is 4234.
Another useful command is `ps -u user` (replace `user` with your username) which lists all processes your user has access to. Basically, these are processes that your user or some process using your user's permissions started.
Another really helpful use of `ps` is `ps -aux`. This command will list all the system's processes, which user owns the process, and even detail the resource usage of the process. This is very helpful for finding processes that you don't know enough about to find using other methods.

Mutitasking 5: Using Job Management in Scripts
Keep in mind that many scripts execute from inside shell environments. This means that you can manage the script's job using the tools I've given you here.

If you look in the `/etc/cron.daily` folder, you'll find many scripts that execute once a day. These scripts can create a huge load on a system while they execute. If around the clock performance is crucial, you can prioritize the shells that the scripts run in which causes all the commands run inside that shell to follow the same prioritization rules.
This all sounds very complex, but it's actually very simple.
Let's open up the `/etc/cron.daily/mlocate` file (pick another file if you don't have this one). At the top, you should see a line like the following:

```
#!/bin/bash
```

This shebang line tells Linux what interpreter is used to execute the script. My example script uses the bash shell as the interpreter. Other scripts may use the sh shell instead of the bash shell, but that won't make any difference. Below the first line, put the following:

```
renice 10 $$
ionice -c3 -p$$
```

The `$$` variable is a special variable that contains the shell's own process ID. The two commands above cause the CPU priority to lower to 10 and changes the IO priority to class 3 (Idle).
Add this type of customization to your resource hungry scripts, and your system will perform more smoothly around the clock. Keep in mind that this also can result in the scripts taking much longer to complete.
You can use those same commands above using the $$ variable to change the CPU and IO priority of shells that you open in order to lower the impact you have while working on the system. This can be a smart thing to do if you need to compile software or run other intensive processes on production machines.

Important Commands
Here's a list of all the commands, key combos, and other important bits that I went over with a brief description for each. It's like the word list at the end of children's books.

- `fg` Brings a stopped or background job to the foreground
- `bg` Sends a stopped job to the background
- `jobs` Lists the current jobs and their corresponding job number
- `[Ctrl]+c` Key combination that sends a SIGINT signal to the foreground process. Use this to close processes that don't stop (such as ping or top) or terminate a process that no longer responds or is no longer needed.
- `[Ctrl]+z` Key combination that sends the SIGTSTP signal to the foreground process. Use this to send a foreground process to a stopped state.
- `&` Use an ampersand at the end of a command to send the resulting job to the background immediately
- `kill` Sends signals to processes based on process ID or job number. Useful to send SIGTERM or SIGKILL signals to forcibly stop processes.
- `nice` Runs the given command with a lower-than-default CPU priority
- `renice` Changes an existing process' CPU priority
- `ionice` Changes an existing process' IO priority

A simple resumé of above and more

End a Command with &
If you want to push a command into the background, using `&` at the end is an easy way to do that. It comes with a catch, though. Using `&` doesn't disconnect the command away from you. It just pushes it in the background so you can continue using a terminal.

```
command &
```

When the terminal session is closed, the command ends. Using `&` is good if you need to push something off for a bit, but don't expect it to continue forever.

& After a Command, Then Disown It
Running a command with just `&` pushes it off to the back and keeps it running as long as the terminal window is open. If, however, you're looking to keep this command running in constant, even with your terminal session ending, you can use the disown command.
To use this method, first start off adding an &.

```
command &
```

As mentioned above, using `&` pushes this command into the background but doesn't detach it from your user. You can verify this by typing

```
jobs
```

into the terminal. It'll show the command running in the background.
Just type

```
disown
```

into the shell, and it'll do just that (and you can once again verify this with the jobs command).

disown command options

Option	Description
-a	Delete all jobs if jobID is not supplied.
-h	Mark each jobID so that SIGHUP is not sent to the job if the shell exits.
-r	Delete only running jobs.

& After a Command with /dev/null
Adding `&` after a command will push a command into the background, but as a result the background command will continue to print messages into the terminal as you're using it. If you're looking to prevent this, consider redirecting the command to `/dev/null`.

```
command &>/dev/null &
```

This does not prevent the command from closing when the terminal closes. However, like mentioned above, it's possible to use `disown` to disown the running command away from the user.

Nohup, with & and /dev/null
Unlike the previous commands, using `nohup` allows you to run a command in the background and keep it running. How? `Nohup` bypasses the HUP signal (signal hang up), making it possible to run commands in the background even when the terminal is off. Combine this command with redirection to `/dev/null` (to prevent `nohup` from making a `nohup.out` file), and everything goes to the background with one command.

```
$ nohup command &>/dev/null &
```

1. nohup Do not terminate this process even when the stty is cut off.
2. &>/dev/null stdout and stderr go to /dev/null. /dev/null is a dummy device that does not record any output.
3. & run this command as a background task.

Difference between nohup, disown and &

Let's assume you've just typed `foo`:

- The process running `foo` is created.
- The process inherits stdin, stdout, and stderr from the shell. Therefore it is also connected to the same terminal.
- If the shell receives a `SIGHUP`, it also sends a `SIGHUP` to the process (which normally causes the process to terminate).
- Otherwise the shell waits (is blocked) until the process terminates.

Now, let's look what happens if you put the process in the background, that is, type `foo &`:

- The process running `foo` is created.
- The process inherits stdout/stderr from the shell so it still writes to the terminal.
- The process in principle also inherits stdin, but as soon as it tries to read from stdin, it is halted.
- It is put into the list of background jobs the shell manages, which means especially:
    - It is listed with `jobs` and can be accessed using `%n` where n is the job number.
    - It can be turned into a foreground job using `fg`, in which case it continues as if you would not have used `&` on it and if it was stopped due to trying to read from standard input, it now can proceed to read from the terminal.
    - If the shell received a `SIGHUP`, it also sends a `SIGHUP` to the process. Depending on the shell and possibly on options set for the shell, when terminating the shell it will also send a `SIGHUP` to the process.

`Disown` removes the job from the shell's job list, so all the subpoints above don't apply any more including the process being sent a `SIGHUP` by the shell. However note that it `still` is connected to the terminal, so if the terminal is destroyed which can happen if it was a `pty`, like those created by `xterm or ssh`, and the controlling program is terminated, by closing the `xterm` or terminating the `SSH connection`, the program will fail as soon as it tries to read from standard input or write to standard output.

What `nohup` does, on the other hand, is to effectively separate the process from the terminal:

- It closes standard input. The program will `not` be able to read any input, even if it is run in the foreground. It is not halted, but will receive an error code or `EOF`.
- It redirects standard output and standard error to the file `nohup.out`, so the program won't fail for writing to standard output if the terminal fails, so whatever the process writes is not lost.
- It prevents the process from receiving a `SIGHUP` .

Note that `nohup` does `not` remove the process from the shell's job control and also doesn't put it in the background but since a foreground `nohup` job is more or less useless, you'd generally put it into the background using `&`. For example, unlike with `disown`, the shell will still tell you when the `nohup` job has completed unless the shell is terminated before, of course.

Making scripts run at boot time

To understand the way scripts or applications are started at boot time, the first thing you should know is run level. There are total 6 run levels.

Run Level	Generic	Debian Raspberry Pi
0	Halt	Halt
1 - S	Single-user mode	Single-user mode
2	Multi-user mode	Multi-user mode
3	Multi-user mode with networking	
4	Not used/user-definable	Multi-user mode
5	Start the system normally with appropriate display manager (with GUI)	Multi-user mode
6	Reboot	Reboot

Each run level have separate locations under `/etc/`:

```
% ls -l /etc/rc* -d
drwxr-xr-x 2 root root 4096 Feb 20 10:44 /etc/rc0.d
drwxr-xr-x 2 root root 4096 Feb 20 10:44 /etc/rc1.d
drwxr-xr-x 2 root root 4096 Feb 20 10:44 /etc/rc2.d
drwxr-xr-x 2 root root 4096 Feb 20 10:44 /etc/rc3.d
drwxr-xr-x 2 root root 4096 Feb 20 10:44 /etc/rc4.d
drwxr-xr-x 2 root root 4096 Feb 20 10:44 /etc/rc5.d
drwxr-xr-x 2 root root 4096 Feb 20 10:44 /etc/rc6.d
-rwxr-xr-x 1 root root  306 Feb  4 18:58 /etc/rc.local
drwxr-xr-x 2 root root 4096 Feb  4 19:01 /etc/rcS.d
```

Note: rc stands for run control

Everytime your system boots, some scripts under the coressponding run level folder is executed. Eg: if you boot in to run level 2, scripts under `/etc/rc2.d/` will be executed. If you show content of these folder, you'll see that scripts are symlink of scripts under `/etc/init.d/`.

```
% ls -l /etc/rc2.d/
total 4
-rw-r--r-- 1 root root 677 Jul 27  2012 README
lrwxrwxrwx 1 root root  20 Feb 19 11:26 S20kerneloops -> ../init.d/kerneloops
lrwxrwxrwx 1 root root  27 Feb 19 11:26 S20speech-dispatcher -> ../init.d/speech-
dispatcher
lrwxrwxrwx 1 root root  20 Feb 19 11:26 S50pulseaudio -> ../init.d/pulseaudio
lrwxrwxrwx 1 root root  15 Feb 19 11:26 S50rsync -> ../init.d/rsync
lrwxrwxrwx 1 root root  15 Feb 19 11:26 S50saned -> ../init.d/saned
lrwxrwxrwx 1 root root  19 Feb 19 11:26 S70dns-clean -> ../init.d/dns-clean
lrwxrwxrwx 1 root root  18 Feb 19 11:26 S70pppd-dns -> ../init.d/pppd-dns
lrwxrwxrwx 1 root root  14 Feb 19 11:26 S75sudo -> ../init.d/sudo
lrwxrwxrwx 1 root root  17 Feb 20 10:44 S91apache2 -> ../init.d/apache2
lrwxrwxrwx 1 root root  22 Feb 19 11:26 S99acpi-support -> ../init.d/acpi-support
lrwxrwxrwx 1 root root  21 Feb 19 11:26 S99grub-common -> ../init.d/grub-common
lrwxrwxrwx 1 root root  18 Feb 19 11:26 S99ondemand -> ../init.d/ondemand
lrwxrwxrwx 1 root root  18 Feb 19 11:26 S99rc.local -> ../init.d/rc.local
```

This give you an ability to control your service to run under which runlevel. You can make your service run in only run level 2 and stop in others run level. But remember, Only one "runlevel" is executed on bootup, i.e. either runlevel 2 OR 3 OR 4 is executed, not 2 then 3 then 4.

Many of the scripts already present in `/etc/init.d` will give you an example of the kind of things that you can do.

In each run level you boot in, after scripts of this run level are executed, the script `/etc/rc.local` is executed. It means that `/etc/rc.local` will run at the end of boot process, regardless of run level you boot in.

To add some further explanation, the script names that begin with "S" Start a service when entering a run-level and "K" Kill a service when leaving a run-level. Also the numbers (0-99) immediately following the initial S or K permit a fine-grained control over the ORDER in which scripts are run.

The script names starting with a "K" in "/etc/rc<n>.d/" are executed in alphabetical order with the single argument "stop". (killing services). These are started first.

The script names starting with an "S" in "/etc/rc<n>.d/" are executed in alphabetical order with the single argument "start". (starting services). These are started after the S-scripts

Basic "How to"

1. Create a script to start at boot time and place it in the `/etc/init.d/` directory.

   Here's a very simple script, let's call it `sample.sh` which is divided into two parts, code which always runs, and code which runs when called with "start" or "stop".

```
#! /bin/sh
# /etc/init.d/sample.sh
#

# Some things that run always
touch /var/lock/sample.sh

# Carry out specific functions when asked to by the system
case "$1" in
  start)
    echo "Starting script sample.sh "
    echo "Could do more here"
    ;;
  stop)
    echo "Stopping script sample.sh "
    echo "Could do more here"
    ;;
  *)
    echo "Usage: /etc/init.d/sample.sh {start|stop}"
    exit 1
    ;;
esac

exit 0
```

   Note: `/etc/init.d/*.sh` scripts must include "#!/bin/sh" at the top. This is useful when running scripts in parallel.

   Before we can use `update-rc.d` (see later) to make this script running at certain levels, we need to add a LSB style header, documenting dependencies and default runlevel settings. The minimum LSB header looks like this:

```
### BEGIN INIT INFO
# Provides:          sample.sh
# Required-Start:    $network $named $syslog $time
# Should-Start:  cyrus ldap ypbind openslp
# Required-Stop:
# Default-Start:     3 5
# Default-Stop:
# Short-Description:    start the Postfix MTA
### END INIT INFO
```

Minimal content needs just the name of the script, required runlevels and short description like this.

```
### BEGIN INIT INFO
# Provides:         sample.sh
# Required-Start:
# Required-Stop:
# Default-Start:  3 5
# Default-Stop:
# Short-Description: run sample
# Description:
### END INIT INFO
```

All possible LSB tags are:

```
### BEGIN INIT INFO
# Provides:          skeleton
# Required-Start:    $local_fs $syslog
# Required-Stop:     $local_fs $syslog
# Should-Start:      $portmap
# Should-Stop:       $portmap
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# X-Interactive:     true
# X-Start-Before:    nis
# X-Stop-After:      nis
# Short-Description: Example initscript
# Description:       This file should be used to construct scripts to be
#                    placed in /etc/init.d.
### END INIT INFO
```

So, we need to change our sample script like this

```
#! /bin/sh
### BEGIN INIT INFO
# Provides:         sample.sh
# Required-Start:   $remote_fs $network $local_fs
# Should-Start:
# Required-Stop:
# Default-Start:    3 5
# Default-Stop:
# Short-Description:   start the Postfix MTA
### END INIT INFO

# Some things that run always
touch /var/lock/sample.sh

# Carry out specific functions when asked to by the system
case "$1" in
  start)
    echo "Starting script sample.sh "
    echo "Could do more here"
    ;;
  stop)
    echo "Stopping script sample.sh "
    echo "Could do more here"
    ;;
  *)
    echo "Usage: /etc/init.d/sample.sh {start|stop}"
    exit 1
    ;;
esac

exit 0
```

With `# Default-Start:  3 5 m`eaning that the script will run in run level 3 and in run level 5 and `# Required-Start:  $remote_fs $network $local_fs` meaning that our script will be executed after the system has mounted all the filesystems, local and remote, and the network facility is on service. If you want/need your script to be executed at the end of the boot, change the line `# Required-Start`: like this

```
# Required-Start: $all
```

2. make the script executable

```
chmod 755 /etc/init.d/sample.sh
```

3. execute `update-rc.d` to activate it.

```
sudo update-rc.d sample.sh defaults
```

As said the update-rc.d command to create symbolic links in the /etc/rc?.d as appropriate. As we specified 3 and 5, we will find a symbolic link in `/etc/rc3.d` as well as in `/etc/rc5.d`

Note: If you wish to remove the script from the startup sequence in the future run:

```
sudo update-rc.d -f sample.sh remove
```

This will leave the script itself in place, just remove the links which cause it to be executed. You can find more details of this command by running "man update-rc.d".

A bit more "How To"
Let's imagine we want a set of programs and/or scripts to start at boot. First we prepare the programs and the scripts. We will call the programs from scripts so we assume from now on that we only have scripts. We assume the name of the scripts are of the kind: script1.sh script2.sh and so on. Those scripts will be located at the folder /usr/local/sbin.
For instance, imagine that script3.sh is a script that calls a program with several options. Then the content of the script should be something like this:

```
sudo nano script3.sh

#!/bin/sh
# I am a very basic script calling a program with some options
/usr/bin/myprogram3 -option1 -option2 -option3
```

Notice that we wrote the path where the program is located.

The programs or services that are started at boot time have an script in the folder /etc/init.d/. Here's is the simplest script which is divided into two parts, code which runs when called with "start", or code that stops running when called with "stop". We'll give this script the same name as our program to easily find our way back later

```
sudo nano /etc/init.d/myprogram3

#! /bin/sh
### BEGIN INIT INFO
# Provides:          myprogram
# Should-Start:      console-screen dbus network-manager
# Required-Start:    $remote_fs $network $local_fs
# Required-Stop:     $remote_fs
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: start myprogram at boot time
### END INIT INFO
#
```

```
set –e

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

SCRIPT="/usr/local/sbin/script3.sh"
PROGRAMNAME="myprogram3"
case "$1" in
    start)
        $SCRIPT
    ;;
    stop)
        kill -9 $PROGRAMNAME
    ;;
    *)
        echo "Usage: /etc/init.d/foobar {start|stop}"
        exit 1
    ;;
exit 0
```

Once you have your script, give to it execution permission

```
chmod 755 /etc/init.d/myprogram3
```

Then you check that the script works by calling it as root:

```
sudo /etc/init.d/myprogram start
```

and you check that it also stops

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

During boot the script `myprogram` will be called in this way. When you stop the computer, it will also stopped in this way. If `kill` does not do its job, the OS will stop it anyway (which is a dirty way).

Execute `update-rc.d` to activate it.

```
sudo update-rc.d myprogram3 defaults
```

Reboot now and check

```
sudo reboot

sudo ps –aux | grep myprogram3
```

You should see at something similar to this

```
root       728  0.7  3.7  38648 33300 ?        S    14:46   0:12
/usr/bin/myprogram3 -option1 -option2 -option3
root      1227  0.0  0.2   4280  1908 pts/0    S+   15:13   0:00 grep
myprogram3
```

The final rc : rc.local
As said, when rc?.d is executed, the one last script the OS will run is `rc.local`. You could add the necessary script at the end of the file `rc.local` in the directory `/etc/`.

Base file:

```
#!/bin/sh -e
#
# rc.local
#
# This script is executed at the end of each multiuser runlevel.
# Make sure that the script will "exit 0" on success or any other
# value on error.
#
# In order to enable or disable this script just change the execution
# bits.
#
# By default this script does nothing.

# Print the IP address
_IP=$(hostname -I) || true
if [ "$_IP" ]; then
  printf "My IP address is %s\n" "$_IP"
fi

exit 0
```

Modified file:

```
#!/bin/sh -e
#
# rc.local
#
# This script is executed at the end of each multiuser runlevel.
# Make sure that the script will "exit 0" on success or any other
# value on error.
#
# In order to enable or disable this script just change the execution
# bits.
#
# By default this script does nothing.

# Print the IP address
_IP=$(hostname -I) || true
if [ "$_IP" ]; then
  printf "My IP address is %s\n" "$_IP"
fi

cd /home/pi
python mypythoncode.py 1>/dev/null 2>/dev/null &

exit 0
```

So this means that whatever run level we run in, we will always execute `python mypythoncode.py` from within `/home/pi` **and** sending the standard output stream to the null device as well as the standard error stream.

Reboot now and check

```
sudo reboot

sudo ps –aux | grep python
```

You should see something similar to this

```
root       728  0.7  3.7  38648 33300 ?        S    14:46   0:12 python
mypythoncode.py 1>/dev/null 2>/dev/null &
root      1227  0.0  0.2   4280  1908 pts/0    S+   15:13   0:00 grep python
```

Note: all commands in rc.local are being executed as root.

Scheduling Tasks With Cron

Cron is a tool for configuring scheduled tasks on Unix systems, used to schedule commands or scripts to run periodically and at fixed intervals. Tasks range from backing up the user's home folders every day at midnight, to logging CPU information every hour.
The command `crontab` (cron table) is used to edit the list of scheduled tasks in operation, and is done on a per-user basis; each user (including root) has their own crontab.

Editing Crontab

Run `crontab` with the `-e` flag to edit the cron table:

```
crontab -e
```

Select An Editor

The first time you run `crontab` you'll be prompted to select an editor; if you are not sure which one to use, choose `nano` by pressing `Enter`.

Add A Scheduled Task

The layout for a cron entry is made up of six components: minute, hour, day of month, month of year, day of week, and the command to be executed.

```
# m h dom mon dow   command
# * * *   *   *     command to execute
# ┬ ┬ ┬   ┬   ┬
# │ │ │   │   │
# │ │ │   │   └──── day of week (0 - 6)
# │ │ │   │         (0 to 6 are Sunday to Saturday, or use names)
# │ │ │   └──────── month (1 - 12)
# │ │ └──────────── day of month (1 - 31)
# │ └────────────── hour (0 - 23)
# └──────────────── min (0 - 59)
```

Note: a * means every period

For example:

```
0 0 * * *  /home/pi/backup.sh
```

This cron entry would run the `backup.sh` script every day at midnight.
There is a handy online tool to generate crontab entries

https://crontab-generator.org/

View Scheduled Tasks

View your currently saved scheduled tasks with:

```
crontab -l
```

Run A Task On Reboot, Hourly, Daily, Weekly, Monthly, Yearly, Annually

To run a command every time the Raspberry Pi starts up, write @reboot instead of the time and date. For example:

```
@reboot python /home/pi/myscript.py
```

This will run your Python script every time the Raspberry Pi reboots. If you want your command to be run in the background while the Raspberry Pi continues starting up, add a space and & at the end of the line, like this:

```
@reboot python /home/pi/myscript.py &
```

These special time specification "nicknames" which replace the 5 initial time and date fields, and are prefixed with the '@' character, are supported:

```
@reboot    :    Run once after reboot.
@yearly    :    Run once a year, ie.  "0 0 1 1 *".
@annually  :    Run once a year, ie.  "0 0 1 1 *".
@monthly   :    Run once a month, ie. "0 0 1 * *".
@weekly    :    Run once a week, ie.  "0 0 * * 0".
@daily     :    Run once a day, ie.   "0 0 * * *".
@hourly    :    Run once an hour, ie. "0 * * * *".
```

How To Use a Simple Bash Script To Restart Server Programs

To ensure that the your application remain active as much as possible, even after a crash, one can create a simple bash script to check if the program is running, and if it is not, to launch it. By using cron to schedule the script to be executed on a regular basis, we can make sure that program relaunches whenever it goes down.

Bash Script
The first step in this process is to create the script itself that simply acts as an ON-switch. In this sample script we check if `apache2` is running

```
nano launch.sh

#!/bin/sh

ps auxw | grep apache2 | grep -v grep > /dev/null

if [ $? != 0 ]
then
        /etc/init.d/apache2 start > /dev/null
Fi
```

Once you have saved the script, you must give it executable permissions in order to be able to run it:

```
chmod +x launch.sh
```

Apache can be replaced with any required application. Should you want to set up the script for a variety of applications, you can create a new script for each one, placing it on its own line in the cron file.

Cron Setup
With the script in hand, we need to set up the schedule on which it will run. The cron utility allows us to schedule at what intervals the script should execute. Start by opening up the cron file:

```
crontab –e
```

Cron has a detailed explanation of how the timing system works at the beginning.

Once you know how often you want the script to run, you can write in the corresponding line.

The most often that the script can run in cron is every minute. Should you want to set up such a small increment, you can use this template:

```
* * * * * ~/launch.sh
```

Every five minutes would be set up like this:

```
*/5 * * * * ~/launch.sh
```

Every hour would be set up like this:

```
0 * * * * ~/launch.sh
```

Every 2 hour would be set up like this:

```
0 */2 * * * ~/launch.sh
```

Run a script at login

All interactive sessions of bash will read the initialization file `~/.bashrc`.

So you can just add the script at the end of the root's or normal user `.bashrc` i.e. `/root/.bashrc` or `/home/$USER/.bashrc` assuming the script is executable:

```
echo '/path/to/script.sh' >>/root/.bashrc
echo '/path/to/script.sh' >>/home/$USER/.bashrc
```

or use nano to edit the file

```
nano /root/.bashrc
nano /home/$USER/.bashrc
```

Now the script will be always run when root or user opens a new interactive shell. If you only want to run while login only and not all all interactive sessions, you should rather use `~/.bash_profile` or `~/.bash_login` or `~/.profile`. The first one available following the order.

If you're trying to do this for every user on the system that may log into a bash shell, basically root, and any other "normal" user, then you'll want to add the command into

```
/etc/profile
```

or add a script to

```
/etc/profile.d
```

Daemons, Services & Processes

1. A **daemon** is a background, non-interactive program. It is detached from the keyboard and display of any interactive user. The word daemon for denoting a background program is from the Unix culture; it is not universal.

2. A **service** is a program which responds to requests from other programs over some inter-process communication mechanism (usually over a network). A service is what a server provides**.** For example, the NFS port mapping service is provided as a separate portmap service, which is implemented as the portmapd daemon.
   A service doesn't have to be a daemon, but usually is. A user application with a GUI could have a service built into it: for instance, a file-sharing application. Another example is the X Window service, which is anything but in the background: it takes over your screen, keyboard and pointing device. It is a service because it responds to requests from applications (to create and manipulate windows, et cetera), which can even be elsewhere on the network. But the X service also responds to your every keystroke and mouse movement.

3. A **process** is one or more threads of execution together with their shared set of resources, the most important of which are the address space and open file descriptors. A process creates an environment for these threads of execution which looks like they have an entire machine all to themselves: it is a virtual machine.
   Inside a process, the resources of other processes, and of the kernel, are invisible and not directly accessible (at least not to a thread which is executing user-space code). For example, there is no way to refer to the open files of another process, or their memory space; it is as if those things do not even exist.
   The process, and its relation to the kernel and other processes, perhaps constitutes the most important abstraction in Unix-like operating systems. The resources of the system are compartmentalized into processes, and nearly everything is understood as happening inside one process or another.

Daemonize your Python script

Install ‘daemonize’ from PyPI - Python Package Index

```
pip install daemonize
```

and then it in your script this way

```
import os, sys
from daemonize import Daemonize
…
…
…

def main()
      # your main code here

if __name__ == '__main__':
        scriptname=os.path.basename(sys.argv[0])
        pidfile='/tmp/%s' % scriptname
        daemon = Daemonize(app=scriptname,pid=pidfile, action=main)
        daemon.start()
```

How to create systemd service unit in Linux

Although systemd has been the object of many controversies, to the point the some distributions were forked just to get rid of it, in the end it has become the de-facto standard init system in the Linux world.

The systemd init system

All the major distributions, such as Rhel, CentOS, Fedora, Ubuntu, Debian and Archlinux, adopted systemd as their init system. Systemd, actually, is more than just an init system, and that's one of the reasons why some people are strongly against its design, which goes against the well established unix motto: "do one thing and do it well". Where other init systems use simple shell script to manage services, systemd uses its own .service files (units with the .service suffix): in this tutorial we will see how they are structured and how to create and install one.

Anatomy of a service unit

What is a service unit? A file with the .service suffix contains information about a process which is managed by systemd. It is composed by three main sections:

- [Unit]: this section contains information not specifically related to the type of the unit, such as the service description
- [Service]: contains information about the specific type of the unit, a service in this case
- [Install]: This section contains information about the installation of the unit

Let's analyze each of them in detail.

The [Unit] section
The [Unit] section of a .service file contains the description of the unit itself, and information about its behavior and its dependencies: (to work correctly a service can depend on another one). Here we discuss some of the most relevant options which can be used in this section

The "Description" option
First of all we have the Description option. By using this option we can provide a description of the unit. The description will then appear, for example, when calling the systemctl command, which returns an overview of the status of systemd. Here it is, as an example, how the description of httpd service is defined on a Fedora system:

```
[Unit]
Description=The Apache HTTP Server
```

The "After" option
By using the After option, we can state that our unit should be started after the units we provide in the form of a space-separated list. For example, observing again the service file where the Apache web service is defined, we can see the following:

```
After=network.target remote-fs.target nss-lookup.target httpd-init.service
```

The line above instructs systemd to start the service unit `httpd.service` only after the `network, remove-fs, nss-lookup targets` and the `httpd-init service`

Specifying hard dependencies with "Requires"

As we briefly mentioned above, a unit (a service in our case) can depend on other units (not necessarily "service" units) to work correctly: such dependencies can be declared by using the `Requires` option.
If any of the units on which a service depends fails to start, the activation of the service is stopped: this is why those are called hard dependencies. In this line, extracted from the service file of the avahi-daemon, we can see how it is declared as dependent from the `avahi-daemon.socket` unit:

```
Requires=avahi-daemon.socket
```

Declaring "soft" dependencies with "Wants"
We just saw how to declare the so called "hard" dependencies for the service by using the `Requires` option; we can also list "soft" dependencies by using the `Wants` option.
What is the difference? As we said above, if any "hard" dependency fails, the service will fail itself; a failure of any "`soft`" dependency, however, doesn't influence what happens to the dependent unit. In the provided example, we can see how the docker.service unit has a soft dependency on the `docker-storage-setup.service` one:

```
[Unit]
Wants=docker-storage-setup.service
```

The [Service] section
In the `[Service]` section of a service unit, we can specify things as the command to be executed when the service is started, or the type of the service itself. Let's take a look at some of them.

Starting, stopping, and reloading a service
A service can be started, stopped, restarted or reloaded. The commands to be executed when performing each of these actions can be specified by using the related options in the `[Service]` section.
The command to be executed when a service starts, is declared by using the `ExecStart` option. The argument passed to the option can also be the path to a script. Optionally, we can declare commands to be executed before and after the service is started, by using the `ExecStartPre` and `ExecStartPost` options respectively. Here is the command used to start the NetworkManager service:

```
[Service] ExecStart=/usr/sbin/NetworkManager –no-daemon
```

In a similar fashion, we can specify the command to be executed when a service is reloaded or stopped, by using the `ExecStop` and `ExecReload` options. Similarly to what happens with `ExecStartPost`, a command or multiple commands to be launched after a process is stopped, can be specified with the `ExecStopPost` option.

The type of the service
Systemd defines and distinguish between some different type of services depending on their expected behavior. The type of a service can be defined by using the Type option, providing one of these values:

- simple
- forking
- oneshot
- dbus
- notify

The default type of a service, if the `Type` and `Busname` options are not defined, but a command is provided via the `ExecStart` option, is `simple`. When this type of service is set, the command declared in `ExecStart` is considered to be the main process/service.
The forking type works differently: the command provided with `ExecStart` is expected to fork and launch a child process, which will become the main process/service. The parent process is expected to die once the startup process is over.
The oneshot type is used as the default if the `Type` and `ExecStart` options are not defined. It works pretty much like simple: the difference is that the process is expected to finish its job before other units are launched. The unit, however, it's still considered as "`active`" even after the command exits, if the `RemainAfterExit` option is set to "yes" (the default is "no").
The next type of service is `dbus`. If this type of service is used, the daemon is expected to get a name from `Dbus`, as specified in the `BusName` option, which in this case, becomes mandatory. For the rest it works like the simple type. Consequent units, however, are launched only after the `DBus name` is acquired.
Another process works similarly to `simple`, and it is `notify`: the difference is that the daemon is expected to send a notification via the `sd_notify` function. Only once this notification is sent, consequent units are launched.

Set process timeouts
By using specific options, it's possible to define some timeouts for the service. Let's start with `RestartSec`: by using this option, we can setup the amount of time (by default in seconds) systemd should wait before restarting a service. A timespan can also be used as a value for this option, as "`5min 20s`". The default is `100ms`.

The `TimeoutStartSec` and `TimeoutStopSec` options can be used to specify, respectively, the timeout for a service startup and stop, in seconds. In the first case, if after the specified timeout the daemon startup process it's not completed, it will be considered to be failed.
In the second case, if a service is to be stopped but is not terminated after the specified timeout, first a `SIGTERM` and then, after the same amount of time, a `SIGKILL` signal are sent to it. Both options accepts also a timespan as a value and can be configured at once, with a shortcut: `TimeoutSec`. If infinity is provided as a value, the timeouts are disabled.
Finally, we can setup the limit for the amount of time a service can run, using the `RuntimeMaxSec`. If a service exceeds this timeout, it's terminated and considered as failed.

The [Install] section
In the `[install]` section, we can use options related to the service installation. For example, by using the `Alias` option, we can specify a space separated list of aliases to be used for the service when using the `systemctl` commands (except `enable`).
Similarly to what happens with the `Requires` and `Wants` options in the `[Unit]` section, to establish dependencies, in the `[install]` section, we can use `RequiredBy` and `WantedBy`. In both cases we declare a list of units which depend on the one we are configuring: with the former option they will be hard-dependent on it, with the latter they will be considered only as weak-dependent. For example:

```
[Install]
WantedBy=multi-user.target
```

With the line above we declared that the multi-user target has a soft dependency on our unit. In `systemd` terminology, units ending with the `.target` suffix, can be associated with what were called runtimes in other init systems as `Sysvinit`. In our case, then, the multi-user target, when reached, should include our service.

Creating and installing a service unit
There are basically two places in the filesystem where systemd service units are installed: `/usr/lib/systemd/system` and `/etc/systemd/system`. The former path is used for services provided by installed packages, while the latter can be used by the system administrator for its own services which can override the default ones.
Let's create a custom service example. Suppose we want to create a service which disables the wake-on-lan feature on a specific ethernet interface (`ens5f5` in our case) when it is started, and re-enables it when it is stopped. We can use the `ethtool` command to accomplish the main task. Here is how our service file could look like:

```
[Unit]
Description=Force ens5f5 ethernet interface to 100Mbps
Requires=Network.target
After=Network.target

[Service]
Type=oneshot
RemainAfterExit=yes
ExecStart=/usr/sbin/ethtool -s ens5f5 wol d
ExecStop=/usr/sbin/ethtool -s ens5f5 wol g

[Install]
WantedBy=multi-user.target
```

We set a simple unit description, and declared that the service depends on the `network.target` unit and should be launched after it is reached. In the `[Service]` section we set the type of the service as `oneshot`, and instructed `systemd` to consider the service to be active after the command is executed, using the `RemainAfterExit` option. We also defined the commands to be run when the service is started and stopped. Finally, in the `[Install]` section we basically declared that our service should be included in the multi-user target.
To install the service we will copy the file into the `/etc/systemd/system` directory as `wol.service`, than we will start it:

```
$ sudo cp -f ./wol.service /lib/systemd/system/
$ sudo systemctl enable wol.service
$ sudo systemctl daemon-reload
$ sudo systemctl start wol.service
```

We can verify the service is active, with the following command:

```
$ systemctl is-active wol.service
active
```

The output of the command, as expected, is active. Now to verify that "wake on lan" has been set to `d`, and so it is now disabled, we can run:

```
$ sudo ethtool ens5f5|grep Wake-on
Supports Wake-on: pg
Wake-on: d
```

Now, stopping the service should produce the inverse result, and re-enable wol:

```
$ sudo systemctl stop wol.service
$ sudo ethtool ens5f5|grep Wake-on
Supports Wake-on: pg
Wake-on: g
```

To uninstall the service, run

```
$ sudo systemctl stop wol.service
$ sudo systemctl disable wol.service
$ sudo rm -f /lib/systemd/system/wol.service
```

Conclusions
In this tutorial we saw how a systemd service file is composed, what are its sections, and some of the options which can be used in each of them. We learned how to setup a service description, to define its dependencies and to declare the commands that should be executed when it is started, stopped or reloaded.
Since systemd, like it or not, has become the standard init system in the Linux world, it's important to become familiar to its way of doing things. The official systemd services documentation can be found on the freedesktop website. You could also be interested in reading our article about managing services with systemd.

Still to intregated into text above

You've written some code for your Raspberry Pi that runs your project. You need to run it every time your system starts, but how do you do this?
Perhaps the easiest method is adding a line to the `/etc/rc.local` file. This is a Bash script that runs each time the computer boots up. It's quick and easy, but there are a few drawbacks to it, namely:

- If your script outputs any information or errors, where do these errors go?
- If your script crashes, how do you know, and how do you restart it?
- Can you stop or reload your script easily?

Fortunately, the operating system used by Raspberry Pi computers (Linux) has a mechanism for managing things in the background: `systemd`.
This is a service layer that's widely used to manage all sorts of bits of essential software on your Raspberry Pi or other Linux computer, from databases to window managers. It provides a way of controlling software that runs in the background. Let's take a look at what this means in practice.
`Systemd` can handle a huge range of things, but we'll be looking at '`services`' which is the name used for software that runs in the background because it typically provides a service.
Our really simple script will simply use Python to output the phrase 'hello world' every second. This is, admittedly, not the most useful script, but it'll show the technique of running our code automatically, and capturing the output. This can be done with the following code:

```
#!/usr/bin/python

import time
import sys

while True:
    print "hello world"
    sys.stdout.flush()
    time.sleep(1)
```

There are a couple of unusual bits in this.
The first is the first line – this is known as a `shebang`, and it's used to tell the operating system what program to use to run the script. This is the complete path to the binary for Python. We got this on Raspbian OS. It might be a little different on other Linux distros.
If you're unsure, you can run the command '`which python`' and it will tell you.
The second unusual thing is the call to `sys.stdout.flush()`. This isn't strictly necessary, and it just tells the operating system to make sure all the output has made its way through the output buffers and on to its destination. We found we had a large lag between the script running and output making its way to the logs if we didn't use it. In practice, you may find that you can live without this in your programs (or just include it once in a main loop).
We saved this in a file called `hello-service.py` in `/home/pi`.
Finally, we need to make this file executable by running the command:

```
chmod a+x /home/pi/hello-service.py
```

That's our script ready. Let's now get it running as a service. In order to let `systemd` know what we want to do with our simple service, we need to create a unit file. There's a lot that can go into a unit file, but for basic usage, they can be quite simple. Ours will be as follows:

```
[Unit]
Description=A service to say hello world
After=systemd-user-sessions.service

[Service]
```

```
Type=simple
ExecStart=/home/pi/hello-service.py
```

This tells a bit about our script. The `After` section tells `systemd` when we want our script to start – in this case, after the `systemd-user-sessions` which is one of the system services that starts every boot.
Type `simple`, as opposed to forking, tells `systemd` that the command will continue to run in the session it was started.
The final line tells `systemd` what command to run.
Save this unit file as `hello.service` in your Home directory, then copy it to the `systemd` directory with:

```
sudo cp /home/pi/hello.service /etc/systemd/system
```

That's everything set up and ready to go. We just need to let `systemd` know what we want to do. You can start your service with:

```
systemctl start hello.service
```

see its status with:

```
systemctl status hello.service
```

and stop it with:

```
systemctl stop hello.service
```

It's this management of the running code that makes this method of running code much easier.
There's no fussing around trying to find PIDs of processes in order to stop or restart them. What's more, once you start a service, `systemd` will monitor it, and if it crashes for some reason, it will attempt to restart it. Obviously, this isn't a reason to create code that's unstable, but it is an extra line of defence if your code has to run all day by itself.
If you want to see the output of the service, you can use `journalctl`:

```
journalctl -u hello -e
```

What we've done so far will get the service up and running, but it won't automatically start it every time you boot the system. For that, you need to enable the service with:

```
systemctl enable hello.service
```

With this done, you can restart your machine, and it will automatically start. If you want to stop it starting automatically, you can disable it with:

```
systemctl disable hello.service
```

There's far more to `systemd` than we've looked at here, but with these basics, you can get your code up and running and make sure it's looked after properly as it whirs away in the background.

---

chmod a+x /home/pi/hello.py

[Unit]
Description=A service to say hello world
Requires=Network.target
After=systemd-user-sessions.service

```
[Service]
Type=simple
ExecStart=/home/pi/hello.py

[Install]
WantedBy=multi-user.target


sudo cp /home/pi/hello.service /etc/systemd/system

systemctl start hello.service

systemctl status hello.service

systemctl enable hello.service
systemctl hello.service
systemctl daemon-reload
systemctl start hello.service

systemctl is-active hello.service


systemctl stop hello.service
systemctl disable hello.service
```

```
# This file is part of ArduSub Altimeter AJ-SR04M project.
#
# copy altimeter.service unit file in /etc/systemd/system/ directory
# and start the service

[Unit]
Description=Altimeter logging service
#Documentation=[git_url]
#Requires=[maybe MAVLink ?]
#BindsTo=[maybe MAVLink ?]
After=multi-user.target

[Service]
Type=simple
ExecStart=/home/pi/altimeter/altimeter.py
Restart=on-abort
KillSignal=SIGINT

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