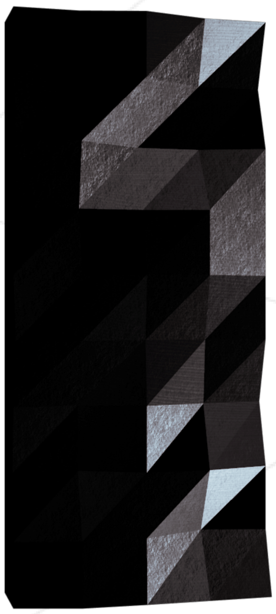

Taskmaster

Leather vest, bullwhip, sadistic tendencies... LGTM.

Maxime Montinet zaz@42.fr
42 staff staff@42.fr

Abstract: The goal of this project is to make a job control daemon, with features similar to supervisor.

Contents

Chapter I

Foreword

Here's a small but useful piece of information on the `Dwellers`:

```
Picking a fight with a species as widespread, long-lived, irascible and (when
it suited them) single-minded as the Dwellers too often meant that just when
(or even geological ages after when) you thought that the dust had long since
settled, bygones were bygones and any unfortunate disputes were all ancient
history, a small planet appeared without warning in your home system,
accompanied by a fleet of moons, themselves surrounded with multitudes of
asteroid-sized chunks, each of those riding cocooned in a fuzzy shell made up
of untold numbers of decently hefty rocks, every one of them travelling
surrounded by a large landslide's worth of still smaller rocks and pebbles, the
whole ghastly collection travelling at so close to the speed of light that the
amount of warning even an especially wary and observant species would have
generally amounted to just about sufficient time to gasp the local equivalent
of "What the fu--?" before they disappeared in an impressive, if wasteful,
blaze of radiation.
```

What are `Dwellers` ? Google it ! No, but seriously, go read `The Algebraist`. This project is way easier if you have read it.

Chapter II

Introduction

Your job here is to make a fully-fledged job control daemon. A pretty good example of this would be supervisor.

For the sake of keeping it simple, your program will not run as `root`, and does not HAVE to be a daemon. It will be started via shell, and do its job while providing a control shell to the user.

Chapter III

Basic features

Your program must be able to start jobs as child processes, and keep them alive, restarting them if necessary. It must also know at all times if these processes are alive or dead (This must be accurate).

Information on which programs must be started, how, how many, if they must be restarted, etc... will be contained in a configuration file, the format of which is up to you (YAML is a good idea, for example, but use whatever you want). This configuration must be loaded at launch, and must be reloadable, while `taskmaster` is running, by sending a SIGHUP to it. When it is reloaded, your program is expected to effect all the necessary changes to its run state (Removing programs, adding some, changing their monitoring conditions, etc ...), but it must NOT de-spawn processes that haven't been changed in the reload.

Your program must have a logging system that logs events to a local file (When a program is started, stopped, restarted, when it dies unexpectedly, when the configuration is reloaded, etc ...)

When started, your program must remain in the foreground, and provide a control shell to the user. It does not HAVE to be a fully-fledged shell like `42sh`, but it must be at the very least usable (Line editing, history... completion would also be nice). Take inspiration from `supervisor`'s control shell, `supervisorctl`.

This shell must at least allow the user to:

- See the status of all the programs described in the config file ("status" command)
- Start / stop / restart programs
- Reload the configuration file without stopping the main program
- Stop the main program

The configuration file must allow the user to specify the following, for each program that is to be supervised:

- The command to use to launch the program

- The number of processes to start and keep running
- Whether to start this program at launch or not
- Whether the program should be restarted always, never, or on unexpected exits only
- Which return codes represent an "expected" exit status
- How long the program should be running after it's started for it to be considered "successfully started"
- How many times a restart should be attempted before aborting
- Which signal should be used to stop (i.e. exit gracefully) the program
- How long to wait after a graceful stop before killing the program
- Options to discard the program's stdout/stderr or to redirect them to files
- Environment variables to set before launching the program
- A working directory to set before launching the program
- An umask to set before launching the program

Chapter IV

Bonus features

You are encouraged to implement any supplemental feature you think your project will benefit from. You will get points for it if it is correctly implemented and at least vaguely useful.

Here are some ideas to get you started:

- Privilege de-escalation on launch (Needs to be started as root, so you would need a VM for this ...)

- Client/server archictecture to allow for two separate programs : A daemon, that does the actual job control, and a control program, that provides a shell for the user, and communicates with the daemon over UNIX or TCP sockets. (Very much like `supervisord` and `supervisorctl`)

- More advanced logging/reporting facilities (Alerts via email/http/syslog/etc...)

- Allow the user to "attach" a supervised process to its console, much in the way that `tmux` or `screen` do, then "detach" from it and put it back in the background.

Chapter V

Constraints

V.1 Language constraints

You are free to use whatever language you want. Libraries are allowed for the purposes of parsing the configuration files and, if you choose to implement it, the client/server bonus. Other than that, you are strictly limited to your language's standard library.

V.2 Defense session

For the defense session, be prepared to :

- Demonstrate that your program correctly implements each and every required feature, by running it with a configuration file you will provide.

- Have your program tested by your grader in various ways, including, but not limited to, manually killing supervised processes, trying to launch processes that never start correctly, launching processes that generate lots of output, etc...

Chapter VI

Appendix

VI.1 Example configuration file

This is what a configuration file for your `taskmaster` COULD look like :

```
programs:
    nginx:
        cmd: "/usr/local/bin/nginx -c /etc/nginx/test.conf"
        numprocs: 1
        umask: 022
        workingdir: /tmp
        autostart: true
        autorestart: unexpected
        exitcodes:
            - 0
            - 2
        startretries: 3
        starttime: 5
        stopsignal: TERM
        stoptime: 10
        stdout: /tmp/nginx.stdout
        stderr: /tmp/nginx.stderr
        env:
            STARTED_BY: taskmaster
            ANSWER: 42
    vogsphere:
        cmd: "/usr/local/bin/vogsphere-worker --no-prefork"
        numprocs: 8
        umask: 077
        workingdir: /tmp
        autostart: true
        autorestart: unexpected
        exitcodes: 0
        startretries: 3
        starttime: 5
        stopsignal: USR1
        stoptime: 10
        stdout: /tmp/vgsworker.stdout
        stderr: /tmp/vgsworker.stderr
```

VI.2 Trying out supervisor

`supervisor` is available on PyPI as a Python package. To try it out, the simplest way is to create a `virtualenv` in your home, activate it, and then install `supervisor` with `"pip install supervisor"`. You may have to install `python` before, it's available on `Homebrew`.

You can then make a configuration file to manage one or two programs, launch `supervisord -c myconfigfile.conf`, then interact with it using `supervisorctl`.

Keep in mind that `supervisor` is a mature, feature-rich program, and that what you must do with `taskmaster` is less complicated, so you should just see it as a source of inspiration. For example, `supervisor` offers the control shell on a separate process that communicates with the main program via a UNIX-domain socket, while you only have to provide a control shell in the main program.

If you have doubts about what behaviour your program should have in a certain case, or what meaning to give to some options... well, when in doubt, do it like `supervisor` does, you can't go wrong.