Lesson #10 - systemd unit file management

Advanced Linux Administration

Aleš Zelinka

Introduction

30 min

Goals

- Creation of a new unit file for a service
- Customization of an existing unit file
- Grouping of services into targets
- Managing resources used up by units (cgroups)
- Non-service unit types

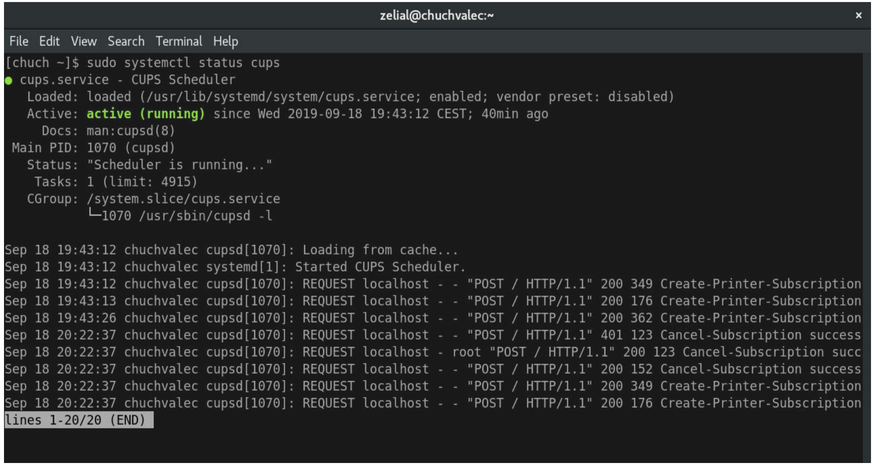
Quick recap of systemd basics from lesson #2

- Everything is a unit: service / target / …
- systemctl status|start|stop|enable|disable
- Units have dependencies on each other and through this create a tree
  - $ systemctl list-dependencies <unit>

Unit vs Unit file

- systemd unit is an abstract object, a basic building block systemd works with
- unit file is a config file, a textual representation of the unit

- sshd.service ~ the unit
  - `$ systemctl status unit`
- /usr/lib/systemd/system/sshd.service
  - `$ man systemd.unit`
  - `$ man systemd.syntax`



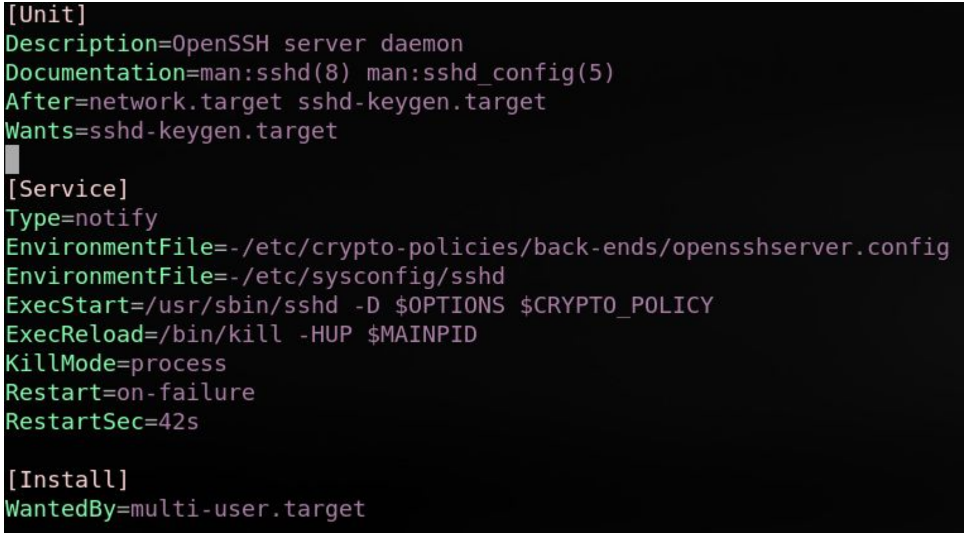
```
[chuch ~]$ sudo systemctl status cups
● cups.service - CUPS Scheduler
   Loaded: loaded (/usr/lib/systemd/system/cups.service; enabled; vendor preset: disabled)
   Active: active (running) since Wed 2019-09-18 19:43:12 CEST; 40min ago
     Docs: man:cupsd(8)
 Main PID: 1070 (cupsd)
   Status: "Scheduler is running..."
    Tasks: 1 (limit: 4915)
   CGroup: /system.slice/cups.service
           └─1070 /usr/sbin/cupsd -l

Sep 18 19:43:12 chuchvalec cupsd[1070]: Loading from cache...
Sep 18 19:43:12 chuchvalec systemd[1]: Started CUPS Scheduler.
Sep 18 19:43:12 chuchvalec cupsd[1070]: REQUEST localhost - - "POST / HTTP/1.1" 200 349 Create-Printer-Subscription
Sep 18 19:43:13 chuchvalec cupsd[1070]: REQUEST localhost - - "POST / HTTP/1.1" 200 176 Create-Printer-Subscription
Sep 18 19:43:26 chuchvalec cupsd[1070]: REQUEST localhost - - "POST / HTTP/1.1" 200 362 Create-Printer-Subscription
Sep 18 20:22:37 chuchvalec cupsd[1070]: REQUEST localhost - - "POST / HTTP/1.1" 401 123 Cancel-Subscription success
Sep 18 20:22:37 chuchvalec cupsd[1070]: REQUEST localhost - root "POST / HTTP/1.1" 200 123 Cancel-Subscription succ
Sep 18 20:22:37 chuchvalec cupsd[1070]: REQUEST localhost - - "POST / HTTP/1.1" 200 152 Cancel-Subscription success
Sep 18 20:22:37 chuchvalec cupsd[1070]: REQUEST localhost - - "POST / HTTP/1.1" 200 349 Create-Printer-Subscription
Sep 18 20:22:37 chuchvalec cupsd[1070]: REQUEST localhost - - "POST / HTTP/1.1" 200 176 Create-Printer-Subscription
lines 1-20/20 (END)
```

Anatomy of a service unit file

A unit file is a plain text ini-style file

- **[Unit]** section - common unit options used during runtime
  - $ man systemd.unit
- **[Service]** section - specific to services
  - $ man systemd.service
- **[Install]** section - common unit options used when enabling/disabling the unit (during ‘installation’)
  - Tells systemd where to put the unit in the dependency tree
  - $ man systemd.unit



```
[Unit]
Description=OpenSSH server daemon
Documentation=man:sshd(8) man:sshd_config(5)
After=network.target sshd-keygen.target
Wants=sshd-keygen.target

[Service]
Type=notify
EnvironmentFile=-/etc/crypto-policies/back-ends/opensshserver.config
EnvironmentFile=-/etc/sysconfig/sshd
ExecStart=/usr/sbin/sshd -D $OPTIONS $CRYPTO_POLICY
ExecReload=/bin/kill -HUP $MAINPID
KillMode=process
Restart=on-failure
RestartSec=42s

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

Anatomy of a service unit file - [Service] section #1

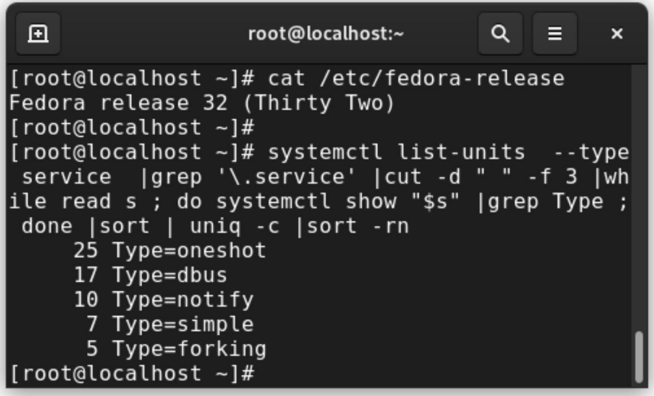
Q: How does systemd know that a service is started / running?

Anatomy of a service unit file - [Service] section #2

Important options: **Type**

How does systemd know that a service is started?

- **Simple** - default. Considered started after the process is forked. For simple, single-process stuff.
- **Notify** - the service tells systemd
- **Forking** - classic unix daemons, considered started after process has exited (after forking a child process)
- **Oneshot** - considered started after the process exits. For stuff that doesn’t keep running
- ...



```
[root@localhost ~]# cat /etc/fedora-release
Fedora release 32 (Thirty Two)
[root@localhost ~]#
[root@localhost ~]# systemctl list-units  --type
 service  |grep '\.service' |cut -d " " -f 3 |wh
ile read s ; do systemctl show "$s" |grep Type ;
 done |sort | uniq -c |sort -rn
     25 Type=oneshot
     17 Type=dbus
     10 Type=notify
      7 Type=simple
      5 Type=forking
[root@localhost ~]#
```

Anatomy of a service unit file - [Service] section #3

Important options: **ExecStart** & friends

- **ExecStart** - what gets executed when systemctl start is called
  - `ExecStart=/usr/bin/my_service`
- **ExecStop** - command responsible for stopping the service, if not specified service gets killed (SIGTERM + SIGKILL)
- **ExecReload -** command used to reload configuration, if not specified `systemctl reload <service>` will not be available

Stdout & stderr goes to journal ~ `$ journalctl -u name.service`

These are not shell commands! You can't use shell constructs directly

✗

```
ExecStart=test && echo yes || echo no
```

✓

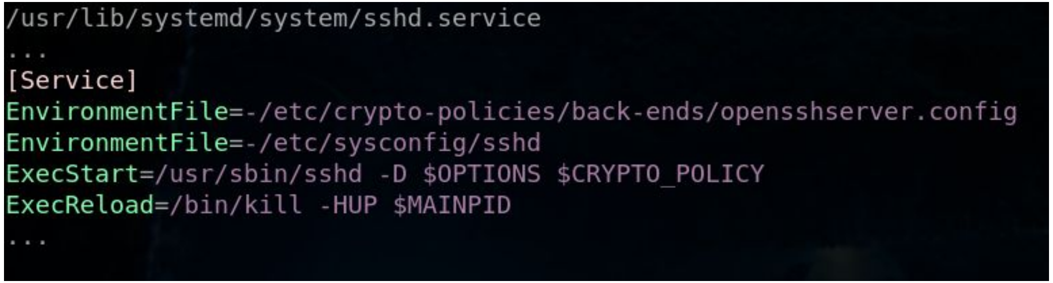
```
ExecStart=bash -c "test && echo yes || echo no"
```

Anatomy of a service unit file - [Service] section #4

For passing options to the executed process without having to edit the unit file itself.

Q: Why should I not directly edit the unit file?

- **Environment** - define environmental variables directly
    - `Environment=LANG=C “OTHER_VAR=a b c d”`
- **EnvironmentFile** - path to a file where environment variables are defined
    - `EnvironmentFile=/etc/my_service_config`



```
/usr/lib/systemd/system/sshd.service
...
[Service]
EnvironmentFile=-/etc/crypto-policies/back-ends/opensshserver.config
EnvironmentFile=-/etc/sysconfig/sshd
ExecStart=/usr/sbin/sshd -D $OPTIONS $CRYPTO_POLICY
ExecReload=/bin/kill -HUP $MAINPID
...
```

Anatomy of a service unit file - Documentation

```
[Unit]
```

Description - A human readable name for the unit.

- `Description=OpenSSH server daemon`

Documentation - A space-separated list of URIs referencing documentation for this unit or its configuration

- `Documentation=man:sshd(8) man:sshd_config(5)`
- `Documentation=man:auditd(8) https://github.com/linux-audit/audit-documentation`

Anatomy of a service unit file - relations

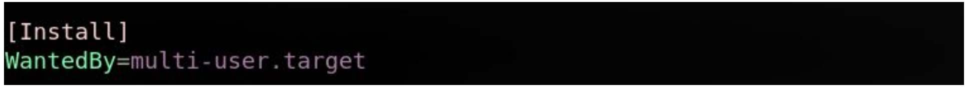
There are two types of relations in between units

- Activation
  - A unit X has to be started before dependant unit Y can be started
  - E.g. a service wants to be autostarted after boot so it makes the default target depend on itself
- Ordering
  - When both X and Y units are started (e.g. because of their activation dependencies), X has to start before Y - Y’s start is delayed after X has finished starting
  - E.g. a web server service only starts after network-online.target is reached

Anatomy of a service unit file - activation dependency #1

[Install] section

- **WantedBy**=foo.service in a service bar.service means that bar will be started when foo is being started.
    - Bar will be started even if foo fails to start
- **RequiredBy**=foo.service in a service bar.service means that bar will be started when foo is being started.
    - Bar will **not** be started if foo fails to start



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

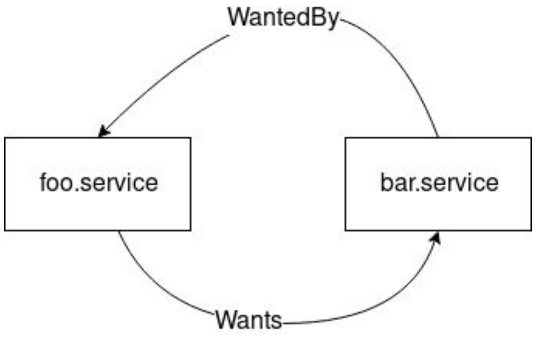
Anatomy of a service unit file - activation dependency #2

Wants and **Requires** are reversed **WantedBy** and **RequiredBy** used in the [Unit] section:

- `bar.service having Wants=foo.service`

is equivalent to

- `foo.service having WantedBy=bar.service`



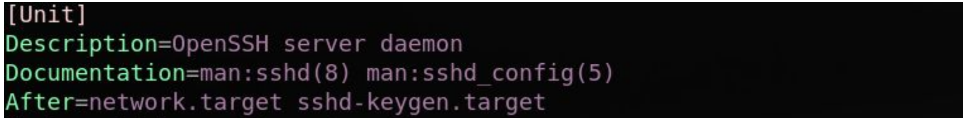
```
$ systemctl list-dependencies
```

- Shows the tree based on activation dependencies

Anatomy of a service unit file - ordering #1

[Unit] section

- **After/Before** - enforce ordering of unit starting



```
[Unit]
Description=OpenSSH server daemon
Documentation=man:sshd(8) man:sshd_config(5)
After=network.target sshd-keygen.target
```

```
$ systemctl --after list-dependencies
```

- Shows the tree based on ordering dependencies

Editing a unit file

Do not edit vendor unit files in `/usr/lib/systemd/system/`!

- New unit file or whole replacement @ `/etc/systemd/system/`
  - `$ systemctl edit --full name.unit`
- Drop-in snippet @ `/etc/systemd/system/name.unit.d/`
  - All snippets get merged with the vendor unit file
  - `$ systemctl edit name.unit`
- Drop all changes, reverting to the vendor file
  - `$ systemctl revert name.unit`
- Show what and from where will systemd load content
  - `$ systemctl status|cat name.unit`

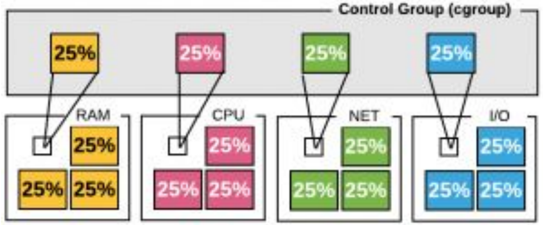
DEMO: edit httpd.service, show status & cat

Managing resources of a service: cgroups

Systemd tracks all processes belonging to a service via cgroups.

cgroups allows you to

- Hierarchically group and label processes
  - Systemd enforces flat hierarchy of a unit == cgroup
- Apply resource limits to the groups
  - Memory
  - CPU
  - IO
  - Network (ipv4 & ipv6)
  - Tasks (processes and threads)

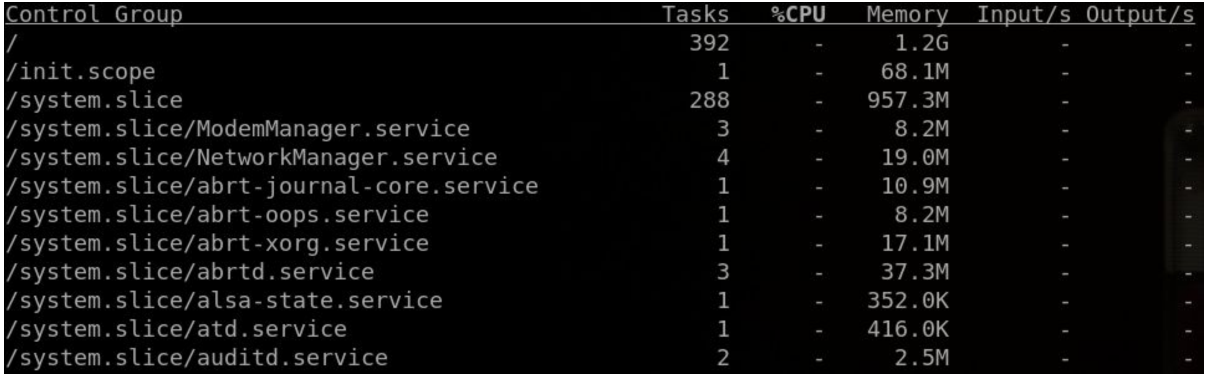


Systemd + cgroups: tooling

```
$ ps axwf -eo pid,user,cgname:50,args
```

Systemd specific tooling for working with cgroups

- `$ systemctl status name.unit`
- `$ systemd-cgls`
- `$ systemd-cgtop`



```
Control Group                                          Tasks   %CPU   Memory  Input/s Output/s
/                                                        392      -     1.2G        -        -
/init.scope                                                1      -    68.1M        -        -
/system.slice                                            288      -   957.3M        -        -
/system.slice/ModemManager.service                         3      -     8.2M        -        -
/system.slice/NetworkManager.service                       4      -    19.0M        -        -
/system.slice/abrt-journal-core.service                    1      -    10.9M        -        -
/system.slice/abrt-oops.service                            1      -     8.2M        -        -
/system.slice/abrt-xorg.service                            1      -    17.1M        -        -
/system.slice/abrtd.service                                3      -    37.3M        -        -
/system.slice/alsa-state.service                           1      -   352.0K        -        -
/system.slice/atd.service                                  1      -   416.0K        -        -
/system.slice/auditd.service                               2      -     2.5M        -        -
```

Systemd + cgroups: configuration #1

You can turn a resource accounting from either a unit or globally.

Unit file, [Service] section	**Global config (/etc/systemd/system.conf)**
`CPUAccounting=yes`	`DefaultCPUAccounting=yes`
`MemoryAccounting=yes`	`DefaultMemoryAccounting=yes`
`TasksAccounting=yes`	`DefaultTasksAccounting=yes`
`IOAccounting=yes`	`DefaultIOAccounting=yes`
`IPAccounting=yes`	`DefaultIPAccounting=yes`

Reload systemd after any config change: `$ systemctl daemon-reload`

Systemd + cgroups: configuration #2

Each of CPU/Memory/Task/… has its own set of options that **limit** that particular resource consumption. See for details:

```
$ man systemd.resource-control
```

In [Service] section:

- **CPUQuota**=20% ~ service will use max 20% CPU
- **CPUWeight**=[1..100..10000] ~ proportional CPU usage, 100 is default => 200 means twice the CPU cycles
- **MemoryHigh**=200M / MemoryHigh=20% ~ limit before service will be preferred to be swapped out
- **MemoryMax**=... ~ limit before service will be forced swapped and if that’s not enough, OOM killed
- **TasksMax**=N ~ maximum number of tasks (processes / threads) service can contain
- ...

Other unit types / activation modes

Unit files name.type that activates name.service when they are themselves activated.

- **Socket** ~ network connection activation. Defines a socket, once client connects to it, a corresponding service unit is started
- **Device** ~ HW activation. device units get created when HW is detected -> services can depend on them and get activated once HW is available
- **Mount** ~ mount a filesystem. Equivalent to a line from `/etc/fstab`
- **Path** ~ monitors a path (directory or file), triggers service on creation or change
- **Timer** ~ time-based activation. Activates after a specified interval relative to boot/unit activation or on a defined time/date
- `$ systemctl list-units --type=<type>`

Workshop

60 min

Workshop labs

- In the following next slides there are 5 labs total
  - 4 regular + 1 bonus
- Each lab has a time estimate how much time should you spend on it if everything goes well
- **We encourage you to help each other or rise your hand to get help from lecturers**
- If you couldn’t finish all labs during this class, you should complete them on your own later because learned skills will be used in following lectures or during a final practical exam.
- *HINT: focus on the lab content and leave any exploration or deep dive desires as a self-study for later*

Lab 1 - Create a new service [10 min]

Create a service that prints current date/time every minute:

```
$ while : ; do  date ; sleep 60; done
```

1. Create `/etc/systemd/system/my_service.service`
2. Use following options with correct values in appropriate sections ([Unit] vs [Service]):
   a. Description
   b. ExecStart
   c. Type
3. Start the service
4. Verify that it’s running
   ```
   a.  $ systemctl status my_service
   ```
5. Verify that it logs the time & date by observing its journal
   ```
   a.  $ journalctl -f -u my_service
   ```

Lab 2 - Make your new service start after boot [5 min]

Make the **my_service** service start after boot.

1. Stop the service if it’s running
2. Edit the service using `$ systemctl edit ...`
3. Make the service start after boot by having it `WantedBy multi-user.target.` This belongs to the Install section.
4. Enable the service: `$ systemctl enable my_service`
    a. This will make the ‘WantedBy’ option to take effect
5. Verify that the service has become part of the boot process (multi-user.target)
    a. By listing everything that is part of the multi-user.target
        i. `$ systemctl list-dependencies multi-user.target`
    b. Alternatively by looking at everything that your service is part of
        i. `$ systemctl list-dependencies --reverse my_service.service`
    c. By activating the multi-user.target and checking that the service has been started
        i. `$ systemctl start multi-user.target`

Lab 3 - Modify existing service [10 min]

Modify the **httpd** service so that it is automatically restarted if it crashes.

1. Start httpd service, check it’s status: has to be running
2. Kill all its processes, simulating a crash `$ killall httpd`
3. Check status, should be `inactive (dead)`
4. Edit its unit file, use `$ systemctl edit httpd.service` to not alter the original file but create a drop-in override file instead
    a. add `Restart=always` to the [Service] section
5. Restart the service
6. Verify with status that
    a. It’s running
    b. The drop-in file is listed
7. Kill all the processes again
8. Verify with status that it is still running (and PIDs are different)
9. Revert the changes with `$ systemctl revert httpd.service`

Lab 3/1 - Enforce correct service order [15 min]

Create 3 separate services called **a**, **b** and **c**. Make sure they are always started in the alphabetical order.

1. Create 3 new minimal services called a, b and c using the template from this slide (change the string after echo to reflect the name of the service)
2. Modify a.service to introduce delay:
   a. `ExecStart=bash -c "sleep 1; echo aaa"`
3. Enable all 3 services, making them part of the multi-user target
4. Test the services by activating the target they are part of
   a. `$ systemctl start multi-user.target`
5. Verify they were started by checking their journal
   a. `$ journalctl -f -u a -u b -u c |egrep '(echo)|(bash)'`
   b. Preferably keep this running in a separate terminal
6. Verify in the journal that the 3 echos are being printed out of order (a is last)

```
[Unit]
[Service]
ExecStart=echo aaa
Type=oneshot
[Install]
WantedBy=multi-user.target
```

Lab 3/2 - Enforce correct service order [15 min]

Alter the services so that they always start in alphabetical order

1. Add the correct **After**=name.service (and/or **Before**=name.service) options to the [Unit] sections of the 3 services to enforce alphabetical order
2. Test by rerunning them
   a.
   ```
   $ systemctl start multi-user.target
   ```
3. Verify correct ordering by looking at their logs
   a.
   ```
   $ journalctl -f -u a -u b -u c |egrep '(echo)|(bash)'
   ```
4. Clean up: disable the 3 services and delete their unit files

Lab 4 - Limiting CPU usage [10 min]

Create a service that is very CPU intensive, limit it to maximum 10% CPU.

1. Create service **`cpueater`**
2. Start the service, observe that it uses all available CPU
   a. openssl process in `$ top`
   b. cpueater.service in `$ systemd-cgls`
3. Edit the service, add to the [Service] sections options that will
   a. Turn on CPU accounting
   b. Set CPU quota to 10%
4. Restart the service
5. Verify the CPU usage again, should be 10% or lower
6. Stop the service, remove the unit file

```
/etc/systemd/system/cpueater.service

[Unit]
Description=Uses as much CPU as it can
[Service]
ExecStart=openssl speed
Restart=always
```

Bonus Lab 1 - Limiting Memory usage [?? min] 2 points

Similarly to previous lab: create a service that allocates at least X units of memory. Limit it using systemd to only be able to use X/2. E.g. let it allocate 1GB, limit it to 512MB.

- You need to come up with the tool/script/program/... that will consistently allocate set amount of memory
- You need to find out the correct systemd options that control memory usage.
    - Hint: check these slides or `$ man systemd.resource-control`