Setting up your dev environment

This document provides instructions for setting up an environment to work on the OS assignments. We provide instructions for Linux, macOS, Windows and Docker (cross-platform).

To summarize the situation per platform:

- **Linux** works natively (we run our tests on Ubuntu 18.04 and 22.04).
  - Other Linux distributions may show slight differences; Docker is an option.
- **Windows** is supported through WSL2 (optionally with Docker).
- **macOS** is supported only through Docker.
- **Docker** *can* be used on any platform (Linux, macOS, Windows) for all assignments.
- You can also use a Virtual Machine (e.g., VirtualBox running Ubuntu).

Linux

All assignments can be developed and tested natively on Linux. You will need to following dependencies:

For all assignments:	`gcc make python3`
Asg1:	`python3-pexpect libreadline flex`
Asg3:	`fuse libfuse-dev libssl-dev`

On Debian/Ubuntu you can install of of these with a single command:

```
sudo apt update && sudo apt install build-essential python3 \
     python3-pexpect libreadline-dev flex \
     fuse libfuse-dev libssl-dev
```

For other distributions packages will be named similarly.

After installing the dependencies, you can download and untar the assignments archive from Canvas, and then run the following command to compile it:

```
make
```

To run the automated tests, use the following command:

```
make check
```

If you run into any issues, remember you can always use Docker as well.

Windows

None of the assignments work on Windows natively. However, on Windows 10 you can use the official/built-in **WSL2** ("Windows Subsystem for Linux") to run Linux inside your Windows. WSL1 does **not** work for all assignments.
To set this up, follow the instructions from Microsoft:
https://docs.microsoft.com/en-us/windows/wsl/install-win10

We recommend choosing Ubuntu as a distribution (18.04 is what we test on, 20.04 should also work).

After installing WSL2, simply follow the instructions in the Linux section.

All assignments should work in WSL2, especially if you choose Ubuntu 18.04 as a distribution. However, you can also use Docker inside WSL2 if you run into any issues.

macOS

While some assignments may appear to (partially) work on macOS natively, none of the assignments can fully work on macOS, and we provide no support for running assignments on macOS natively. Instead, you should use Docker. To install Docker on macOS, follow the instructions at https://docs.docker.com/docker-for-mac/install/

After launching the Docker app for the first time, the `docker` command should be available in your terminal. For further instructions on Docker, refer to the following section.

Docker

Docker allows you to run containers with a pre-setup and deterministic environment. Docker is available on all platforms, and the best way to make sure your code works well during autograding. For OS, we provide a docker container for each of the assignments.

For each assignment, you can pull in the most recent version of the docker image using the following command. Note that you generally only have to do this once per assignment.

```
make docker-update
```

You can run the automatic tests on your code with the following command:

```
make docker-check
```

In most cases, this is all you will need to know about Docker. However, read on if you want to use Docker for more, such as interactively running and debugging your code inside a container.

Interactive Docker usage (for debugging)

You may also want to use docker for further development and debugging, in which case you may want to manually enter the docker container to run your own commands. Do do this, run something like the following command, in the directory containing your code:

```
docker run -it --rm --read-only -v "`pwd`:/code" -w /code \
      DOCKERIMG /bin/bash
```

Where **DOCKERIMG** is the assignment-specific image (e.g., `vusec/vu-os-shell-check`). Refer to the makefile of the assignment to double check (which defines `DOCKERIMG` somewhere at the start).

This command will start a container and launch a shell (bash) for you to use. The current directory on the host is mounted at /code. Any changes you make to files in this directory inside the container will be visible on the host, and vice-versa. **Any changes *outside* the /code directory will be lost when you exit the container!**

On **Linux** and **WSL2**, you may want to add the ``-u `id -u`:`id -g` `` flag, so files on your host are not suddenly owned by the root user/group. This is not required on macOS.

For assignment 3 you need to add the `--privileged` flag. More instructions are available in the assignment description there.

Multiple terminals with same Docker container

For manually testing some assignments (in particular, asg3) you may want to open multiple terminals in the same Docker session.

First start the container as usual:

```
docker run --privileged -it --rm -v "`pwd`:/code" -w /code \
      --name os_fs  vusec/vu-os-fs-check /bin/bash
```

Here we named our container “os_fs”, which we can then use to start another program from another terminal:

```
docker container exec -it os_fs /bin/bash
```

Advanced - Creating your own Docker image

If you plan to do all your development in Docker you may want to create your own image. This way, you can install additional dependencies in the image automatically, add additional files, and better fix the issue with file permissions (since the `-u` flag mentioned for Linux/WSL2 previously may not be ideal).

First, we need to create our own Dockerfile, which tells docker how to build our new image. Create a new directory called `docker_build_dir` and add a file called `Dockerfile` with the following contents:

```
# Use the asg1 (shell) docker image as a starting point
FROM vusec/vu-os-shell-check

# Install additional packages, like vim and gdb
RUN apt update && \
      DEBIAN_FRONTEND="noninteractive" \
      TZ="Europe/Amsterdam" \
      apt install -y \
      sudo vim-nox gdb

# Create a new user, matching the uid/gid of the host to fix files
# getting owned by root:root on Linux/WSL2.
ARG UID=1000
ARG GID=1000
ARG USER=user
RUN groupadd --gid $GID --non-unique $USER && \
      useradd --uid $UID --gid $GID --groups sudo --password '' \
      --shell /bin/bash --create-home --non-unique $USER
USER $USER
WORKDIR /home/$USER

# Copy in additional files from your host, e.g. some config file
COPY .bashrc /home/$USER/.bashrc

CMD /bin/bash
```

Tweak this file to suit your needs, e.g., add additional files or packages. Copy any host files to the docker_build_dir directory (such as the .bashrc in the example above). Then, run the following to build the image:

```
docker build --build-arg UID=$(id -u) --build-arg GID=$(id -g) \
      -t shell-dev-img .
```

The build-args are only required for the user creation part (for Linux/WSL issues). Your new image is now created with the name shell-dev-img. To run your image, similarly to the previous run command (run from the directory containing your assignment files):

```
docker run -it --rm -v "`pwd`:/home/user/host" -w /home/user/host \
      shell-dev-img
```

Advanced - Reusing the same Docker container

Every time you run the above Docker commands, a new container is created from the image. For development you may want to keep a single container around. You can do so with the following commands:

```
docker run -it -v "`pwd`:/home/user/host" --name dev-shell \
      shell-dev-img
# Press Ctrl-D to exit
docker start dev-shell
docker exec -it dev-shell bash  # To resume, use Ctrl-D again to exit.
```

Note that we omit the `--rm` flag, and name the container with `--name dev-shell`. This allows us to later resume the same container.