NTNU

Kunnskap for en bedre verden

Department of Computer Science

TDT4186 - Operating Systems

Lab 1

Processes & Schedulings

Professor:
Di Liu
Teaching Assistant:
Roman K. Brunner
Student Assistants:
Eik Hvattum Røgeberg
Halvor Linder Henriksen
Ole Ludvig Hozman

Handout: 19.01.2023

Introduction

Hello, and welcome to the first lab. In the labs you will have the opportunity to practically try out things you learned in the lectures and the theoretical exercises. The labs are compulsory coursework, you need to **complete three out of four labs** to sit in the exam. As the labs are mandatory coursework, they are also subject to NTNU's plagiarism rules. More on that in Section 3.

In order to help you get started, we encourage you to go to the lab sessions, which take place each week. The lab sessions are not mandatory, but the material discussed there is relevant for the exam. In those sessions we encourage you to ask question to the student assistants, but also to discuss with the tasks with other students. The format of these lab sessions generally follows the following structure:

1. In the week when a lab is released, we allocate time in the lab session such that you can sit together in groups and discuss what a potential solution could look like. The idea is not to start coding yet but to plan and think about the design of the piece of software that you are going to write. The motivation for doing so is that you can exchange ideas about how to solve the lab, but also to improve your skills with planning ahead of implementation. This should also help in understanding the problem before programming starts, which means that you spend less time trying out approaches that do not work. Additionally, these design meetings are also widespread in software teams in both academia and industry to create a common understanding of the problem, which usually helps the teams to come up with better solutions than each individual member of the team.

2. After the design meeting, we expect you to implement a solution on your own. Please do not copy code from the internet or colleagues, as this would constitute as plagiarism. For more details on plagiarism, see section Section 3. Nevertheless, we want to give you the opportunity to discuss and ask questions. During the lab sessions in the implementation phase, you can ask questions to the student assistants and fellow students. Additionally, the theoretical exercises will be discussed in those sessions.

3. For the lab session after the deadline, you get assigned the solution from one of your colleagues. Please read it carefully and try to identify things that could be improved, but also take note of things that were solved exceptionally well. During the lab session, you will go through the code of each other together, giving feedback and explaining the reasoning behind particular implementations. The motivation for that review session is that you get detailed feedback on your solution, you see another solution (and not only our reference solution) and hear about the attached reasoning, and you also learn to comment on someone else's code. Additionally, code reviews can help to spot errors that went unnoticed before, both in the implementation and the theoretical understanding of the problem or its solution. Those are also reasons why code reviews are prevalent in engineering teams.

Please sign up for one of the lab sessions under https://s.ntnu.no/LZZtgbcC. You need to be logged in with your NTNU account in order to access the file.

1 Task Description

In this lab, you will get to know XV6, a small Unix-like operating system that we will use as a sample for this course. XV6 has been developed for the course Operating System Engineering at MIT and is based on Unix V6[1]. We will be working with XV6 for the rest of the semester. More specifically, we are working with XV6 for RISC-V. RISC-V is a specific CPU architecture, which we will not dive into a lot further, but to execute it, we will use a virtual machine that provides a RISC-V system for us.

[1] If you want to know more about Unix and its place in history, see https://unix.org/what$_i s_u nix/history_timeline.html$

Before we jump into the tasks for the first lab, we hence go through the setup required to complete the labs.

1.1 Setup

To compile the operating system and the user-level programs, as well as run it, you will need a few tools installed. You can also use our Virtual Box system image from Blackboard, which already comes with all the preinstalled tools. If you want to install it locally, you find the steps to setup below[2].

1.1.1 Windows

If you did not yet set up the Windows Subsystem for Linux as described in the Exercise for C-Programming, head over to the C Programming Exercise sheet and follow the described steps there. After that, you can follow the below steps for Debian/Ubuntu.

1.1.2 MacOS

1. We first need to install the developer tools. For that, we open a terminal and execute the following command:

   ```
   $ xcode-select --install
   ```

2. Next, we need to install Homebrew (if you don't have that already). We use Homebrew to install the dependencies.

   ```
   $ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent
     .com/Homebrew/install/master/install)"
   ```

3. Now we can install the dependencies, which includes the RISC-V compiler toolchain[3].

   ```
   $ brew tap riscv/riscv
   $ brew install riscv-tools
   $ brew install qemu
   ```

   Check if you can execute the installed tools. If not, check where the binaries got installed and add the directory to `$PATH`.

1.1.3 Debian/Ubuntu

Open a terminal and install the toolchain using the following command

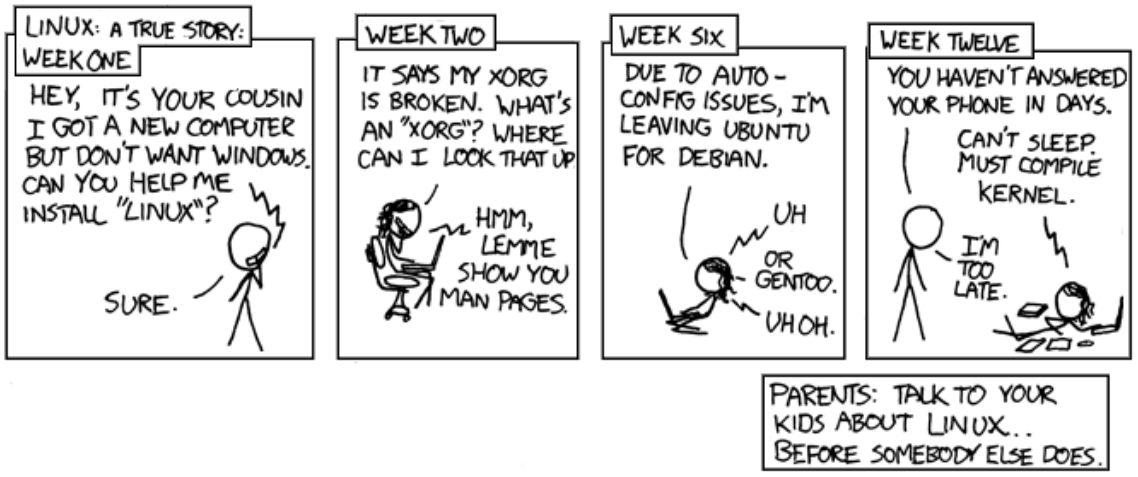
```
$ sudo apt update
$ sudo apt upgrade
$ sudo apt install git build-essential gdb-multiarch qemu-system-misc
$ sudo apt install gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu
```

1.1.4 Arch Linux

Similar to Ubuntu, open a terminal and update the packages. Then install the toolchain, using the following command

```
$ sudo pacman -S riscv64-linux-gnu-binutils riscv64-linux-gnu-gcc
  riscv64-linux-gnu-gdb qemu-arch-extra
```

[2]The setup follows the setup described under https://pdos.csail.mit.edu/6.828/2022/tools.html
[3]https://github.com/riscv-software-src/homebrew-riscv



"This really is a true story, and she doesn't know I put it in my comic because her wifi hasn't worked for weeks."
Source: https://xkcd.com/456/

1.1.5 Testing the Installation

Download the handout code for this Lab from Blackboard and unpack its content. In the directory, run

```
$ make qemu
init: starting sh
$
```

This should start a shell within xv6. If not, please check the version of the installed QEMU and GCC and check if the RISC-V versions are available.

If the shell appears (`$` symbol), you can try out xv6. To exit the virtual machine and XV6, press `Ctrl + A`, then `X`. This will shut down the virtual machine.

1.1.6 Debugging Kernel and Userspace

Debugging an operating system, in general, is not easy, especially during the beginning of the development, as not many options are available. Often we revert to print statements, which is considered a "poor man's" debugger.

A better option is to use either JTAG or a remote debugger. No matter which option we choose, we must first implement respective stubs. However, since we are working on a more mature operating system in this course, a remote debugger already exists. To debug the OS and any user-space program contained in the image, execute `make qemu-gdb`. This launches a debugger stub, opening a port to which we can connect with GDB, and writes a config file into the current directory. When starting GDB (e.g. `gdb-multiarch`) from within the project directory, it should use the config file and automatically connect to the port. If GDB warns that auto-loading has been declined, see https://sourceware.org/gdb/onlinedocs/gdb/Auto_002dloading-safe-path.html#Auto_002dloading-safe-path on how to solve the problem.[4]

To debug in the kernel space, you can start debugging right away. If you want to debug a user-space program, you first need to load that. See the sequence of commands below for an example.

```
(gdb) file user/_ls
    Reading symbols from user/_ls...
(gdb) b fmtname
```

[4] If you want to learn more about the GDB init file, see https://www.cse.unsw.edu.au/~learn/debugging/modules/gdb_init_file/

```
    Breakpoint 1 at 0x0: file user/ls.c, line 8.
(gdb) c
    Continuing.
```

From there on, it behaves as a normal GDB, so you can set breakpoints, check values and state of the program as you know it from a normal debugger.

1.2 Hello World

The first task in the first lab is to write a small user-space program that takes one optional parameter. This program should output the string `Hello World` if no argument has been supplied and `Hello [argument], nice to meet you!`, where `[argument]` should be replaced with the argument the user of the program provided. The `hello` program should only read the first argument and ignore any additional arguments.

Sample in/output:

```
$ hello
  Hello World
$ hello Roman
  Hello Roman, nice to meet you!
$ hello Roman Brunner
  Hello Roman, nice to meet you!
```

A word of caution: The C standard library has not been ported to XV6 (which is very common for small/research operating systems). Thus, you cannot assume that certain C standard headers exist. If you need specific functionality, you a) can check back with the existing code if something similar has already been programmed (e.g. `printf` exists - do you find out which header file you have to include?) or b) have to write the code yourself (e.g. if you require a certain data type).

Below are a few questions that you should be able to answer after completing this task:

- What do you have to do in order to compile a user space program on XV6?
- Which header defines the user-space `printf`?

A few additional questions, that you might be able to answer after completing the Hello World task are:

- Where is `printf` implemented? What is the call stack from your call to `printf` in your program to the printf implementation?
- Which syscall is required to print something to screen? How is a syscall called in XV6?
- How is the output represented? How would we change it if we want to output something to stderr instead of stdout? Would that be possible in its current state?

1.3 Process Management

For this task, we want you to write a small tool that helps us keep track of the currently running processes. In order to achieve that, it should list all processes that the OS is currently aware of in the following format: `proc_name(proc_id): proc_status`, one process per line. Implement all required parts and think about what should go in the kernel space and the user space.

Sample input/output of the program:

```
$ ps
  init (1): 2
  sh (2): 2
  sh (8): 2
  sh (9): 2
  ps (10): 4
```

If you find it difficult to start with the task, see the optional tasks. Some of them might help you to break down the problem into more manageable pieces.

1.3.1 Optional Tasks

With some of the tasks, we provide a list of optional tasks, some of which are easier than the original task and some of which are harder. The easier ones are thought to help you get started designing and implementing the main task, while the harder tasks are there for you if you want to have some extra challenge. We indicate the difficulty of the different tasks at the beginning. You don't have to do either of those tasks, as long as you complete the main task described above.

EASIER: Implement a "Hello World" syscall, that just prints a simple message when invoked from a user-level process. Have a look at other syscalls that are already implemented as a sample.

EASIER: Implement a user-space program that calls the "Hello World" syscall. If either the call on the user-level or something on the kernel-level fails, try debugging it with the remote GDB. You find a description of how to debug XV6 in Section 1.1.6.

EASIER: Implement a syscall that returns information about the current process (e.g. its process ID and state). Think about who is responsible for allocating the memory that contains the process information that will be returned and how it can be freed again after use (if it needs to be freed).

SIMILAR: How could one implement a syscall that returns an array or list of arbitrary length? Can you also come up with a fixed-length solution for getting an arbitrary number of results from a syscall (You can call the syscall multiple times)?

HARDER: The processes in XV6 also store information about the process tree. Can you build a tool that uses that information and prints the tree of processes? The binary should be called `proctree` and print out a tree in the following format: if a process is the child of another process it will be prefixed with `|-` and start at the same indentation as its parent process. If you have a look at the sample output below, you see for example that the first shell is a child of `init`. **NOTE:** This is a difficult task and not required to pass the lab. It also won't give you more points, but it will impress us and your colleagues ;).

Sample output:

```
$ proctree
  init (1): 2
  |-sh (2): 2
    |-sh (8): 2
      |-sh (9): 2
        |-ps (10): 4
```

2 Handing In

Before handing in, we advise you to use the command `make test`, which will test your implementation with a small set of test cases. If you want to see the tests in greater detail, you find the commands that were executed and the expected output in the file `test-lab-l1`. The test script also tests for some of the optional tasks, so you don't need to pass all the tests. We marked the

test cases that belong to the mandatory tasks with a tag `[MANDATORY]`. Try to make sure that all the mandatory tests pass before handing in.

After you tested and potentially debugged and fixed your solution, you can run `make prepare-handin` to create an archive of your solution. The archive will be written in the current directory and the file is called `lab-l1-handin.tar.gz`. Take that file and upload it to blackboard on the submission page for lab 1. You can submit multiple solutions, we are going to grade the last submission we received before the deadline.

Missing Deadlines

We expect you to start working early on the labs, to ensure submitting before the deadline. The deadlines are fixed and there is little we can do to move them. However, if you encounter a problem and notice that you can't make it in time, please contact us as soon as you notice. If you contact us early we have time to come up with a solution.

Last lab: Since we need to hand in a list of people that get to sit in the exam, handing in late for the last lab is not possible. We need a bit of time to grade the labs and if you hand in late, we cannot grade your assignment in time before the list must be handed over to administration.

3 Plagiarism

You must submit your **own work**. You must write your **own code** and **not copy** it from anywhere else, including your classmates, internet, and automated tools[5]. Failure to do so is considered plagiarism. Detailed guidelines on what constitutes plagiarism can be found at: https://innsida.ntnu.no/wiki/-/wiki/English/Cheating+on+exams.

We check all submitted code for similarities to other submissions and online sources. Plagiarism detection tools have been effective in the past at finding similarities. They have gotten excellent over time, so it is inadvisable to try and outsmart them. So don't do it, not only because we will most likely catch you, but because it is morally wrong and can undermine your academic integrity, even a long time into the future. For more references, see https://www.google.com/search?q=resigns+over+plagiarism+allegations (statistics on the 8th of August: 467'000 results).

[5] This is not an exhaustive list. Don't copy code from any source