Exercise in subject:	**DAT320 Operating Systems**
Title:	**Lab 1: Unix, programming tools and C**
Deadline:	**Tuesday Sep 2 2014 18:00**
Expected effort:	**20 hours**
Grading:	**Pass/Fail**
Submission:	**Individually**
Remarks:	Links in this document are clickable.

University of Stavanger

Faculty of Science and Technology

1 Introduction

The overall aim of the lab in this course is to learn how to develop systems, where some degree of low-level tuning is necessary to obtain the desired performance. We will do this through a series of lab exercises that will expose you to developing an application in the Go programming language, and some of the tools that people frequently use to tune such applications. Specifically you will learn about a performance (CPU/memory) profiler and race detector. In addition to developing this application, we will also require that you develop a Linux kernel module (a driver) that we may eventually use together with the Go application. This will also expose you to the challenges of interacting with Linux system calls written in C from the Go language. To develop the Linux kernel drivers, it is convenient to use a virtual machine running Linux, so you'll also learn a little bit about running with virtual machines.

2 The Linux Lab

Lab sessions will be held in the Linux lab in E353, where the machines are named: pitter1 - pitter40. All necessary software should be installed on these machines. Also, the lab project will include a networking part, requiring you to run your code on several machines, or at least from different ports on the localhost. This can be conveniently done using the machines in E353.

To be able to log into the machines in E353 you will need an account on the Unix system.

Task: *Get an account for UiS' Unix system, following the instructions on* ***user.ux.uis.no****.*

3 Remote Login with Secure SHell (SSH)

You can use `ssh` to log on to another machine without physically going to that machine and login there. This makes it easy to run and test the example code and your project later. To log onto a machine using `ssh`, open a terminal window and type a command according to this template, and make sure to replace username and hostname:

```
ssh username@hostname
```

For example to log on to one of the machines in the Linux lab, run:

```
ssh username@pitter18.ux.uis.no
```

This will prompt for your password. Enter your account password and you will be logged into that machine remotely.

You can avoid having to type the password each time by generating a public-private key-pair using the `ssh-keygen` command (see the man pages for `ssh-keygen`). Type

```
man ssh-keygen
```

and read the instructions. Then try running this command to generate your key-pair; make sure that once asked to give a password, just press enter at the password prompt. Once the key-pair have been generated, copy the public-key file (ends with `.pub`) to a file named `authorized_keys`. If you have multiple keys in the latter file, make sure not to overwrite those keys, and instead paste the new public-key at the end of your current file. After having completed this process, try `ssh` to another machine and see whether you have to type the password again.

Note that the security of this passphrase-less method of authenticating towards a remote machine hinges on the protection of the private key file stored on your client machine. Thus, it is actually recommended to create a key with a passphrase, and instead use the `ssh-agent` command at startup, along with `ssh-add` to add your key to this agent. Then, the `ssh`, `scp`, and other `ssh`-based client commands can talk locally with the `ssh-agent`, and you as the user only needs to type your password once. Please consult the `ssh-agent` and `ssh-add` manual pages for additional details.

Another tip: If you are running from a laptop and wish to remain connected even if you close the laptop-lid, you can check out the `mosh (click me)` command.

4 Basic Unix commands

To get a feeling for how to work with the Unix shell, we are going to try out several different commands. We will use the online tutorials one to eight found at here (click me).

Task: *Do the exercises from tutorials one to eight. (summarized in the lab1 project on github).*

For later lookup, a more condensed tutorial can be found here (click me).

Shell questions

1. What option with the command "rm" is required to remove a directory‘?
2. What is the command used to display the manual pages for any command?
3. What command will show the first 5 lines of an input file?
4. What command can be used to rename a file?
5. What option can we given to "ls" to show the hidden files?
6. What will the command "cat -n file" do?
7. What will the command "echo -n hello" do?
8. What command will display s list of the users who currently logged in in the system?
9. How do you change password on your account?
10. How can you list a file in reverse order?
11. What does the "less" command do?
12. With "less" how do you navigate?
13. What command will display the running processes of the current user?
14. What command can be used to find the process(es) consuming the most CPU?

5 The vi / vim editor

In later labs we will be working in a constrained environment where you will only have access to the *vi*-editor or the improved version *vim*. This editor can be a bit hard to get used to, but it is quite useful to have some hands-on experience with it, because it is almost always available on Unix systems. On very stripped down Linux systems, e.g. on embedded systems like routers, set-top boxes, typically the only editor available is *vi*. However, once you get used to it, *vi* is actually quite powerful, and many people prefer it, even over fully fledged IDEs for programming.

Task: *Walk through the information found here (click me).*

vi questions

1. How do we save a file in `vi` and continue working?
2. What command/key is used to start entering text?
3. What are the different modes the editor can be in?
4. What command can be used to place the cursor at the beginning of line 4?
5. What will *dd* command do (in command-mode)?
6. How do you undo the most recent changes?
7. How do you move back one word?

6 Git and GitHub

This course uses git and github for handins of lab exercises, and thus basic knowledge of the these two tools are needed when working on the lab assignments. The procedure used to submit your lab exercises is explained on the front page of lab1 on github. This section only gives you a few pointers to where you can find information to learn more about git and github.

Git Git is a distributed revision control and source code management system. Basic knowledge of Git is required for handing in the lab exercises. There are many resources available online for learning Git. A good book is *Pro Git* by Scott Chacon. The book is available for free at git-scm.com/book. Chapter 2.1 and 2.2 should contain the necessary information for delivering the lab exercises.

GitHub GitHub is a web-based hosting service for software development projects that use the Git revision control system. An introduction to Git and GitHub is available in this video: http://youtu.be/U8GBXvdmHT4. Students need to sign up for a GitHub account to get access to the needed course material.

7 The *C language*

In this lab we will first provide some basic introduction to the *C language*. You will learn how to build a small program in C, how to compile and execute it, and how to extend the program with libraries. You will also get to know the *make* utility and learn how to automate the process of building an application written in C. This is typically a little bit more difficult than doing the

same in Java, but since lots of legacy code out there rely on programmers with C knowledge, by completing this course, you will have an edge over your peers who doesn't know these tools.

Even though the *C language* is almost 40 years old, it is still one of the most important programming languages. This is mainly because it enables the programmer to do low-level bit manipulation and access memory in a way that most higher-level languages prevent users from doing. The *C language* is the primary language for write operating systems such as Windows, Linux, and MacOS X is also partially written in C. And for building embedded systems there is virtually no other viable language.

7.1 Hello world

To get started with C, it is customary to implement the good old *Hello, world!* program.

Task: *Start a terminal window on a lab-machine and create the file shown in Listing 1 using vim, and save it as* `hello.c`

Listing 1: Hello, world! program.

```
#include <stdio.h>

int main(void)
{
  printf("Hello, world!\n");
  return 0;
}
```

Compile the program by using the following command:

```
gcc hello.c -o hello
```

The `-o` flag lets you specify the output (executable) file name, `hello` in this case. Before continuing, inspect the current directory using the `ls` command to confirm that the file has been compiled. Run the command by typing `./hello` in the terminal window, which should give you:

```
Hello, world!
```

As will be explained in the introduction lecture before this lab, the `gcc` compiler does a lot of things behind the scenes.

Questions:

1. How do you use `gcc` to only produce the `.o` file? What is the difference between generating only the `.o` file, and building the `hello` executable done in the previous compilation above?

Returning to the compile to executable binary. Rewrite `hello.c` so that parts of the program only executes if compiled in `debug` mode. That is, it should output:

```
Debugging
Hello, world!
```

Questions:

2. Give the command for compiling with *debug* enabled instead of normal compilation for the two examples shown in Listing 2 and Listing 3. Explain how to turn debugging on/off for the two cases.

3. Give a brief pros and cons discussion for the two methods to add debug code shown in Listing 2 and Listing 3.

Listing 2: Conditional debugging

```
#include <stdio.h>

int main(void)
{
#ifdef DEBUG
    printf("Some␣smart␣debugcomment\n");
#endif
  printf("Hello,␣world!\n");
  return 0;
}
```

Listing 3: Selectable debugging

```
#include <stdio.h>

int debug = 0;

int main(void)
{
  if( debug )
  {
    printf("Some␣smart␣debugcomment\n");
  }
  printf("Hello,␣world!\n");
  return 0;
}
```

To examine the internals of an executable file, the linker can be told to generate a *map* file. The content of the *map* file contains several sections, including: `.text`, `.data`, `.bss`, and `.rodata`.

Questions:

4. Provide the command for generating the *map* file. Which of the `gcc` tools is responsible for producing a *map* file?
5. What is the content of each of the sections in a *map* file. Explain briefly.
6. Rewrite `hello.c` to produce entries in the *map* file for `.data`, `.bss`, and `.rodata`. Hint: This can be done by adding one variable for each type to the file.
7. Add the following function to `hello.c`: `double multiply(double x1, double x2)`, which returns `x1*x2`. Use `gcc` to generate an assembly code listing for the program, and examine the assembly code. What assembly instructions are used to do this? Repeat this task, but now replace `double` with `float`. Explain!

8 The 'make' tool

On Unix systems `make` is an important build tool. In particular, many programs written in C/C++ depends on `make`[1] to compile, build, and install software. For this part of the lab, run through the example found at http://mrbook.org/tutorials/make/

[1]For those of you familiar with Java, a popular tool for building large Java projects these days are `Maven`, which can also help to manage dependencies between libraries used by your project. Another popular tool for Java projects is called `Ant`, which is very similar to `make`.

Questions:

8. How does `make` know if a file must be recompiled?
9. Provide a `make` command to use a file named `mymakefile` instead of the default `makefile`.

8.1 Using Make: A simple example with library dependencies

In this section of the lab, we are going to develop a small C application that makes use of a library. The library is called: `lib1`. The dependency is illustrated here:

$$app \rightarrow lib1$$

A skeleton of the code is given in the following listings:

```
void main( int argc, char *argv[] )
{
  //Make a table of size give by an argument on the command line.
  //Fill the table with random numbers between MIN and MAX
  //MIN = 0.0 if not specified with another value on command line, i.e. optional argument
  //MAX = 100.0 if not specified with another value on command line, i.e. optional argument
  //Call tab_sort_sum() in lib1
  //Print out table and sum
}
```

```
double tab_sort_sum( double *tab, int tab_size )
{
  //Sort the table, return the sum and the sorted table
}
```

Task: *Write the two C files (main.c, l1.c) and prepare the corresponding makefile to compile an executable. You must also make corresponding* `.h` *files (l1.h): Make sure that dependencies are handled correctly. The makefile should also contain a 'TEST' target, where all C files are compiled and tested. You can choose if you want to use specific C files for testing, or if you add test code to the existing files. When executing the 'TEST' target necessary app's and lib's must be built and a test executed. A failing test should fail 'make'.*

Questions:

10. How do you implement an *include guard*, and why is it needed?

8.2 Tips

The following should be done to check that your `makefile` is correct:

- If you delete a single `.o` file. Does `make` rebuild only the necessary files?
- If you delete a `lib` file, does the rebuild work as it should?
- If you change a `.h` file (e.g. save it to refresh its timestamp), does `make` rebuild the correct files?

9 Deliverable

A *small* report where all questions are answered. In assignments where there are several programming steps, only the last need step need to be included. For example, only the last version of the *Hello, world!* program needs to be delivered.