# Operating Systems (OPS)

Lab class

Marc van der Sluys
HAN University of Applied Sciences
Arnhem, The Netherlands
http://han.vandersluys.nl

July 12, 2021

## Contents

# 1 Introduction

This document contains the Operating Systems (OPS) lab-class exercises. Before you start using it, you should check the OPS web page [1] to see whether a newer version is available. You can recognise the different versions by the date on the title page.

In this lab class, we will practice our Linux system-programming skills using the material we have treated in the lectures and as described in the lecture notes. We program in **plain C**, and use the Linux **system calls** in our programs.

In Operating Systems, we use the **Bash shell** ("command line") to write, compile and run our code. We compile 'by hand' (using `gcc`) or with a Makefile. In this course, we will use the **GNU Emacs**[1] text editor to write our code. Information on system calls and C functions can be found in the **man pages**. Finally, we will use `git` as our version-control system.

In order to pass the lab class, you show your result to the teacher. You demonstrate that the binary executable works, show *how* it works in the source code and answer questions to convince the teacher that you understand what you have done.[2] It is a good idea to have an exercise signed off soon after you finished it, so that your knowledge is still fresh. Once you have finished all eight exercises, and you have installed Arch Linux on your Raspberry Pi[3] as instructed, you have passed the lab part of OPS.

Note that the Operating Systems class requires 140 hours of work in total, while there are only 42 contact hours between teacher and students for the full-time course and 21 contact hours for the part-time course. This means that 70% (full time) or 85% (part time) of your work for this class is done *outside* of the classroom. Therefore, use your hours in lab class wisely. Much of the actual work must be done elsewhere. Ask your fellow students or use the internet if you get stuck. Use the lab class to solve issues that could not be solved in any other way, and to validate and sign off your results with the teacher.

# 2 Exercise 0: programming on the command line

We will write, compile and execute our programming exercises on the command line, using the Bash shell. We will use the Emacs text editor for writing, git for version control, and the man pages as a source of information.

## 2.1 The Bash shell

Chapter 4 of the lecture notes describes the basics of the Bash shell. Please read it before you start programming. More tips on how to make your life on the command line easier can be found in *Efficient use of the Linux command line in the Bash shell* [2].

## 2.2 Creating directories

Before we start writing our first C program for OPS, we will first create the necessary directories and a hello-world program. Ensure that you are in your *home directory*[4] and create a new directory called `exercises`, which will contain all our lab exercises:[5]

[1] not XEmacs or Aquamacs or any other flavour of emacs

[2] You are allowed to work together with your colleagues, as long as *you* understand and can explain what you have done. You will need this knowledge as well on the written exams, where you *cannot* collaborate.

[3] As described in the documents *Operating systems (OPS) Final assignment* and *Installing Arch Linux ARM on a Raspberry Pi*.

[4] Your home directory is in `/home/<username>/` and has the alias `~`. You can check your current directory with `pwd`. You can always jump (back) to your home directory by typing `cd ~` or `cd` without any argument.

[5] Note that $ is Bash's *command prompt*, which you do not need to type.

```
$ mkdir exercises
```

Now `cd` into the new directory

```
$ cd exercises
```

Here, we will create a subdirectory called `hello` and `cd` into it:

```
$ mkdir hello && cd hello
```

You can now check that you are indeed in the directory `/home/<username>/exercises/hello/` by typing `pwd`.

More information on working with directories is provided in Section 4.2.3 of the lecture notes and Section 3 of *Efficient use of the Linux command line in the Bash shell* [2].

## 2.3 Creating a source-code file using Emacs

Next, we will use the text editor Emacs to write a small *hello-world* program. See Appendix A to find out how to start Emacs, and to create a file and save it. Then type (or copy) the program shown in Code listing 1 and save it as `hello.c`.

Listing 1: hello.c: a *hello-world* program

```
1 // Hello-world program
2
3 #include <stdio.h>  // Needed for printf()
4
5 int main(void) {
6   printf("Hello world!\n");
7 }
```

Check the contents of the current directory:

```
$ ls -l     # Note l is a lowercase L
-rw-r--r-- 1 student student 115 Feb 5 17:22 hello.c
```

This shows you the file you just created, its permissions, ownership, size, the timestamp of its last change, and more.

See for more information on Emacs *Getting started with Emacs* [3] and the references in that document.

## 2.4 Compiling and running a C program

We use the *GNU C compiler* (`gcc`) to compile our hello-world program:

```
$ gcc -Wall hello.c -o hello
```

The option `-Wall` tells the compiler to print **all W**arnings. Whenever you see a warning (or error), you should read it carefully and repair your code until it compiles without warnings (or errors). The option `-o` specifies the name of the output file, in this case the binary executable.

You can check which files were created with `ls -l`. Note that the '`x`' in the permissions column that you have *execution* permission (as well as read and write) for the binary executable, but not for the C file.

The compiled binary executable can be run by prepending `./` to the name of the program:

```
$ ./hello
Hello world!
```

## 2.5 Using git

We will now create a git repository and add our source file hello.c to it. This repository will also contain our programming exercises. Therefore, we will go up one directory:

```
$ cd ..
```

You should check that you are now indeed in the directory `~/exercises/` before you continue. When we create the git repository here, all subdirectories, including `hello/`, will be part of it.

Creating and configuring a repository and adding and committing a file is described in Appendix B.1. Carry out those steps, and then return here. After setting up the repository, you can add the file hello.c by doing

```
$ cd hello
$ git add hello.c
```

or in a single command, with

```
$ git add hello/hello.c
```

Next, follow Appendix B.2 and push your code to GitHub.[6] This will ensure that you have a backup.[7]

See *Git for coworkers* [4] for more basic information on git and links to more detailed information.

## 2.6 Man pages

While developing our code, we will use the **man pages** to look up function prototypes of system calls and functions from the standard C library, header files that must be included and more. In Exercise 1, we will even copy a piece of code from a man page to use as a template for (a part of) our code. Section 4.3.1 of the lecture notes provides some more information and Section 7 of *Efficient use of the Linux command line in the Bash shell* [2] more details on the man pages and how to navigate them. In short, use `/` to search, `n` for the next hit, `g`/`G` to jump to the top/bottom of the page and `q` to quit. See Section 8.4 of the same document to see how to add colours to your man pages and make them easier to read and navigate.

## 2.7 Set up your environment

You are encouraged to use the example configuration files at [5] to enhance your Bash, emacs and git environments with colours, aliases and more.

[6] Alternatively you can use Bitbucket, where you will still have free private repositories for small projects after you leave the university (see Appendix B.3).

[7] You would not be the first student who loses her data and has to restart programming from scratch. In particular when you are using Virtual Box, a corruption of your disc image is easily obtained, in which case you may lose *all* your data.

# 3 Programming exercises

## 3.1 Exercise 1: Command-line parameters

In this exercise we will develop a C program that uses the **`getopt_long()`** function to parse **command-line options** and their **arguments**. The program must provide the following functionality:

1. When **no argument** is provided, information about all the options must be printed to `stdout`.
2. When the option **`-h`** or **`--help`** is specified, the same information about all the options must be printed to `stdout`.
3. When the option **`-f`** or **`--file`** is provided, with a required argument, the specified file is opened and its first line is printed to `stderr`.
   (a) before the file is opened, the program tests whether the file exists. If not, an error message is printed to `stderr`. The function **`perror()`** is used to print an error (see `man 3 perror`).
   (b) the file name must have the extension **`.txt`**. If not, an error message is printed to `stderr`.
4. When **`-e`** or **`--end`** is specified, the *last* line of the specified file is printed. The file must also exist and end in `.txt`.
5. When the option **`-v`** or **`--env`** is provided, a list of all **environment variables** is printed to `stdout`.

Read Sections 6.2 and 6.3 of the lecture notes for background information. A template for Exercise 1 is available on the OPS web page [1], so that you do not have to program everything yourself.

Steps:

1. Create the directory **`~/exercises/ex01/`** and `cd` into it.
2. Download the tarball **`template_01.tar.gz`** from the OPS web page [1] and save it in the new directory (*e.g.* using `wget http://han.vandersluys.nl/OPS/tarballs/template_ex01.tar.gz`).
3. Unpack the tarball with **`tar xfz template_01.tar.gz`** or **`tar -xvz template_01.tar.gz`**.
4. Add the file **`ex01.c`** to the git repository and commit your changes.
5. Remove the tarball.
6. Open `ex01.c` with emacs to have a look at the file.
7. Check `man 3 getopt` to see how the function `getopt_long()` works.[8]

A part of the functionality of the program is already provided in the functions in the template. However, `main()` is still empty. The template should compile and run, but do nothing. It is your task to write the `main()` function and make sure the program does what it is supposed to do. For this, you use the function **`getopt_long()`** and an array of structs of type **`option`**. The second example (in the section EXAMPLE) of `man 3 getopt` gives an example for `getopt_long()`.

[8] Typing `/EXAMPLE` Enter in the man page will get you to the examples section. Scroll down for the `getopt_long()` example.

Note that only the option `-c/--create` is fully implemented (for both the short and long options) in the example. Copy and adapt this option to write your program.

Hints (from `man 3 getopt`):

1. The first field of the struct **`long_options[]`** specifies the long option (without `--`), the second specifies whether an argument is wanted or even required, the third is `0` and the fourth specifies the short option (see the line with `create`). The last line of the struct contains four zeros to indicate that the struct is complete.
2. The third argument of the function `getopt_long()` is a string of all short options. A colon (:) after a character indicates a required option.[9] The short options in this string must match those in the `struct long_options[]`.
3. Note that some of the information is provided twice — both in the struct and the call.
4. The function `getopt_long()` parses the next command-line option at each call and returns a single character, which is identical to the short-option character (without dash). A `switch/case` structure is used to ensure that the proper action is taken for each (short) option.
5. If an argument is specified for an option (*e.g.* `file.txt` in `--file file.txt`), it is stored in the variable **`optarg`**.
6. Since we don't need the variable `option_index` from the man-page example, you can either declare it as an integer and leave it unused, or, more graciously, replace it with `NULL` in the `getopt_long()` call.

---

## 3.2 Exercise 2: Syntax check and multitasking

In this exercise we will develop a program that takes three command-line arguments and produces screen output using different print methods. The code tests whether each of the arguments have the correct type, and prints an error message if they do not. We then run different instances of the program concurrently and see what happens to the screen output.

### Exercise 2a: Download and read the code template

We will develop the program **`display`**. However, we do not have to start from scratch; in fact we only need to add the code that performs the syntax check for the arguments. You can download the tarball **`template_02.tar.gz`** from the OPS web page [1] and untar it in the directory `~/exercises/ex02/`.[10] Add all `.c` and `.h` files to git and commit them. Read the existing code carefully. We only need to adapt the file **`syntaxCheck.c`**. The program already has the following functionality implemented:

1. The program `display` prints a specified character to the terminal for a specified number of times, using a specified method;
2. The syntax of `display` is
   ```
   display <print method> <# of times> <character>
   ```
   Hence, the program can be called with *e.g.* `./display e 1500 A` to print the letter 'A' 1500 times using the method 'e' (see below);

[9] A double colon (::) indicates an optional argument, which we shall not use here.

[10] `wget http://han.vandersluys.nl/OPS/tarballs/template_ex02.tar.gz`

3. The print method must be one of **e**, **p** or **w**:

   **e:** Print using the shell command **echo**. The C function `system()` is used:
   ```
   (void) system("/bin/echo -n ...");
   ```
   where `-n` suppresses a newline, and the print character must be provided in the place of the dots. Note that this is already implemented.[11]

   **p:** Print one character at a time using the **printf()** function, without a newline.

   **w:** Print one character at a time using the **write()** function, without a newline.

See `man echo`, `man 3 system`, `man 3 printf` and `man 2 write` for more details.[12]

The main program calls the three functions implemented in the file **displayFunctions.c**:

**SyntaxCheck()**: checks the syntax and returns an error code. If an error was found, **DisplayError()** is called and the program exits. Otherwise **PrintCharacters()** is called.

**DisplayError()**: prints an error message and the correct syntax of `display`.

**PrintCharacters()**: prints the characters in a loop.

The prototypes of these functions are found in `displayFunctions.h`.

### Exercise 2b: Implementation of syntax check

The function `SyntaxCheck()` calls the functions **TestType()**, **TestNr()** and **TestChar()**, which you must write in the file **syntaxCheck.c**. These functions should perform the following tasks:

1. Check that the first argument (the print method) is a **single character**, and **one of** `e`, `p` or `w`;
2. Check that the number of times the character will be printed (the second argument) is a **positive integer**;
3. Check that the last argument, the character to be printed, is a **single character** (letter, number or symbol).

You should use the error codes that are defined in the file **errorCodes.h**. If one of these tests fails, a useful error message will be printed.

Hints:

1. Commit your changes after you have finished each function. Push your commits to the cloud when you stop working on OPS.
2. Use the **Makefile** to compile and link the code by typing **make**.
   - What happens when you type **make clean**?
   - And when you type **make** twice in a row?
   - What does **make -j4** (after `make clean`) do (see `man make`)?
3. Use the C-standard-library function **strtol()** (or **atoi()**) to convert a string to a (long) integer. See `man 3 strtol` for details.

[11] If the default implementation does not suppress newlines on your (Apple) system, try
```
char echoCommand[] = "bash -c \"echo -n  \""; // Note the two spaces after -n
echoCommand[17] = printChar;
```

[12] Note from the man section that `printf()` is a C-standard-library function while `write()` is a system call.

4. Use the C-standard-library function **`strlen()`** to determine the length of a string. Which header file do you need?

**Exercise 2c: Testing and running the program `display`**

1. What happens when you provide an incorrect argument for each of the three arguments?
2. Run **`./display e 1500 .`** What happens when, during execution:
   - you press **`Ctrl-C`**
   - you press **`Ctrl-Z`**
     - after `Ctrl-Z` type `jobs`
     - then type **`fg`**
     - what does `fg` stand for? See `man fg`
     - what happens when you type **`kill %i`** (instead of `fg`), where **`i`** is the number between square brackets returned by `jobs`?
3. Run **`time ./display p 100000 .`** and then **`time ./display w 100000 .`**
   - which is faster? Why?
   - what happens when you press **`Ctrl-C`** during one of these commands?

**Exercise 2d: Multitasking**

Start two instances of `display` as follows[13]

```
./display e 1500 .  &  ./display e 1500 +  &
```

What happens when you replace the ampersand (&) by a semicolon (;)? Why?

**Exercise 2e: Nice values and priorities**

The default **nice value** of a task is **0**. Higher nice values indicate lower priorities. We can increase[14] the nice value of a new process using the **`nice`** command (see `man nice`).

Start three instances of `display` as follows

```
nice -19 ./display e 1500 .    &   nice -10 ./display e 1500 -   &   ./display
e 1500 +  &
```

Explain what happens.

**Exercise 2f: The `printf()` and `write()` functions**

Repeat the three commands above, once with the **`p`** (`printf()`) and once with the **`w`** (`write()`) printing mode. If you don't see differences, print more characters. One of these functions uses **buffering**. Which function is that? How do you know? How does buffering affect the speed of the program?

---

[13] The extra spaces are unimportant, but added for clarity.
[14] Only root can decrease nice values.

### 3.3 Exercise 3: Forking tasks using `fork()` and `wait()`

In this exercise, we will fork off new tasks from an initial task. All tasks will undertake (slightly) different actions. Sections 5.4.2, 5.4.5, 7.2 and 7.3 of the lecture notes contain background information for this exercise.

#### Exercise 3a: Copy `ex02/`

We will start from our results in Exercise 2:

```
cd ~/exercises/ex02/
make -j4     # generate .o files
cd ..
cp -r ex02 ex03     # make a *recursive* copy of ex02/ called ex03/
cd ex03/
```

In Exercise 3, we do not need the file **`syntaxCheck.c`**:

```
rm syntaxCheck.c
```

Of course, you must add and commit the remaining files:

```
git add *.c *.h
git commit -m 'Initial commit for ex03'
```

#### Exercise 3b: Adapt the Makefile

Makefiles are like recipes; they describe how to *build a target* (prepare a dish) by listing the necessary input files or dependencies (ingredients) and describing how to obtain the target from these dependencies (*i.e.*, how to cook the ingredients to obtain the dish). In the case of Exercise 2, the binary executable **`display`** is the final target. It is built by linking three `.o` files using the system's default C compiler `$CC` (*e.g.* `gcc`). In turn, each `.o` file is created by compiling the corresponding `.c` file, using `$CC -c`. The `.c` files do not have to be created by **`make`**; we write them ourselves. Hence, the recipe is now complete (note that there are more recipes to clean the directory).

In this Exercise, we no longer need to build **`syntaxCheck.o`** — we will use the unchanged file from Exercise 2. Hence, we must remove the lines from the Makefile that describe how to build this file, and remove the file from the clean line so that it will not be deleted. Read the Makefile carefully and adapt or remove these lines:

- `syntaxCheck.o: ...` (2 lines)
- `clean: ...` (which `.o` files should still be removed?)

The code should now compile and clean properly, without overwriting or removing `syntaxCheck.o`. Test this before you continue(!) by running

```
$ make clean all
```

#### Exercise 3c: Use `fork()` and `wait()` in `display.c`

The new **`display`** must print a given number each of multiple ($N$) given characters to the screen, using a given nice increment. Hence, this version of `display` does what the version of Exercise 2 did, but in a single call rather than in $N$ calls.

The syntax of the new **`display`** shall be:

```
display <print type> <# of times> <nice increment> <char 1> [<char 2> ...
        ...  [<char N>]]
```

Note that the first three arguments keep their meaning, even when the total number of arguments can vary between runs.

We need to make the following changes in **`display.c`**:

1. Determine the number of different print characters the user has given as command-line arguments, and print it to the screen. Hint: from which variable can you compute this number?
2. Store the nice increment in the integer variable **`niceIncr`**.
3. Create a `for` loop over the integer **`iChild`**, which covers the number of different print characters. In each iteration (*i.e.*, for each `iChild`), the code must
   (a) fork off a child;
   (b) print a line (using `printf()`) with `iChild`, `iChild*niceIncr` and the `iChild`-th print character to screen to see whether the numbers are correct. Check whether your `for` loop works with this line only, before continuing. The output would currently look something like
   ```
   $ ./display  e 1 5  a b c
     0  0  a
     1  5  b
     2 10  c
   ```
   (c) assign the child with a nice value that is **`iChild*niceIncr`** higher than the default nice value — ensure that the first child uses the nice value 0;
   (d) print the `iChild`-th print character as often as specified using the function **`PrintCharacters()`**;
   (e) ensure the child exits and doesn't iterate the `for` loop.
4. The parent waits for **all** of its children to return, before it prints a newline character. Hint: use the **`wait()`** call as often as you have children.

Starting `display` as follows

```
./display e 1000 2  a b c d 1 2 3 4
```

should print the characters a–d and 1–4 1000 times each, using nice values of 0, 2, 4, . . . , 12, 14 respectively. Commit your changes when the code compiles correctly.

### Exercise 3d: Update the syntax checks

Next, we update the syntax checks done by `display` to the new syntax. You will have to edit the function **`SyntaxCheck()`** in the file **`displayFunctions.c`**. Note that one of the existing test functions can be used to test the syntax of the nice increment. Test your program and commit your changes.

### Exercise 3e: Check the nice value

After setting the nice value of a child, verify the value using the **`getpriority()`** system call and printing the result instead of `iChild*niceIncr`. Use `which=PRIO_PROCESS` and `who=0` (see `man 2 getpriority` for details). What happens if you assign a nice value larger than 19?

---

## 3.4 Exercise 4: Forking *different* tasks using `execl()`

In this exercise, we will fork off new tasks from an initial task. In contrast to the previous exercise, these children will load a different binary executable using **`execl()`** and perform a very *different* task than their parent. See Section 7.4 of the lecture notes for more information.

### Exercise 4a: Copy files from `ex03/`

We will start from our results in Exercise 3. Hence, copy the directory:

```
cd ~/exercises/
cp -r ex03 ex04
cd ex04/
```

Rename `display.c` to **`parent.c`** and adjust the `Makefile` accordingly.[15] Commit your files to git.

### Exercise 4b: Adapt the file `parent.c`

The new program **`parent`** must comply with the following demands:

1. `parent` shall have the same syntax as `display` in the previous exercise.
2. `parent` will fork off $N$ children. Each of the children will start the binary executable **`display`** from Exercise 2, **in the directory `../ex02/`** with the proper arguments, and with the nice value **`iChild*niceIncr`**.
3. Each of the $N$ instances of `display` shall print one of the specified characters multiple times. Hints:
   (a) use **`execl()`**. See `man 3 exec`;
   (b) in the first argument of `execl()`, specify the *relative* or *absolute* path of `display`;
   (c) the second argument of `execl()` will be available in `display` as **`argv[0]`**. What is the last argument?
4. An error message will be printed if the `exec` call fails. Use **`perror()`**. Hint: what happens after a successful call to `exec`?
5. Upon a syntax error, the child is stopped.
6. After all the children have started, the parent waits for their exit actions before printing 'All children have finished.' followed by a newline to standard output. When is this message printed if you comment out the `wait()` calls?

### Exercise 4c: Adapt the files `displayFunctions.c` and `displayFunctions.h`

In the files **`displayFunctions.c`** and **`displayFunctions.h`**, make the following changes:

1. Reduce the function **`SyntaxCheck()`**, to perform only the *necessary* checks: the number of parameters and the nice increment. The other parameters are checked by `display`.
2. Reduce the function **`DisplayError()`**. Only errors regarding the number of command-line parameters and the nice value must be printed here.
3. Remove the function **`PrintCharacters()`**. Note that this program is now no longer capable of printing characters itself.

[15] In `emacs`, press `M-%`, type `display` `Enter` `parent` `Enter`. Press `y` to replace an instance and go to the next. Press `n` to skip to the next instance. Press `.` to replace this instance and quit. Press `q` to quit.

**Exercise 4d: Test your program**

---

## 3.5 Exercise 5: Signals for synchronisation

In this exercise we will write two programs from scratch. The first program, called **`getsignal`**, will print a number every second. When a **signal** is caught, the number will be increased by one. We will send signals to `getsignal` from the command line, using the **`kill`** command.

The second program will be called **`sendsignal`**. It will send a signal to `getsignal` every three seconds, in order to change the number that program prints. See Chapter 11 of the Lecture notes for background information on signals.

### Exercise 5a: Write `getsignal`

Write a program called **`getsignal`** which prints a single-digit number (0–9) *unbuffered* to `stdout` every second. Use the **`write()`** system call for unbuffered output and print all numbers to the same line. You can use the **`sleep()`** function (see `man 3 sleep`) to generate the wait time of 1 s.

The program `getsignal` keeps printing the same number, until it receives signal number 25, upon which the number it prints will be increased by one. If the number was 9, it will become 0. Use the POSIX standard (**`sigaction()`**).

Hint: use characters for your numbers (`'1'` etc.). Note that `'5'+1` yields `'6'`.

### Exercise 5b: Test `getsignal`

Start the task `getsignal` so that it starts printing numbers. Open a different terminal and find the PID of `getsignal` using the command **`ps -a`**. You can send a signal with number **25** to `getsignal` using the **`kill`** command:

```
$ kill -25 <PID>
```

**Question:** What happens when you send signal number **9** instead of 25?

### Exercise 5c: Improve `getsignal`

Calling `ps` every time becomes tiring quickly. Hence, when `getsignal` starts, it should print its PID before it starts printing numbers.

### Exercise 5d: Write and test `sendsignal`

Write a program called **`sendsignal`**, which sends signal 25 to a given PID every **three** seconds. Use the functions **`kill()`** (see `man 2 kill`) and `sleep()`. The program should obtain the PID of `getsignal` from the command line:

```
$ ./sendsignal <PID of getsignal>
```

Test `sendsignal`.

#### Exercise 5e: Bonus: create a Makefile

Create a Makefile that compiles the two programs. You can take the Makefile from *e.g.* Exercise 2 as a template, and use your newly obtained Makefile for the next exercise.

---

### 3.6 Exercise 6: Synchronisation and communication using FIFOs

Here we will improve upon the previous exercise. Rather than passing the PID from one process to the other manually, we will use a **FIFO** (named pipe). Before you start, read Section 9.4 in the Lecture notes.

#### Exercise 6a: Copy and adapt `getsignal` and `sendsignal`

Copy the programs **`getsignal.c`** and **`sendsignal.c`** from the previous exercise, and rename them to **`getsignal_sendpid.c`** and **`sendsignal_getpid.c`** respectively.

Create a FIFO (named pipe) called **`PIDpipe`** on the command line using the command **`mkfifo`** (see `man mkfifo`). How can you recognise the pipe from the output of **`ls -l`**?

The program `getsignal_sendpid` should open the FIFO and write its PID to it before it starts printing numbers. The program `sendsignal_getpid` should not read any command-line parameters, but instead obtain the PID of `getsignal_sendpid` from the FIFO. Both programs should close the FIFO when they no longer need it, but not remove it.

Test `getsignal_sendpid` and `sendsignal_getpid` by starting them a few seconds after each other. Why does `getsignal_sendpid` wait for `sendsignal_getpid`? What happens if you change the order in which you start them?

---

### 3.7 Exercise 7: Redirection between parent and child using pipes

In this exercise, we will write the program **`redirect`** from scratch, which will **redirect** input and output to and from a child program called **`filter`** using (unnamed) **pipes**. The program `filter` is provided on the OPS web page. Read Section 9.2 and 9.3, and study Figure 9.3.3 of the Lecture notes before you begin.

#### Exercise 7a: Study the program `filter`

Download[16] and unpack the tarball for this exercise from the OPS web page [1], and compile the code using the `Makefile`. The `Makefile` will compile **`filter.c`** and produce an error when trying to compile `redirect.c`, because that code has not been written yet. Start the program **`./filter`** in the foreground, type some text containing upper- and lower-case characters and press `Enter`. The program `filter` will filter all lower-case characters and convert them to upper case. Type another line to filter more text. In order to quit, pass an `ESC` character (by pressing `Ctrl-[`) followed by `Enter`. Study the code until you understand how it works. The ASCII code for `ESC` is `0x1B`.

[16] `wget http://han.vandersluys.nl/OPS/tarballs/template_ex07.tar.gz`

### Exercise 7b: Design the program `redirect`

The program **`redirect`** will do nothing else than start `filter` and provide the redirection of data using (unnamed) pipes. It will redirect its input from its **`stdin`** (the keyboard) to the **`stdin`** of `filter`, and redirect the `stdout` from `filter` to its own `stdout` (the display).

Study Figure 9.3.3 in the Lecture notes and draw a similar schematic diagram for the case of `redirect` and `filter`. Note that more pipes are needed in the current situation.

### Exercise 7c: Write the program `redirect`

Write the program **`redirect`** and call the source file **`redirect.c`**, so that it can be compiled using the `Makefile`. The program should start the binary executable `filter` as a child. Create the necessary pipes, and ensure that they are properly redirected, according to your design.

Since `filter` processes data character-by-character, `redirect` should do the same, using the same system calls for reading and writing. The program `redirect` should exit when an `ESC` character is read.

**Question:** how can you demonstrate that it is really `redirect` which is writing to `stdout`, and not `filter`?

---

## 3.8 Exercise 8: Multithreading, a shared queue, mutexes and Valgrind

In this exercise, we will add **multithreading** to the queue code from an old OPS exercise to create the program **`sharedQueue`**. In the new program, four different threads (three producers and a consumer) will use the shared queue, and the critical actions will be protected by a mutex. Note that we do **not** use shared memory here.

### Exercise 8a: Download the code template from the OPS website and study it

In this exercise we start with an implementation of a *singly-linked circular list* or **queue**. Download[17] the code template from the OPS web page [1], and commit it to git.[18] Read the code carefully and make sure you understand how it works. Also make sure that the template compiles and runs on your system before you start coding. Do you understand the output?

You can use the Makefile that comes with the code for compilation. Note that the option **`-g`** is specified. This option tells the compiler to generate extra *debug information* in the binary executable. The program **Valgrind**, which can o.a. *profile* code and find *memory leaks*, uses this information to generate more detailed messages (*e.g.* including the code line number).

### Exercise 8b: Memory leaks and Valgrind

In order to detect *memory leaks*, we will use the program **Valgrind**[19] to run and analyse your code in the following way

```
$ valgrind --leak-check=full ./queue
```

How can you determine whether a memory leak is present?

[17] `wget http://han.vandersluys.nl/OPS/tarballs/template_ex08.tar.gz`

[18] This example is based on [6]. The interface is similar to that of the C++11 STL class *queue*. Function names have the word `Queue` appended, because C does not have function overloading.

[19] Check whether Valgrind is installed on your system — if not, use your package manager to install it.

### Exercise 8c: Adding and removing memory leaks

Create a memory leak in your code, editing only your **`main.c`**. Use Valgrind to show that you succeeded, and that you can remedy this problem. Hint: which queue functions that you call allocate memory? Which queue functions free it? Which C standard-library functions are used to allocate and free memory? Make sure that you can create and remove a memory leak by commenting out and in a (few) line(s) of code in `main.c`. Commit your changes.

### Exercise 8d: Adapt the code to make it multithreaded

Copy the file `main.c` to **`sharedQueue.c`**:

```
$ cp main.c sharedQueue.c
```

From this point on, you should **not** edit `main.c` anymore. Adapt the Makefile so that it uses `sharedQueue.c` rather than `main.c`. Also, make sure that the binary executable will be called **`sharedQueue`**. The program should start the following threads:

1. a thread that **writes data** (*i.e.* a node) to the shared queue every **2** seconds. This is a **producer thread**;
2. a **producer thread** that writes data to the shared queue every **3** seconds;
3. a **producer thread** that writes data to the shared queue every **4** seconds;
    - you should use the **same thread function** for all three producer threads. Hint: use different arguments when starting your threads to pass data and production intervals;
    - the producer threads do **not** produce screen output.
4. a **consumer thread** that **reads data** from the shared queue every **15** seconds, **appends** these data to a **file**, **prints** them to **`stdout`** and **empties** the queue. In the interval between two consumer iterations, **no output** is shown on the screen.

Your program must meet the following requirements:

- Ensure that the **data are different** for each thread, so that we can recognise which data come from which producer. Hint: pass these as arguments when starting your threads.
- Use the C function **`sleep()`** in a loop to generate the desired production and consumption intervals. Hint: these loops must be in the thread functions.
- Use a mutex or mutexes to guard the critical actions. How many mutexes do you need?
- When **`Ctrl-C`** is pressed, all threads must finish their last cycle, rather than terminate immediately. In the end, the consumer saves the last data produced and also quits. This will require a **shared variable** that is accessible to all threads.
- The **`main()`** function in `sharedQueue.c` only installs the SHR and creates and joins the four threads.

Hint: you should do most of your coding in **`sharedQueue.c`** (and you will have to change it drastically), and only very little in the other source files.

# Appendices

## Appendix A Basic use of Emacs

I treat GNU Emacs [7] here, not XEmacs, Aquamacs or any other flavour.

You can open a text file by specifying its name as an argument:

```
$ emacs hello.c
```

If a graphical version of emacs and graphical desktop are available, it will open in a new window and you may want to append an ampersand (&) in order to start it in the background. You can force the terminal version of emacs with the option `-nw`.

A default Emacs screen consists, from top to bottom, of:

1. The menu (if shown);
2. The main text-edit panel (called *buffer*);
3. A status line;
4. A *minibuffer*, where commands show up (usually one line).

The menu can be accessed (and exited) with F10, and navigated with the arrow keys and Enter. For most options, the keyboard shortcut is shown, so that you can quickly learn it.[20] Once you know the shortcut, you'll probably never use the menu again. Note that only a small fraction of Emacs's functionality is available through the menu!

The most important key strokes to know in order to start learning Emacs are:

- F10 Access or exit the **menu**;
- C-/ **Undo** text edit;
- C-g **Cancel** current command (press more than once if needed);
- Esc Esc Esc **Cancel everything** and go back to edit mode;
- C-x C-c **Exit** Emacs, prompting to save changed buffers.

Here, C-/ stands for Ctrl /, *i.e.* holding Ctrl whilst pressing /, then releasing both.[21] Similar combinations exist for Alt, which is written as M-.[22] Most commands have a (long) name as well as a shortcut, which can be typed using M-x, followed by the name of the command and Enter. Tab-completion works while typing the command. For example Undo can also be achieved by M-x `undo` Enter.

The Undo function in Emacs is different from those in many other programs. Consider for example the case where you type `a`, remove it, and then type `b`, then press Undo, then type `c`. If you then start pressing Undo, many programs will never recover `b` — that version of the document has been lost forever. Instead, when pressing C-/ repeatedly, Emacs will Undo *exactly* what happened before, including undoing previous Undo's. While perhaps unorthodox, this means that you will always be able to get back to any previous version by pressing Undo (within the limits of your Undo history).

[20] The same is true for gvim.

[21] For C-x C-c you can hold Ctrl while pressing X and then C.

[22] Pressing and releasing Esc has the same effect as pressing and holding Alt.

This information is an excerpt from *Getting started with Emacs* [3]. See that document for more information on how to get started with Emacs and for more references. A basic Emacs configuration file can be found at [5].

# Appendix B Getting started with git and GitHub or Bitbucket

## B.1 Getting started with git

Before we start programming, we set up the directories needed for the first exercise, and a git repository. First, ensure that you are in the correct directory. Next, we create a `git` repository in the current directory (which will therefore contain all our `.c` and `.h` files):

```
$ git init .
```

If you haven't used git before (as this user on this system), you have to supply a user name and email address. If this is going to be a *local* repository only, you can make something up. However, for OPS we will push our commits to a *remote* repository, for example for backup purposes or to share them with others. Hence, you should choose a name and email address that others are allowed to see.

```
$ git config --global user.name "<your name>"
$ git config --global user.email "<your@email.addre.ss>"
```

Leave out the option `--global` to use these settings for this repository only.

It is good practice to add a source file to the repository as soon as you have a first version. Note that this first version does not have to do anything spectacular yet, but it should compile and run correctly!

Let's add (and then commit) the source code for hello.c (assuming we are in `~/exercises/hello/`):

```
$ git add hello.c  # Add file to repo
$ git commit -m "Initial commit of hello.c"  # Commit changes
```

If we later edit this file, we will:

```
$ emacs hello.c  # Edit your source code
$ gcc -Wall hello.c -o hello && ./hello  # Ensure your code compiles and runs
$ git diff  # Check your changes before committing them; exit with q
$ git commit -a -m "Add useful feature foo to hello"  # Commit all your changes
$ git log  # Check your commit history; exit with q
```

If you don't specify a commit message with `-m`, git will open the editor specified in the environment variable `$EDITOR`.[23] Write a single line (or a single line as title, a blank line and below that more details), save it and exit your editor. See *Git for coworkers* [4] for more basic information and links.

A longer introduction into git and version control can be found in GitHub's Git handbook [8]. Introductions to related topics can be found in the GitHub guides [9].

## B.2 Using GitHub with git

If you want free, private (*i.e.*, others can't see it unless you want them to) `git` repositories, you can use GitHub on a student account.[24] I use private repositories as backup and/or to share my code with collaborators and public repositories for open-source projects. At the end of a day, GitHub can show me what I achieved that day.

[23] You can set this variable with `EDITOR=emacs` and check the result with `echo $EDITOR`.

[24] See [10] to apply for a student pack using your school email address. You can do this after creating an account.

Before you create an account at GitHub, find an email address that you are willing to share with others. Your email address is how GitHub recognises you. When signing up, give your full name (or as much of it as you want to share) and make up a good password — you'll have to type it each time you *push* your commits to GitHub.[25,26]

Once your account has been set up, you are ready to create your first repository. In order to do so, in your home screen, click the **plus** in the top right of the page, choose **New repository**, choose a good (not too long, without spaces) **name**, add a description for your repo, choose a **private** repository, **do not** initialise the repository with a README, **do not** add a `.gitignore` or licence file, and click **Create repository**.

GitHub will now show you how to push an **existing repository** from the command line:

```
$ git remote add Server git@github.com:<user>/<repo>.git
$ git push -vu Server master
```

The command **`git remote add`** adds a remote location called `Server`. I usually use the name `Server` (or `GH` if I intend to push my commits to multiple remote locations) rather than `origin`.[27] The **`git push`** command pushes your master branch to `origin` (or `Server`). You have to specify the name of the branch only once — after this

```
$ git push
```

suffices (though I add the option `-v` to see what happens in more detail).

After pushing your commits to the GitHub server, reload their web page with instructions to see your code.

The opposite of pushing, *i.e. pulling* changes that others made from the remote location to your local repository, is then simply done with

```
$ git pull
```

If you want to share your private repo with other GitHub users, go to the page of the desired repository, click **Settings** (top bar, right) / **Collaborators** (left menu), type and select the **user name**, and click on **Add collaborator**. The other user will receive an email and can now **clone** (*i.e.* do an "initial pull") your repo and push their changes. See [11] for detailed help.

### B.3 Using Bitbucket with git

Using Bitbucket is similar to using GitHub. Hence, read that section first, create an account on the Bitbucket website, and come back here for the Bitbucket specifics.

In order to create a repository on Bitbucket, in your home screen, click the **plus** in the blue bar on the left, choose **Repository**, choose a good (not too long) **name**, answer **NO** to the question whether to include a README, ensure that your repository is **private** and uses **Git**, and click "Create repository".

If you already have a local repository, **do *NOT* get started "the easy way"**, but follow the two steps/three commands under **Get started with command line / I have an existing project**:

```
$ git remote add Server https://<user>@bitbucket.org/<user>/<repo>.git
$ git push -vu Server master
```

[25] Unless you use an ssh key. We will encounter ssh keys when setting up an Arch Linux system.

[26] As an intermediate solution, use the credential helper: `git config --global credential.helper 'cache --timeout=3600'` will remember your credentials for an hour (3600 s).

[27] The name `origin` for the GitHub server doesn't make sense when the repository originates on my laptop.

After pushing your commits to the Bitbucket server, reload their web page to see your code.

If you want to share your private repo with other Bitbucket users, click **Settings** / **User and group access**, type the **user name**, select **Read** or **Write** and click on **Add**. The other user will receive an email and can now **clone** (*i.e.* do an "initial pull") your repo, and, if given write permission, also push their changes.

# References


[1] **van der Sluys, M.** *OPS web page*. URL `http://han.vandersluys.nl/OPS/`. Visited 2018-03-12.

[2] ——. *Efficient use of the Linux command line in the Bash shell*. URL `http://eubs.sf.net`. Visited 2016-08-10.

[3] ——. *Getting started with Emacs*. URL `http://han.vandersluys.nl/?title=Publications`. Visited 2017-11-13.

[4] **AstroFloyd**. *Git for coworkers*. URL `https://astrofloyd.wordpress.com/2012/12/16/git-for-coworkers/`. Visited 2017-08-29.

[5] **van der Sluys, M.** *Environment settings (bash, emacs, git)*. URL `https://github.com/MarcvdSluys/han-ese-ops-env`. Visited 2019-02-11.

[6] **Ahmad, N. & Jawawi, D.** *Data structures and algorithms: 11b - Queue - Linked List Implementation*. URL `http://ocw.utm.my/course/view.php?id=31`. Visited 2018-03-12.

[7] **Stallman, R.** *Emacs*. URL `https://www.gnu.org/software/emacs/`. Visited 2016-03-20.

[8] **GitHub**. *Git handbook*. URL `https://guides.github.com/introduction/git-handbook/`. Visited 2020-04-10.

[9] ——. *GitHub guides*. URL `https://guides.github.com`. Visited 2020-04-10.

[10] ——. *Applying for a student developer pack*. URL `https://help.github.com/articles/applying-for-a-student-developer-pack/`. Visited 2018-12-18.

[11] ——. *Inviting collaborators to a personal repository*. URL `https://help.github.com/articles/inviting-collaborators-to-a-personal-repository/`. Visited 2018-12-18.