PB152cv Operating Systems Seminar

Petr Ročkai

Part A: Preliminaries

The document in front of you is a printable version of the study materials used in the operating systems seminar. The seminar covers basic user-level interaction with a POSIX-compatible operating system and the basic programming interfaces thereof (i.e. the C API, as described by POSIX). It is probably a good idea to keep a computer at hand when following this text.

Part A.1: Intro

1. We show commands starting with `$`: this is the prompt. Do not type it.
2. To look at this text, connect to `aisa.fi.muni.cz` and type:

```
$ pb152 update
$ cd ~/pb152/info
$ cat intro.txt
```

3. If you see this in your terminal, you got it right.
4. One of the most useful UNIX commands ever is `man`.
5. To find more about `man`, type

```
$ man man
```

6. You will get a pager, which allows you to read longer text in a fixed-size terminal. You can use arrow keys on your keyboard, or pgup and pgdown to move about. To learn more about the pager, type

```
$ man less
```

7. Manual pages are divided into sections. First comes a very short description of the command (or C function), and then a Synopsis: how is the command (function, etc.) written when you want to use it. Usually, all options are listed (there can be quite a few!).
8. After Synopsis, you usually get a 'Description' which explains what the command does in more detail.
9. You are not forbidden from using Google, but `man` is often more efficient.
10. If you want to learn more about `cat`, you can type:

```
$ man cat
```

11. Otherwise, you can continue on to:

```
$ cat grading.txt
```

Part A.2: Rules of the Game

1. First of all, I am very sorry in case you lost.
2. The seminar is worth 2 credits and there is no exam. Instead, you will collect points as we go and when you have 15 of those, you pass. There will be various ways to collect those points. There is only one additional requirement: 2 of those points must be for activity (see below).
3. For starters, you can obtain 1 point by coming to the seminar 4 times (only whole points are awarded: 0–3 times = 0 points, 4–7 times = 1 point, 8–11 times = 2 points, 12–13 times = 3 points).
4. Each week, there will be a simple homework (only slightly more complicated than the exercises in the seminar). Working it out and handing it in will get you one point. More about this later.
5. You can collect 2 points for writing feedback (peer reviews) on your classmates' exercises. But only if you passed those exercises yourself! Each instance is worth 0.25 points.
6. There will be 4 'student' exercises in each seminar, except weeks 7 and 11, in which there will be more. Either you or one of your classmates will solve each of those (see `activity.txt` for details). When you do this, you earn 1 point each time (there's a limit of 12 points).
7. You can come to a different group in any given week (if there is room – ask the seminar tutor beforehand). This counts towards attendance, but you will be unable to collect activity points there.

The summary of obtainable points:

- max 12 points for homework
- max 6 points for meeting early deadlines
- max 3 points for attendance
- max 2 points for peer review
- min 2 points for activity (max 12)

A.2.1 Homework

Each homework is a simple extension of what was done in the seminar. There is a soft deadline one week after each assignment is given – if your solution passes the tests at this first deadline, you get 1.5 points for the exercise. If you fail, you can try 4 more times, one week apart and still get 1 point. More details about the homeworks can be found in `homework.txt`.
The schedule is as follows:

	given	try 1	try 2	try 3	try 4	try 5
hw01	19.2.	26.2.	4.3.	11.3.	18.3.	25.3.
hw02	26.2.	4.3.	11.3.	18.3.	25.3.	1.4.
hw03	4.3.	11.3.	18.3.	25.3.	1.4.	8.4.
hw04	11.3.	18.3.	25.3.	1.4.	8.4.	15.4.
hw05	18.3.	25.3.	1.4.	8.4.	15.4.	22.4.
hw06	25.3.	1.4.	8.4.	15.4.	22.4.	29.4.
hw07	1.4.	8.4.	15.4.	22.4.	29.4.	6.5.
hw08	8.4.	15.4.	22.4.	29.4.	6.5.	13.5.
hw09	15.4.	22.4.	29.4.	6.5.	13.5.	20.5.
hw10	22.4.	29.4.	6.5.	13.5.	20.5.	27.5.
hw11	29.4.	6.5.	13.5.	20.5.	27.5.	3.6.
hw12	6.5.	13.5.	20.5.	27.5.	3.6.	10.6.

NB. The 10th of June is also the final deadline to collect the required points: peer reviews will no longer be accepted after this date either.

A.2.2 Plagiarism

Please work on your own. If you get caught cheating, you will fail the course and you will have to explain yourself to the disciplinary committee. You are also responsible for keeping your solutions private. If you only use the `pb152` command on `aisa`, it will make your `~/pb152` directory inaccessible to anyone else (this also applies to school-provided UNIX workstations). Keep it that way. If you work on your solution using other computers, make sure they are secure. Do not publish your solutions anywhere (on the internet or otherwise). All parties in a copying incident will be treated equally.

A.2.3 Summary

- 15 points to pass
- 2 out of the 15 must be for activity
- get them by the 10th of June

That's the rules. We can now start with the seminar proper.

```
$ cd ../01
$ cat intro.txt
```

To share your screen, use the following command:

```
$ pb152 beamer
```

To follow on your own screen, use:

```
$ pb152 follow
```

In case you are visiting (i.e. not enrolled in this group) use:

```
$ pb152 follow --login xlogin
```

where xlogin is the login name of the student whose screen is being beamed. You can often see this login name on the beamer (part of the shell prompt). If unsure, please ask.
More detailed instructions are available in ../01/activity.txt.

Part A.3: Assignment Guidelines

The general principles outlined in this section apply to all assignments. The first and most important rule is, use your brain – the assignments are not exhaustive and sometimes leave room for different interpretations. Pick the most reasonable. Do not try to find loopholes: technically correct is not the best kind of correct.
Do not print anything that you are not specifically directed to. Solutions which print garbage (i.e. anything that wasn't specified) will fail tests.

A.3.1 Submitting Solutions

The easiest way to submit a solution is this:

```
$ cd ~/pb152/01
<edit files until satisfied>
$ pb152 submit
```

The number of times you submit is not limited (but see also below). NB. Only the files listed in the assignment will be submitted and evaluated. Please put your entire solution into these existing files.
You can check the status of your submissions by issuing the following command:

```
$ pb152 status
```

In case you already submitted a solution, but later changed it, you can see the differences between your most recent submitted version and your current version by issuing:

```
$ pb152 diff
```

The lines starting with - have been removed since the submission, those with + have been added and those with neither are common to both versions.

A.3.2 Evaluation

Your solutions are tested once a week (see grading.txt for the deadlines), always at 10pm (on Wednesday). This deadline is common for all seminar groups. There is a set of automated tests that decide whether you passed or not. If you fail and do not understand why, talk to your seminar tutor after the seminar and they should be able to help you. Alternatively, you can ask on the discussion board in the IS (but we cannot guarantee a timely response: please allow for a day or two of delay).
Only the most recent submission is evaluated, and each submission is evaluated at most once. You will find your latest evaluation results in the IS in notepads (one per assignment).

Part A.4: Peer Reviews

You can optionally participate in peer reviews, both as a reviewer and as a review recipient. Reviewers get points for their effort, the recipients do not, but instead get (hopefully) useful information.

A.4.1 Requesting Reviews

If you would like to have your code reviewed, you can issue the following command:

```
$ pb152 review --request 01
```

Substitute other assignments for 01 as appropriate. You can request a review on an assignment which you did not pass yet. You may get up to 3 reviews for any given request. The reviewer will work with the submission that was current at the time they agreed to do the review. Make sure you submit the code you want reviewed before requesting the review.
The pb152 update command will indicate whether someone reviewed your code, by printing a line like A reviews/01.from.xlogin. To read the review, look at the files in ~/pb152/reviews/01.from.xlogin – you will find a copy of your submitted sources along with comments provided by the reviewer. After you read your review, you should write a few sentences for the reviewer into note.txt in the review directory (please wrap lines to 80 columns) and then run:

```
$ pb152 review --accept 100
```

Instead of 100, you can use a smaller number, indicating what percentage of the points the reviewer deserves for their job. Please make sure that you grade the review honestly – the reviews will be screened for abuse and depending on the type of misconduct, one or both parties will be punished.
NB. Please do not request reviews of assignment 03, since the submitted tarball cannot be annotated with comments.

A.4.2 Writing Reviews

To participate as a reviewer, start with the following command:

```
$ pb152 review --list
```

You will get a list of review requests for which you are an eligible reviewer. In particular, only assignments that you have already successfully solved will show up. If you like one of the entries, note its number (e.g. 7) and type:

```
$ pb152 review --checkout 7
$ cd ~/pb152/reviews/
$ ls
```

There will be a directory for each of the reviews you agreed to write. Each directory contains the source code submitted for review, along with further instructions (the file readme.txt).
In shell scripts, please use ## (double hash symbol) in the first column for your reviewer comments, like this:

```
## You are incorrectly using uniq without using sort first, this
## will leave duplicate lines if they are not next to each other.
cat file.txt | grep pat | uniq
```

In C files, use double `**` in comments, like this:

```
/** A short, one-line remark. **/
```

or for longer comments:

```
/** A longer comment, which should be wrapped to 80 columns or
 ** less, and where each line should start with the ** marker.
 ** It is okay to end the comment on the last line of text like
 ** this. **/
```

You can write up to 8 reviews, each for a maximum of 0.25 points (and a total of 2 points). The limit is applied at checkout time: once you agree to do a particular review, you cannot change your mind and 'uncheckout' it to reclaim one of the 8 slots.

Part 1: First Contact

Now that we know the basic rules and goals, we can start finding our way around the operating system. We will learn the basics of shell (the command interpreter) and some basic utilities today. You can start by typing this:

```
$ cat tricks.txt
```

Part 1.1: A Few Useful Tricks

1. Before we continue, I will try to teach you a few tricks for working with the command line a little more efficiently. The first one is using the command history.
2. For instance, if you just typed a long command and now want to type a rather similar long command, it might be useful to hit 'arrow up' on your keyboard. Then you can tweak the command before running it again.
3. If a while passed since you typed a command but want to run it again (or edit it slightly), you can use `^R` (that is `ctrl+r`) to find it in your history. Hit `^R` and type a few letters or a word from the command. If you typed the right fragment but you want an older command than you got, just hit `^R` again, until you find the right one.
4. Using `^R` is often faster than hitting 'arrow up' 20+ times.
5. Sometimes it's faster to type the command again instead of looking it up in your history. Might be also better for your memory.
6. The other trick is using the TAB key, especially when working with files. We will learn more about files shortly. If you want to read this file again, you could do it this way:

   ```
   $ cat tr<TAB><ENTER>
   ```

   Saves a few keystrokes.
7. The last two items have to do with the terminal emulator itself. First is about copying and pasting: using `^C` and `^V` (`ctrl+c` and `ctrl+v`) won't work. You could use shift+insert to paste. Or you can use the mouse to both copy and paste. You copy by selecting the text with a mouse and nothing else.
8. You can also paste by clicking the middle button (the scroll wheel). The same approach (select+middle click) also works with the browser and other programs.
9. You can look back to what you typed and the output of programs that scrolled off the screen by scrolling in the terminal (e.g. by using the scroll wheel). However, it is sometimes more convenient to use a pager, e.g. `less`. You can read this file in a pager like this:

   ```
   $ cat tricks.txt | less
   ```

   In this case, this would also work:

   ```
   $ less tricks.txt
   ```

10. The last thing that is often useful is to have more than one terminal open. For instance, you can follow these instructions in one and try out the commands or solve the exercises in another. I find it most convenient to put them side by side, so I can see both at once.

That are all the tricks for now. You may want to practice them for a bit – the next section will be a good chance to do that, because we will work with files. Continue to:

```
$ cat files.txt
```

Part 1.2: Files

1. As you know from the lecture, files are pretty fundamental in most operating systems, especially in those from the POSIX family.
2. There is a rich set of commands for working with files. You already met `cat` which prints files. It is called `cat` because it can con-cat-enate multiple files together. Like this:

   ```
   $ cat intro.txt tricks.txt
   ```

   I will remind you that you can learn more about cat by typing:

   ```
   $ man cat
   ```

3. Another very useful command is `ls`. It is short for 'list files'. Many commands are shorthands and they are easier to remember if you know what the shorthands mean. You can try `ls` like this:

   ```
   $ ls ..
   ```

   Of course, you can also consult the man page. I'm sure you already know the command to do that?
4. Then there is `cp` for copy and `rm` for remove. You can use them like this:

   ```
   $ cp intro.txt ~/pb152-intro.txt
   $ cat ~/pb152-intro.txt
   $ rm ~/pb152-intro.txt
   ```

5. It would be a good time to create a directory in which you could play without messing up your home directory. Let's use `mkdir` (from 'make directory') for that, and let's also enter it right away using `cd` (for 'change directory'):

   ```
       $ mkdir scratch
       $ cd scratch
   ```

6. You can create an empty file using `touch`. This command will also update the modification timestamp on a file if it already exists. This is not very important to us for now. Try:

   ```
   $ touch file.txt
   ```

7. If you forget where you are, you can always ask the system to `pwd` (print working directory):

   ```
   $ pwd
   /home/yourlogin/pb152/01/scratch
   ```

   (when a line in an example like above does not start with `$`, it is not a command – it is what you should see in the terminal after typing the command that's marked with `$`)
8. You can refer to the working directory using `.` or often nothing at all. Try the following commands:

```
$ ls .
$ ls
$ cp intro.txt /tmp/yourlogin.txt
$ ls
$ cat ./intro.txt
$ cat intro.txt
$ rm /tmp/yourlogin.txt
```

What follows is a few philosophical remarks about using the command line. We often shorten 'command line interface' to CLI or cli. Hence, continue to:

```
$ cat cli.txt
```

Part 1.3: Command-Line Interfaces

1. If this is new to you, your head might be spinning from so many cryptic commands. For many tasks, you can use graphical programs. However, even if it might not look like it, the command line is one of the most efficient interfaces for working with files (and doing many other things besides).
2. Unfortunately, you would have to learn many more complicated commands and master combining them using the shell. If you want to become a wizard, I would recommend trying to do that. Nonetheless, there are many good programmers who are not command line users. It is up to you.
3. The one exception is when I ask you to show how to do something to your classmates. Then, you will have to use the terminal and command line exclusively. But I won't get mad at you if you have to think about the commands to use and it takes you a few tries to type them correctly.
4. One last thing I wanted to mention is that you can probably appreciate that it is much easier to give instructions as a list of commands to type, than it is to tell another person over the internet how to do something in a point-and-click program. You simply copy down the commands (or use `script`) instead of taking many screenshots and drawing red circles and arrows telling the user what to click in what order.

Sorry for the detour. You can now continue to:

```
$ cat files.sh
```

Part 1.4: `files.sh` (source code)

This file is a shell script. Lines starting with `#` are comments, while everything else in the file are commands. Commands in a shell script work the same like they do in a command line. Make sure you are in your `~/pb152/01` (you can use `pwd` to check).
You can run this script (the file that you are currently reading) by typing this command (you should probably do that in your other terminal):

```
$ sh files.sh
```

Doing that is the same as typing all the commands listed in this file one by one. Shell scripts are very useful when you find yourself typing a long sequence of commands over and over again. This happens when compiling C programs, for instance. We will look at the C compiler next week, and then at automating it in two weeks.
For now, we will demonstrate some of the things that shell scripts can do. First, we will run some of the commands we already know:

```
mkdir scratch
cd scratch
cp ../intro.txt .
cp ../../info/grading.txt .
```

The above two commands copied two of the text files you have seen earlier into our working directory. Let's also copy this shell script:

```
cp ../files.sh .
```

You may remember `echo` from earlier, too. Let's tell the user that we did something:

```
echo "we are done copying the files"
echo
```

Let's look at something new. You remember `ls` which lists files. We can list individual files too, instead of entire directories:

```
echo listing just files.sh
ls files.sh
echo
```

Now we will use something called 'globbing' – we will refer to multiple files with a shorthand, using the character `*`:

```
echo listing all text files
ls *.txt
echo
```

One of the useful commands that we didn't mention yet is `head`, which prints a few lines from a given file. We can tell it how many we want using `-n`. You can learn more, as usual, in `man head`.

```
echo first line of intro.txt
head -n 1 intro.txt
echo first line of grading.txt
head -n 1 grading.txt
echo
```

Sometimes, we want to run some commands for multiple files. We could type them out, like we did above. But often, we can use a loop instead. In Python, you would write a `for` loop. We can do something similar in a shell script:

```
for f in *.txt; do
    echo "first line of $f (in a loop)"
    head -n 1 $f
done
echo
```

Notice that we used double quotes in the `echo` this time. That is because parentheses are special to the shell, but we just want to print them. When you want to print something and you are not sure whether it contains special characters, use double quotes. They won't hurt if they are not required.

```
echo "no double quotes were needed here"
echo
```

Since this script is getting long, we will stop and continue in another one. Let's first clean up the files though:

```
rm intro.txt grading.txt files.sh
cd ..
rmdir scratch
```

Now you can finally do an exercise for yourself. Let me give you the instructions right away:

```
echo that is all, instructions for an exercise follow
echo
cat 1.txt
```

Part 1.5: Exercise 1

1. You can volunteer to do this exercise 'at the blackboard'. You may want to review the instructions here:

```
$ cat ../info/activity.txt
```

2. Perform the following tasks (using commands you already know):
   - change into the directory `pb152` in your home: `~/pb152`
   - list the contents of the directory
   - create a directory called `scratch`
   - copy all the text files (but only the text files) from `~/pb152/01`
   - print first two lines of each of them
   - list the directory content again
   - remove all the exercise description files, that is `{number}.txt`
   - list the directory once more
   - remove all the remaining files
   - remove the now-empty directory
3. It may be helpful to know that you can type a `for` loop on a single line and use it as a command. Like this:

   ```
   $ for f in *.sh; do ls $f; done
   ```

4. Another useful tidbit: `?` (question mark) matches a single character when globbing.

That was all for the exercise. The next thing to learn will be text editors, because we will want to write some simple scripts. Go on to:

```
$ cat editors.txt
```

Part 1.6: Text Editors

1. Editing text files is a very essential activity, especially when interacting with UNIX. We have two basic options.
2. The traditional UNIX editor is `vi` (its modernised, extended variant is known as `vim`). This editor is hard to master, but very efficient. I encourage you to try learning it, especially if you see yourself as a UNIX person in the future. However, it takes a lot of practice and this seminar is not about text editors.
3. If you want to use `vim`, you need to know these: use `i` to enter text, ESC to stop entering text and when you are not entering text, `:wq` to save your file and exit the editor and `:q!` to exit the editor and not save the file. You can open a text file like this:

   ```
   $ vim file.txt
   ```

   Unlike traditional `vi`, you can use keyboard arrows to move around the file while in insert mode (i.e. when entering text). This makes it almost possible to use `vim` even if you are a novice. It is up to you.
4. Your other terminal-based option is `micro`. This is a very simple text editor, and it is not very efficient. It will be sufficient for this course though. Micro is invoked the same way as `vim`:

   ```
   $ micro file.txt
   ```

5. If you are not at the 'blackboard' at the moment, feel free to use graphical text editors like `gedit`. Those are quite simple and intuitive. Sometimes, combining graphical and command line programs is the most efficient way to do things.

Try editing some files in an editor. When satisfied, you can move on to the second exercise for the day. Like before, you can do it at the blackboard if you wish:

```
$ cat 2.txt
```

Part 1.7: Exercise 2

1. You may want to start by reviewing `file.sh`. In this exercise, we will write a simple script.
2. The script should do the following:
   - create a new directory called `scratch`
   - enter this directory
   - copy all exercise files from `~/pb152/01` (1.txt, 2.txt, …)
   - count how many lines there are in each (try `man wc`)
   - change directory back to where you started (hint: try `cd ..`)
   - erase everything in `scratch` along with the directory (`man rm`)
3. Be careful with `rm`! The option you are looking for is called 'recursive'. You should also use option `-i` if you are not 100% sure! This will ask about each file to erase before doing so.
4. Run the script. If you do not remember how, this was mentioned near the top of `files.sh`.

We will need to review pipes before we go any further. Hence the way forward is:

```
$ cat pipes.txt
```

Part 1.8: Pipes

1. It will be a few more weeks before we get to describe pipes in the lectures. For now, we can consider them to be magical.
2. The way we will use pipes is to connect programs together in a simple way.
3. If you remember, it was mentioned earlier that you can run a file through a pager like this:

   ```
   $ cat file.txt | less
   ```

   This is a pipe. The entire command is sometimes called a 'pipeline'.
4. Pipes take output of one command and present them as input to another. Most UNIX utilities are designed to work well with pipes.
5. If you want to count all the lines in all the text files in your current directory, you can say:

   ```
   $ cat *.txt | wc -l
   ```

6. One useful command to use in pipelines is `cut` (try `man cut`). You can also try the following command:

   ```
   $ echo hello:world | cut -d: -f2
   ```

   The switch `-d` gives the delimiter, `-f` asks for a particular delimited field. You can use ranges too, like `-f 2-4`.
7. Pipelines are often used for working with text. A very powerful tool is `grep`. It allows you to use regular expressions to select a subset of lines that match a pattern. We will only use it in a simpler version, without regular expressions for now. This variant is called `fgrep` (where `f` stands for 'fixed'). To print all lines from `file.txt` that contain the word `frob`, you could run:

   ```
   $ fgrep frob file.txt
   ```

   Or with pipes:

   ```
   $ cat file.txt | fgrep frob
   ```

8. I will show you one last command for today, and that is `sort`. This is also quite useful in pipelines. It rearranges input lines into alphabetical order.

To practice working with pipes, check out `3.txt`.

Part 1.9: Exercise 3

1. This is a simple exercise for pipes. It is also interactive: you do not need to write a script.
2. Perform the following tasks:
   - make a copy of `oslist.csv`
   - look at the content using `cat`
   - print names (and only names) of all operating systems made by Apple
   - print vendors of all microkernel operating systems

- print names and vendors of operating systems sorted by year
- erase your copy of `oslist.csv`

Now let's move on to the final exercise for today. What to do is described in `4.txt`.

Part 1.10: Exercise 4

The final exercise for today is to chain a few commands into a single pipeline that takes all the text files in today's tutorial and lists all the command names (programs) that appear in the examples, one per line. Each command should appear exactly once. For instance, `fgrep`, `echo` or `less` should be listed (among quite a few others).
That's all. The homework is described in `hw.txt`. Good luck and have fun.

Part 1.11: Homework

1. To submit the homework, type `pb152 submit` in this directory. The first deadline is Wednesday next week (26.2.) at 10pm.
2. The exercise is to write a shell script (save it as `hw.sh`). The script should do the following:
   - create a directory called `scratch`
   - change into the directory
   - make a copy of all shell scripts from `~/pb152/01` (files ending `.sh`)
   - make a copy of `oslist.csv` from `~/pb152/01`
   - count operating systems by Apple, Google and Microsoft
   - print the vendor of the most recent operating system
   - count the number of different vendors (`man uniq`)
   - change back to the original directory
   - erase everything in `scratch` and the directory itself
3. The script will not be running in `~/pb152/01`. Use the entire path to refer to that directory.
4. The expected output would be:

   ```
   OS count by Apple:
   3
   OS count by Google:
   2
   OS count by Microsoft:
   3
   Latest operating system by:
   Google
   Number of different vendors:
   7
   ```

   Please follow the above output format exactly.
5. That's it for today. See you next week!

Part 2: Descent into C

1. Hopefully, you remember how to use the shell and how to browse these documents from last week. If you need to review, just run:

   ```
   $ cat ~/pb152/info/intro.txt
   ```

   The lecture for today, however, starts here:

   ```
   $ cd ~/pb152/02
   $ cat intro.txt
   ```

2. Today, we will get acquainted with the C compiler, the linker and we will write some very simple C programs.
3. The traditional UNIX name for the C compiler is `cc`. On most Linux systems (this one included), the default C compiler is known as `gcc` (short for GNU `cc` – this is mainly because you could install `gcc` on computers that already had their own system `cc`).
4. You can usually run `gcc` using the command `cc` whenever it is the default compiler. I would actually recommend that you do it this way, unless you require the GNU compiler specifically. Not all modern UNIX systems use `gcc` as the default compiler, sometimes for a good reason.
5. You can read the documentation for how to run `gcc` using:

   ```
   $ man gcc
   ```

   Unfortunately, on this computer `man cc` does not work.
6. The linker is available as `ld` (probably from Link eDitor, some people think it is from LoaDer, but that's more tenuous). See also `man ld`.
7. The linker takes multiple binary artifacts (object files, libraries) and produces a single file which contains an executable program. How this works will be explained in more detail in the lectures.
8. We will not run the linker directly here. This is rarely needed or useful even in the real world.

You can continue by looking at `hello.c`:

```
$ cat hello.c
```

You can also use `vim` or `micro` to open it (whichever you normally use), to make it easier to read:

```
$ micro hello.c
```

Part 2.1: `hello.c` (source code)

As you may recall, in shell scripts, lines which start with `#` are comments, and not part of the script to be executed. In C programs, comments are instead written like this. Text of this form will be ignored by the compiler.
You have probably seen a C program before. That is okay, but let me remind you about the basic structure, and how to compile and run the program anyway. The first thing is #include statements:

```
#include <stdio.h>
```

The `stdio.h` header contains useful functions that let us print things to the screen. They are part of `libc` and use operating system facilities to do their work, like system calls. The functions are really complicated on the inside, but we don't have to worry about that yet.
There is only one other thing in this very simple C program, and that is the `main` function. Every C program has one, but not every C source file does. We will learn about that later. When you run a C program, the operating system (via its libraries) ensures that the `main` function starts executing.

```
int main()
```

This is a simple function definition: it has the return type (`int`), the name of the function (`main`), the argument list (empty parentheses) and the body (starting on the next line).

```
{
```

This is where execution of the program starts. Statements follow.

```
    puts( "Hello world!" );
```

The `puts` function simply prints the string provided as its first argument to the terminal. It also prints a newline afterwards.

```
    return 0; /* <main> returns an integer and 0 means 'no error' */
```

```
}
```

You can compile and run this program by issuing the following commands:

```
$ cc hello.c
$ ./a.out
```

The name `a.out` is short for 'assembler output'. We will learn more about that shortly. To continue, run:

```
$ cat compiler.txt
```

Part 2.2: Compiler

1. Going from a source (text) file to something a computer can execute is a fairly complicated process. What we normally call a 'compiler' is, in fact a driver: a program that supervises the process.
2. The commands `cc` and `gcc` actually invoke this driver.
3. The usual steps the driver invokes are as follows:
   a. preprocessor: `cc -E`
   b. compiler proper: `cc -S` α. lexical and syntactical analysis β. semantic analysis and type checking γ. optimisation δ. assembly (machine code) generation
   c. assembler: `cc -c`
   d. linker: `cc`
4. The commands in the parentheses above are how to invoke that particular stage. Let's try:

   ```
   $ cc -E -o hello.i hello.c
   $ micro hello.i
   ```

5. The `.i` file contains the program after the first step of the compilation process.
6. All preprocessor directives have been processed. Those are mainly `#include` but also `#if`/`#else`/`#endif` and `#define` macros.
7. You can find our `main` function near the end of the `.i` file.
8. Now try:

   ```
   $ cc -S hello.i
   $ micro hello.s
   ```

9. And finally:

   ```
   $ cc -c hello.s
   $ objdump -r --disassemble hello.o
   ```

Let's move on to a slightly more interesting program.

```
$ micro read.c
```

(As always, when I say `micro`, feel free to use `vim` instead.)

Part 2.3: `read.c` (source code)

This is a simple program that uses the POSIX API to work with a regular file. As usual, we start with `#include` statements. In this case, we keep the familiar `stdio.h` but add `unistd.h`. You may recall from the lecture that the latter contains prototypes for basic POSIX functions. Additionally, we will need one more `#include` to get access to the `open` function, since it is not part of `unistd.h`.

```
#include <stdio.h>   /* printf */
#include <unistd.h>  /* read, write */
#include <fcntl.h>   /* open */
```

Since this is a simple program, it only has a single function: `main`. As you may remember, this is the entry point – execution starts here.

```
int main()
{
```

We will start by opening a file. The function `open` takes 2 arguments: the path to the file that we wish to open, given as a string, and the mode, which tells the OS what we intend to do with the file later. We are only interested in reading, so we will say `O_RDONLY` which is short for 'open read only'.

```
    int fd = open( "read.c", O_RDONLY );
```

`fd` is short for file descriptor, and this name is used quite commonly. The variable is of type `int`: it is specified by POSIX this way. We should check whether the `open` call worked as we expected it to, so let's do that. If we look at `man 2 open` and scroll a bit, we will find a section that's named return value:

> If successful, `open()` returns a non-negative integer, termed a file descriptor. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

Analogous sections are in almost all manual pages which describe the POSIX C API.

```
    if ( fd < 0 ) /* this would indicate an error */
    {
```

We don't know what would be the right thing to do here yet... let's just return 1, indicating the program failed.

```
        return 1;
    }
```

If the program made it to this point, we know `fd` is a valid file descriptor. Let's try reading from it. We will need some space for the data. Let's make a character array for that. If you are not familiar with the syntax, don't worry about it too much yet. Don't worry about the `ssize_t` either, it is just another signed integer type.

```
    char buffer[451] = { 0 };
    ssize_t bytes = read( fd, buffer, 450 );
```

The `read` function takes three arguments. The first is the file descriptor, the second is the space where to load the data and the final is how many bytes to read from the file.

```
    close( fd ); /* let's not forget to close the file descriptor */
```

With that out of the way, a number of things can happen. Let's turn to `man 2 read`. It tells us that if we get a positive number back, that is the number of bytes that were actually read. This will never be more than the third argument, but could be less for various reasons. For instance, the file may be shorter than 450 bytes.

```
    if ( bytes < 0 )
        return 2; /* we don't care much about errors just yet */
```

We now use the familiar `puts` function to print the data we obtained from the file. Here it is:

```
    puts( buffer );
```

That's all. Can you say what the program prints before you run it?

```
    return 0;
}
```

We learned how to compile programs just a short while back, so perhaps you can do that on your own. But in case you forgot, let me remind you:

```
$ cc read.c
$ ./a.out
```

Sorry about the long file! We had a lot to cover. Please continue to:

```
$ cat 1.txt
```

Part 2.4: Exercise 1

1. Let's dust off some of the knowledge we gained last week. Let's write a shell script. Since we learned some new commands today, let's also use those: we will compile a program.
2. Write a shell script, `compile.sh` that does the following:
   - create an empty directory, called `scratch`
   - enter this newly created directory
   - make a copy of `read.c`
   - preprocess `read.c` into `read.i`
   - compile `read.i` into `read.s`
   - assemble `read.s` into `read.o`
   - link `read.o` into `a.out`
   - run the resulting program, `a.out`
3. Normally, most of the steps would be done by `cc` automatically. But we are trying to learn about how `cc` works, which is why we do all those extra steps.
4. To compensate for making you do these boring steps by hand, I'll teach you a cool trick. Write `set -x` near the start of your shell script (before any other commands it runs). This will make `sh` print every command as it is about to be executed by the shell script.

As usual with those exercises, feel free to volunteer. You may want to review the instructions here:

```
$ cat ../info/activity.txt
```

Let's get back to some C:

```
$ cat args.c
```

Part 2.5: `args.c` (source code)

We will need to make a small detour from the filesystem. You remember how you can write `cat file.txt` to have `cat` work with `file.txt`? We call the things that come after a command 'arguments' (or sometimes, to be explicit about it, we call them 'command-line arguments'). We will now learn how to work with those in our C programs.
For now, we are back to only needing our favourite #include:

```
#include <stdio.h>
```

And again, this is a simple program, with only a `main` function. But now we will give `main` some (function) arguments:

```
int main( int argc, const char **argv )
```

By convention, those arguments are called `argc` and `argv`. The first is an integer and tells us how many command-line arguments there were. But be careful: this includes the program name itself. So for instance:

```
$ ./a.out file.txt
```

would give us `argc` = 2. The other (`argv`) is more complicated. Let's not worry about that for now. It is always OK to write it as it is written here.

```
{
```

We start by checking how many arguments are there. If there aren't any interesting ones, we give up right away.

```
    if ( argc < 2 )
```

It would be nice to tell the user what went wrong – we will do that another time.

```
        return 1;
```

If we made it here, we know that there's at least one argument, in addition to the program name. We don't know much about C yet, but I'll tell you that the way to get the first argument is `argv[1]`. It is a string, just like `"Hello world!"` was.

```
    puts( argv[1] );
    return 0;
```

This program was hopefully simpler than the last one. Unfortunately, it uses some mysterious syntax and it'll take us a few weeks to decipher that properly.

```
}
```

You can run this program by issuing these commands:

```
$ cc args.c
$ ./a.out hello there
hello
```

As you can see, the program prints its first command-line argument. Let's try using that right away:

```
$ cat 2.txt
```

Part 2.6: Exercise 2

1. Let's try to write a C program now. You can refer to the two programs I have shown you today when writing the new one.
2. Let's call the program `kitten.c`, because it'll be like a very tiny version of `cat`. It should do the following:
   - check that it has at least 1 argument
   - if so, take that to be a file name and open it
   - read the first kilobyte of the file
   - and print it
   - don't forget to close the file descriptor
3. Compile and run your program.
4. As a bonus, you can try the following:

   ```
   $ ./a.out ../01/oslist.csv | cut -d, -f3
   ```

   We can pipe the output of our little program to other commands, just as we could with `cat`.

Let's turn to system libraries for a bit:

```
$ cat syslibs.txt
```

Part 2.7: System Libraries

1. We will first look at some of the system libraries we have mentioned in the lecture. Let's look at `/usr/lib` first:

   ```
   $ ls /usr/lib/*.a
   $ ls /usr/lib/*.so
   ```

2. I want to show you an interesting new command. It is called `file` and its job is to identify the type of a file based on the content (not the file name). Let's try:

   ```
   $ file /usr/lib/libc_nonshared.a
   /usr/lib/libc_nonshared.a: current ar archive
   $ file /usr/lib/libc-2.17.so
   /usr/lib/libc-2.17.so: ELF 32-bit LSB shared object, [...]
   ```

3. Using `file`, we learned a few things. Even before that, however, we notice that the file names are all wrong. Unfortunately, there is no `libc.a` on `aisa`, because many people consider static libraries obsolete. The `libc_nonshared.a` is a helper library that only contains a small fraction of `libc`.
4. The shared version of the library has a version number in the file name. This is unusual, but `libc` on this computer is somewhat

special. If you are interested, you can peek at the file named libc.so, which is not an actual shared library, as confirmed by file:

```
$ file /usr/lib/libc.so
$ ls -l /usr/lib/libc.so.6
```

5. The text file libc.so instructs the linker to look for libc.so.6, which is a symbolic link (we don't know those yet, but think of it as an alias for a different file). It points to our libc-2.17.so.
6. The other kind of versioning – the 6 in libc.so.6 is common and this is how most shared libraries are. It means that this is the 6th major revision of libc.
7. The second thing we learned is that the shared version of libc is 32-bit. But surely, aisa is a 64b machine? Yes, and for historical reasons, the actual system libraries on aisa are located in /usr/lib64. I am sorry, life is complicated.

This was a little disappointing. Let's move on to making our own (static) libraries instead:

```
$ cat archives.txt
```

Part 2.8: Archives

1. You might remember that static libraries are basically zipped-up object files. Hence, we will need some object files to make a static library.
2. I prepared two very simple math functions, each in its own C file. You can have a peek if you like:

   ```
   $ cat factorial.c
   $ cat fib.c
   ```

3. Let's compile those into object files:

   ```
   $ cc -c factorial.c
   $ cc -c fib.c
   $ ls *.o
   ```

4. And make a static library out of them:

   ```
   $ ar r libmath.a factorial.o fib.o
   ```

5. And we can use the simple math.c I prepared for you for our main function:

   ```
   $ micro math.c
   $ cc -c math.c
   $ cc math.o libmath.a
   $ ./a.out
   ```

6. It might be interesting to see what happens, if we try to link our math.o without the library:

   ```
   $ cc math.o
   math.c:(.text+0xe): undefined reference to `fib'
   collect2: error: ld returned 1 exit status
   ```

   The linker tells us that it did not find a definition for fib and cannot produce an executable for this reason. We also see that the compilation driver in cc invoked ld internally, and told us that ld failed with an exit code 1.

Now we should be ready to work out exercise 3:

```
$ cat 3.txt
```

Part 2.9: Exercise 3

1. In this exercise, we will practice working with object files and archives (static libraries) a little.
2. We will start by creating an empty directory. Let's call it ex3.
3. In that directory, let's create a 3-part program, made of main.c, hello.c and bye.c. We will also need greetings.h.
4. In each of the .c files, there will be a single function. In hello.c, it'll be void hello(), in bye.c it will be void bye() and in main.c, it'll be int main().
5. Our greetings.h will look like this:

   ```
   void hello( void );
   void bye( void );
   ```

   There will only be those two lines. We can use greetings.h by using an #include statement, like this:

   ```
   #include "greetings.h"
   ```

6. Each of hello() and bye() will simply write a message using puts. You can choose the message. And main() will simply call hello() and bye(), in this order.
7. Now let's build our program in stages. We will want main.o, bye.o and libhello.a. Then link them all together to form a.out, as usual.
8. Check that a.out works as expected.

We can now go back to reading and writing files in C. Let's have a look:

```
$ micro stdio.c
```

Part 2.10: stdio.c (source code)

In this program, we will work with 'standard input' and 'standard output'. We will also look at the relationship between puts and write. As usual, we need some #include statements:

```
#include <stdio.h>
#include <unistd.h>
```

Notice that we did not #include <fcntl.h>. We will not need open.

```
int main() /* we won't need command line arguments either */
{
```

First we print a message using puts, as we did before.

```
    puts( "hello from puts" );
```

We now need to take a small precaution. Near the start of today's lecture, I told you that puts does complicated things on the inside. One of those things is called buffering: it will collect characters that you asked it to print in a piece of memory (a buffer) instead of printing them right away. It will print the buffer only when it fills up, or when we ask it to do that explicitly. We can use fflush for that. The stream into which puts sends the string is called stdout. A stream is basically a file descriptor along with the buffer.

```
    fflush( stdout );
```

Internally, what fflush does (on a UNIX system) is call write with the content of the buffer that was stored away by puts. The buffers are always flushed by the standard library when the program exits normally.
We can also try to call write directly. The file descriptor that hides behind the stdout stream is also called stdout and is, by convention, assigned the number 1. We could also call it STDOUT_FILENO, but we don't need to bother. The 18 in the third argument is, however, more of a problem. We will let it slide for now. The \n in the string means a newline – unlike puts, write won't add that for us.

```
    write( 1, "hello from write\n", 18 );
```

We will try to read from stdin now, and we need space for that, so let's make some.

```
        char buffer[20] = { 0 };
```

stdin is file descriptor 0. Let's try reading from it.

```
        ssize_t bytes = read( 0, buffer, 19 );

        if ( bytes >= 0 )
        {
```

And just print back whatever we got back to the user.

```
            write( 1, "what we got from stdin:\n", 24 );
            write( 1, buffer, bytes );
        }
        else
        {
```

Something failed. I promised earlier that I will tell you what to do with errors. This is not very pretty, but since we are using write already, let's stick with that. The fd number 2 is called stderr and that's where we send error messages.

```
            write( 2, "reading stdin failed :(\n", 24 );
            return 1;
        }
    }
```

Now compile and run this program as follows:

```
$ cc stdio.c
$ echo hello from echo | ./a.out
```

What do you think it will print? Next on:

```
$ cat redirects.txt
```

Part 2.11: Redirects

1. We have already seen pipes last week. But pipes are not the only way to redirect inputs and outputs of programs. The other is redirections, which work with files.
2. A simple thing to try is redirecting the output of cat, like this:

   ```
   $ cat hello.c > redirected.c
   $ cat redirected.c
   ```

3. We see that the output that cat would normally send to the screen, or to another program using a pipe, was written into the file redirected.c. This was easy.
4. In the other direction, we can read things from files instead of from the keyboard, or from other programs via pipes. Consider:

   ```
   $ cut -d, -f3 < ../01/oslist.csv
   ```

5. You see that the input of cut came from the file we specified using <.
6. Let's try something slightly different. There are special files on UNIX, and one of those special files is /dev/null. This is a file that cannot be read and throws away anything you write to it:

   ```
   $ echo hello > /dev/null
   $ cat /dev/null
   ```

7. We still have the a.out which came from stdio.c. Let's try redirections with that one:

   ```
   $ ./a.out < /dev/null
   $ ./a.out < /dev/null > /dev/null
   ```

Can you explain the output?
That's all I had, we can move on to the final exercise:

```
$ cat 4.txt
```

Part 2.12: Exercise 4

Let's revisit input and output for a bit. You may not know how to write loops in C yet, but your seminar tutor will surely help you out. That said, write a program that does the following:

1. read 50 bytes from stdin
2. write those 50 bytes to stdout
3. wait for 100 milliseconds (see man 3 usleep)
4. repeat until there are no more bytes to read

You can try it on one of the longer files in here, e.g. read.c like this:

```
$ ./a.out < read.c
```

Part 2.13: Homework

1. There will be two parts to the homework.
   a. Write a C program, write.c.
   b. Write a shell script, test.sh.
2. The C program will work as follows:
   a. It will read (at most) one kilobyte of data from its standard input.
   b. It will (create) and write this data into a file that is given to it as the first argument.
   c. It will complain using stderr if something fails, and return a non-zero exit code.
3. A hint: to create and write into a file, you will want to use these flags to open: O_RDWR | O_CREAT. This is the second parameter. You also must pass a third argument to open for this to work. This will be 0666 (a number). This will be explained later.
4. Now for the shell script:
   - compile write.c into a.out in the 4 stages
   - the files write.i, write.s and write.o should exist
   - use a.out to make a copy of write.c under copy.c
   - show that a.out prints an error when writing fails (try to write into a directory instead of a file, or somewhere under /usr where you don't have the permissions to create files)

That is all. Happy hacking.

Part 3: More Shell Scripts, make

1. In this lecture, we will learn more about writing shell scripts. We will also learn about makefiles, which primarily exist to automate compilation.
2. But the first part will be about shell scripts, of the sort that we started with in the first lecture 2 weeks ago.
3. We will learn about variables, conditionals, loops and a collection of small programs that make writing scripts much easier.
4. There is no single 'shell language': instead, this is a family. The most important of various shells are those descended from the Bourne shell.
5. POSIX mandates that /bin/sh is always a Bourne-compatible shell.
6. That said, we will only deal with Bourne shells: GNU bash is one widely used implementation (the default /bin/sh on many Linuxes), some other are ksh and dash.

That said, let's get on with it:

```
$ micro variables.sh
```

Part 3.1: variables.sh (source code)

This is a shell script that will demonstrate basic variable substitution. We will be ahead of the lecture a fair amount, but the things we will do are not hard. I hope you remember pipes from the first week (we also used those in the second week, they are written as `program1 | program2`).
Let's also ask the shell to tell us what is going on. We saw this last week:

```
set -x
```

There are two types of variables that we are interested in. First are shell variables, the second are environment variables. There is a fair amount of overlap, which can be confusing. We need to be careful about the distinction.

```
variable=value
```

The above syntax sets the value of VARIABLE to be value. In a traditional shell, there are no types: everything is a string. There must be no spaces around the `=` 'assignment operator'.
To use a variable, we need to use special syntax. Compare

```
echo variable
echo $variable
```

You can also set a variable to the output of a program. Incidentally, the `$(command)` construct can be used in other places too, we may encounter that later.

```
variable=$(echo hello from echo)
: $variable
```

The command `:` does nothing. It is useful when we are using `set -x` because it shows up in the trace, along with all the arguments (which are `$`-expanded).
You should run this script now. The invocation is:

```
$ sh variables.sh

: "That's it for now. Continue to: env.sh"
```

Part 3.2: env.sh (source code)

As I just mentioned, there are 2 types of variables floating around in UNIX. The other are the environment, a set of variables passed around to each program. In a short while, we will write a C program to look at them, so you see that they are not the same as shell variables.
In any case, the program `env` prints all environment variables. You can run it like this:

```
echo '# looking for PATH in the environment'
env | fgrep PATH
```

Shells typically make it possible to add their own variables into the environment, and they make all environment variables available as their own (via `$` expansion).

```
echo '# doing echo $PATH'
echo $PATH
```

Adding variables into the environment is done with `export`:

```
echo
echo '# setting VAR, echo-ing it and looking for it in env'
VAR=17
echo $VAR
env | fgrep VAR

echo
echo '# exporting VAR, echo-ing it and looking for it in env'
export VAR
echo $VAR
env | fgrep VAR

echo
echo '# changing the value of VAR'
VAR=13
echo $VAR
env | fgrep VAR

echo
echo 'That said, let us go on to getenv.c'
```

Part 3.3: getenv.c (source code)

This is a C program, I am sure you remember those. They start with some `#include` statements – you are also familiar with `stdio.h`, but we now encounter a new one: `stdlib.h`. This is because we want to use the standard C function `getenv` which is declared in this header.

```
#include <stdio.h>  /* printf */
#include <stdlib.h> /* getenv */

int main( int argc, const char **argv )
{
```

You probably remember how command-line arguments are passed into C programs. The `argc` and `argv` arguments to `main` contain the number of arguments and the strings representing them. That said, `argv[0]` is the name of the program itself. So we only care for argv[1], argv[2] and so on. What follows is a `for` loop: you don't need to be able to write one, but hopefully you can read the one here without much trouble. It just runs its body (the `printf`) for every value of `i` between 1 and `argc` (the latter not included).

```
    int i;

    for ( i = 1; i < argc; ++i )
    {
```

We have seen `printf` before, too, when we printed numbers in our simple math programs last week, but we always used `puts` to print strings until now. Well, using `printf` to do this is often more convenient. Like with the `for` loop, for now you only need to be able to read what this does and not write your own `printf` calls.

```
        printf( "%s → %s\n", argv[ i ], getenv( argv[ i ] ) );
    }
}
```

In case you forgot, compile this with:

```
$ cc getenv.c
$ ./a.out PATH
```

You should see the value of the `PATH` variable. You can also pass in more variable names:

```
$ ./a.out PATH TERM
```

Let's continue to:

```
$ cat 1.txt
```

Part 3.4: Exercise 1

1. Write a shell script that does the following:
   a. count and print how many environment variables are set
   b. set the variable `ENV_SIZE` to this value

   c. make ENV_SIZE part of the environment
   d. compile getenv.c from earlier (in 1 step)
   e. run it with ENV_SIZE as argument
2. That's it, now we can move on to test.sh.

Part 3.5: test.sh (source code)

This shell script demonstrates basic control flow: conditional statements, and how to write some basic conditions that you may want to test. The basic structure is simple: `if condition; then commands; else commands; fi`. We can either use semicolons or newlines. Different people prefer different styles.
The condition part of the if statement is a command. If the command is successful (i.e. if it returns exit code 0), the then branch of the conditional is taken. Otherwise, the else branch is. The else branch is optional.

```
echo 'we first try some if statements and the ‹test› program'
echo
```

The program that is most often used in if conditions is called test. You can use it to check basic properties of files (existence, whether they are regular files or directories, if one file has been modified more recently than another and so on). It can also test properties of numbers (basic relational operators) and strings (emptiness and equality). A few examples follow.

```
if test -e file.txt; then
    echo file.txt exists
fi

if test -f file.txt; then
    echo file.txt exists and is a regular file
fi

if test 3 -gt 4; then
    echo "3 is greater than 4"
fi

if test 3 -lt 4; then
    echo "3 is less than 4"
fi
```

There is another way to call test, and that is [. Yes, there is a command with the name 'left square bracket'. It is usually a (sym)link to test. In some shells, it is also built-in (both test and [).

```
if [ -e file.txt ]; then
    echo file.txt still exists
else
    echo file.txt does not exist
fi
```

The closing square bracket is an argument to [and it is required: [will complain if you don't supply it, like this:

```
[: the last argument must be ']'

echo
```

For more details, see man test. There are two other utilities that sometimes come in handy, called true and false. Let's try running them. I will also mention a new variable (this one is quite special) called ?. It holds the exit code of the last executed command. It is not possible to assign your own values into this variable.

```
true
echo exit code from true: $?

false
echo exit code from false: $?
```

```
if true; then
    echo "if true: then branch";
else
    echo "if true: else branch";
fi

if false; then
    echo "if false: then branch"
else
    echo "if false: else branch"
fi
```

I would suggest you run this script now, and also make sure that test -f does what we think it does:

```
$ sh test.sh
$ touch file.txt
$ sh test.sh
```

Let's move on:

```
$ micro chaining.sh
```

Part 3.6: chaining.sh (source code)

In this script, we will demonstrate how commands can be chained. This is sometimes useful in if conditions, but also in other scenarios. We already know pipes: those are a form of chaining too. But we are now interested in simpler things. You know that if you enter two commands on separate lines (in a script, or in an interactive session), those commands will run in a sequence. A semicolon does the same thing:

```
echo foo ; echo bar
```

That is not very interesting, but sometimes useful, if you want to run a sequence of commands at once and each of them takes a while and you don't want to watch for each to finish so you can type in the next one.
The next two ways are more interesting: the && and || operators:

```
true  && echo "this command will run"
false && echo "this command will not run"
```

Just like in C, && is short-circuiting: if the first command fails, the result must always be false and running the second command is not needed. Same (but opposite) with ||:

```
true  || echo "this command will not run"
false || echo "this command will run"
```

We can form a longer chain, too:

```
true  && echo "this will run"  || echo "but this won't"
false && echo "this won't run" || echo "but this will"
```

Of course, you can use those operators in conditionals:

```
if true && false; then echo "not happening"; fi

if test -f file.txt && test -f file.bak; then
    echo "both file.txt and file.bak exist and are regular files"
fi
```

This is all I wanted to say about chaining for now. We might look at the & operator later in the semester, if there's enough time. For now, let's move on to loops:

```
$ micro loops.sh
```

Part 3.7: loops.sh (source code)

We have already seen simple for loops, even though we didn't quite understand variables. For loops are quite simple:

```
for var in x y z; do
    echo $var
done
```

By now, you should understand what echo $var means, and the for loop simply means this: set var to x, run the body of the loop, then set it to y, run the loop again and then set it to z and repeat once more. The basic structure is like with if: you can use semicolons or newlines, as you prefer.
Of course, the above form of a loop can be useful, but the things to iterate will often be files. You may remember that we can use globs (patterns) to match multiple files. This works universally, not just in a for statement:

```
echo *.mk
echo

for file in *.txt; do
    if test $file -nt make.txt; then
        echo "$file is newer than make.txt"
    else
        echo "$file is older than make.txt"
    fi
done
```

The last form of a loop that I want to show you is a while loop. It is written like this:

```
echo
touch scratch.txt
while test $(wc -l < scratch.txt) -lt 5; do
    echo . >> scratch.txt
done
wc -l scratch.txt
```

We can also pipe things to and from loops:

```
for i in 1 2 3 4 5; do
    echo .
done | wc -l

echo
```

Especially useful (when piping things into loops) is the while read construct. The read command is provided by the shell to read inputs and put them into variables. command | read var is similar to var=$(command). You can try running:

```
$ echo bla | read foo
$ echo $foo
```

The loop version looks like this:

```
cat scratch.txt | while read line; do
    echo the line is this: $line
done
```

That is all about loops. You can now go on to 2.txt:

```
$ cat 2.txt
```

Part 3.8: Exercise 2

1. Write a shell script, that:
   a. creates and enters a scratch directory
   b. writes lines file1 through file5 into list.txt
   c. creates (touches) all files listed in list.txt
   d. renames each fileN to fileN.x (use globs)
   e. rename each fileN.x to fileN.y (use list.txt)
   f. clean up
2. A small hint: For b., use for i in 1 2 3 4 5. You can also use seq 5, like this: for i in $(seq 5). If you use seq, you can easily make more files without typing yourself to death.
3. Go on to make.txt:

   ```
   $ cat make.txt
   ```

Part 3.9: make

1. We have built C programs before by hand. This is OK if they are small.
2. But the compiler is not the fastest program in the world. We often want to compile and test our programs after modifying them.
3. If the program is big (it has many C files), it is wasteful to compile every one of those every time we want to test.
4. This is where make can help us.
5. We would like to only compile files that have been modified since their respective object files were built.
6. For instance, we have foo.c, bar.c and baz.c and we build the program, so we also have foo.o, bar.o and baz.o. We edit foo.c and want to build the program. We only need to compile foo.c and link the object files.
7. However, consider there is a header file, which is included by bar.c and baz.c. If we edit this header, both bar.o and baz.o need to be rebuilt. Do you understand why?
8. The program make can deal with exactly those problems. It reads a makefile – a script of sorts, in a special language – and only updates files that need to be updated based on this makefile.
9. The makefile can be called anything, and if we have build.mk, we can tell make to process it like this:

   ```
   $ make -f build.mk
   ```

   However, if the name is makefile or Makefile, it is enough to say:

   ```
   $ make
   ```

   You can try renaming any of the subsequent makefiles that we encounter to makefile (or Makefile) and run make without -f.
10. Now you can continue to build.mk.

    ```
    $ micro build.mk
    ```

Part 3.10: build.mk (source code)

This is a makefile (a script for the make program) that builds a simple C program. You should make a copy of this file, along with getenv.c and you can then build getenv using this command:
$ make -f build.mk
Like shell, make has variables. We will use them from the start, because they are very useful. One of the common variables we will encounter is CFLAGS, which is what make passes to the C compiler. We can use the += operator to add things to existing variables.
Here, we will tell the compiler to emit additional warnings and to switch to C99 mode.

```
CFLAGS += -Wall -std=c99
```

Make uses rules, which describe how to build files from other files. In make, the rules have 3 parts: the target (what we want to build), followed by a colon and a list of ingredients that are required to build the target (so-called dependencies or prerequisites). On the next line, a sequence of shell commands follows. We will get to those in a short while.

The first rule in the file is special, because it is the one that executes when we just 'run' the makefile. It is often called all, so the command then looks as make all or just make. In our case, to make all, we ask make to build getenv but nothing else. This is written so:

```
all: getenv
```

In this case, all is a phony target: it is not a file. Most targets in a makefile are, however, files. We will only deal with files from now on. For instance, let's describe how to build getenv.o from getenv.c:

```
getenv.o: getenv.c
        cc $(CFLAGS) -c getenv.c
```

The 'body' (the shell commands) of the rule is like a small shell script. But you can't use shell variables, because make runs each command in a new shell. Each line of the body starts with a <TAB> character. Using 8 spaces does not work.
Please take note that variable expansion works differently in make than it does in shell: we write $(VAR). In a shell, this would mean 'run the command VAR', but is just standard variable expansion in a makefile. While we talk about variables: make variables are not environment variables. Unlike in a shell, we also can't get environment variables using $-expansion.
Sorry for the detour. Now we also want to tell make how to build getenv from the object file:

```
getenv: getenv.o
        cc -o getenv getenv.o
```

That is all. You should be able to write a simple makefile yourself. Try 3.txt.

Part 3.11: Exercise 3

1. Write the programs (or take them from your solutions from last week) from points 1-6 of ../02/3.txt (i.e. exercise 3 from the last week)
2. Write a simple makefile that tells make how to build hello.o, main.o and bye.o. We will not build a library at this time.
3. Make it so that a binary called hello is built from these 3 object files.
4. Make sure that greetings.h appears in the prerequisites where it is required.
5. Use touch on various source files and check that make does the right thing.

You can now continue to 4.txt.

Part 3.12: Exercise 4

Write a shell script that mimics what the makefile from previous exercise does. That is, compiles the three object files and links them, but only does those things if either the output files do not exist, or the inputs have changed since the outputs have been generated.
You can now continue to suffixes.mk.

Part 3.13: suffixes.mk (source code)

We start with a rule to make getenv again:

```
getenv: getenv.o
        cc -o getenv getenv.o
```

We also add -std=c99 to CFLAGS:

```
CFLAGS += -std=c99
```

Out of the box, make knows how to compile C sources (.c files) into object files (.o files). This time, however, we will tell it how to do that anyway. The traditional way to tell make that .c and .o are suffixes (which are special for make) is this:

```
.SUFFIXES: .c .o
```

If .c and .o are suffixes, we can write special rules (called inference rules) which tell make how to go from any .c file to a corresponding .o file like this:

```
.c.o:
        cc -c $(CFLAGS) -o $@ $<
```

We have used some magic variables there. The $@ is the name of the target to make and is defined in any rule. The $< is the name of the prerequisite (the .c file in this case) and may or may not be available in standard (non-suffix) rules.
Now, since we have the suffix rule, we don't have to tell make how to build getenv.o. Try running those commands:

```
$ rm -f getenv.o getenv
$ make -f suffixes.mk
```

You can now go on to read autodep.mk.

Part 3.14: autodep.mk (source code)

This file is a makefile again. You can run it using make -f autodep.mk.

```
CFLAGS += -std=c99
```

Building programs in C poses some special challenges when header files are involved. Recall that the right hand side of a make rule lists the ingredients that enter the build process. For C programs, this list should contain any header files that are included: if the header file changes, we should rebuild all object files that could possibly be affected by the change. But maintaining such lists is tedious and error-prone. Nobody wants to do it.
Fortunately, compilers and make offer some features to automate this tedious task. When compiler builds an object file, it can also produce a makefile fragment that describes what header files were used in the process. The flags to do that are -MD -MP. This makefile fragment will be stored in a file called file.d (assuming we build file.o). You can read more in man gcc.

```
CFLAGS += -MD -MP
```

Being able to refer to all sources (and object files) that entail a single program is often very useful. We will call the list of sources SRC and the list of object files OBJ. But we are lazy programmers and we do not want to repeat ourselves. We just list the source files and tell make that the object files are the same files, but with .c replaced with .o. Notice the syntax: we use curly braces here. The syntax is ${VAR:old=new} where old and new are suffixes: each whitespace-separated part of the variable is checked and if it ends with the string old, that part is replaced with new in the output.

```
SRC = getenv.c foo.c
OBJ = ${SRC:.c=.o} # getenv.o foo.o
DEP = ${SRC:.c=.d} # getenv.d foo.d, see below for an explanation

getenv: $(OBJ) # we need all the object files
        cc -o getenv $(OBJ)
```

Recall that make already knows how to build .o files from .c files. It also knows to use $(CFLAGS), so it will use our special flags that we specified at the start and cc will generate a .d file for each .c file. This .d file will contain dependency information (in makefile format).
We add another (phony) rule, so we can do make clean to remove all the files we built and start over if we wish.

```
clean:
	rm -f $(DEP) $(OBJ) getenv
```

The final ingredient is the `-include` directive, which causes `make` to include rules from other files. In this sense, it is quite similar to the `#include` directive from C. The `-` at the start means that it is not an error if some of the files do not exist. After you run this makefile, you may want to inspect the file `getenv.d` using a text editor.

```
-include $(DEP)
```

That's it. You should be ready to work out the homework now, described in `hw.txt`.

Part 3.15: Homework

1. We did not have time to properly introduce `tar` and `gzip` yet. Those are archiving programs: `tar` creates a single file (a tarball) from a number of files – it is a bit like `ar` archives (static libraries), but more generic.
2. The `gzip` program is a compression utility. It makes files smaller (usually).
3. If you have a directory with some files in it, you can create a tarball (a single file) as follows:

   ```
   $ tar czf file.tar.gz directory/
   ```

   You can also list the content of an archive, like this:

   ```
   $ tar tzf file.tar.gz
   ```

   To unpack the file, you can use the following command:

   ```
   $ tar xzf file.tar.gz
   ```

   Be careful though! This will overwrite files without asking. Better try decompressing somewhere safe, like in `/tmp`:

   ```
   $ mkdir /tmp/xrockai-untar
   $ cd /tmp/xrockai-untar
   $ tar xzf ~/file.tar.gz
   $ ls -l directory
   ```

4. Please hand in this homework as a tarball. It should contain a single directory named `hw03`.
5. Write a makefile that builds a static library and a program using automatic dependencies (`.d` files should be created during compilation). Use libmath.a from last week, along with the `math.c` we had which used it. The name of the resulting program should be `math`. It should be possible to (at least) run `make libmath.a` and `make math`.
6. The directory `hw03` should contain all the source files that are required to build `libmath.a` and `math`, along with a `makefile`, and nothing else (i.e. only C source files and the makefile). Especially not any binary files.
7. The tarball should be called `hw03.tar.gz` (a different file name will not be accepted by the submission program).

Part 4: Working with Files

1. This seminar will look at files from two perspectives: through user commands (i.e. from a shell) and from C (using file-related system calls).
2. We will look at file management, like directory creation, file removal (unlinking), renaming, copying and so forth.
3. We will try to highlight both the commonalities and the differences of the two viewpoints, and notice what is easy with one but hard with the other.
4. We will always look at the user (shell) side of things first. This part will be a shell script, and will usually also contain a bit of theory in comments.
5. What will follow will be one or at most a few C programs that demonstrate the same concepts at the level of `libc` functions or system calls.

We will start by looking at i-node metadata:

```
$ micro stat.sh
```

Part 4.1: `stat.sh` (source code)

We will first look at (some of the) metadata stored in i-nodes. The simplest way to learn those is the `stat` command. Unfortunately, `stat` is not standardized by POSIX and may be missing on a system or behave differently. Anyway, on Linux it works like this:

```
stat intro.txt
echo
echo compare:
stat /
echo
echo also compare:
stat /dev/null
```

Depending on the file type (printed as the last thing on the second line), there are some interesting bits of info:

1. the size (in bytes)
2. the inode number
3. the number of links pointing at this inode (the reference count, basically)
4. the owner and group (`uid` and `gid`)
5. permission bits (e.g.: `0644` or `-rw-r--r--`)
6. modification, access and change timestamps

We will delve into the details of 4 and 5 in the second half of the semester. The timestamps record at what time last operation of given kind happened with the file. Those are not authoritative: users are allowed to change them (see `man touch`). More detais in `man 1 stat` and also `man 2 stat`.
Well this was somewhat easy. Let's look at the C version:

```
$ micro stat.c
```

Part 4.2: `stat.c` (source code)

We will try to reproduce (part of) what our shell script did in C. If you didn't open `man 2 stat` yet, now would be a good time. We start by using the `#include` statement from the manual page (and add `stdio.h` for a good measure):

```
#include <sys/stat.h>
#include <stdio.h>
```

Let's go straight for `main()`.

```
int main()
{
```

First, we need a place for the OS to put the information about the file(s). This is an aggregate type, with a number of fields in it which correspond to the items printed by the `stat` user command. The type is called `struct stat`:

```
        struct stat st;
```

In the man page, we immediately notice that there is a function called stat which takes a path, and a function called fstat which takes a file descriptor. This is a common pattern. For now, we will stick with the version that uses paths:

```
        if ( stat( "intro.txt", &st ) == 0 )
        {
```

A quick glance at the 'return value' section of the manpage reveals that 0 means success. Anything else would be an error (but the only other thing that we can get should be -1).

```
            printf( "stat() on intro.txt returned OK:\n" );
```

If the above stat call succeeds, it fills in the struct stat that we passed to it. The fields are described in detail in man 2 stat, we will print some of them here:

```
            printf( "size: %lu\n", st.st_size );
            printf( "inode: %lu\n", st.st_ino );
            printf( "links: %lu\n", st.st_nlink );
            printf( "uid: %u\n", st.st_uid );
            printf( "gid: %u\n", st.st_gid );
            printf( "mode: %o\n", st.st_mode ); /* incl. permissions */
        }
        else
```

This would also be a good time to introduce the function perror, which is short for 'print error'. Often, if a system call (like stat above) fails, we want to tell the user that it failed and how. This is what perror does.

```
            perror( "calling stat on intro.txt" );
```

We could do the same thing for another file, but I think you get the point. This was relatively easy, but also hard to compare with the shell code. If all we wanted to do was print the info for the user to read, it was a lot more work in C. If we wanted to act on the information, doing that in shell would be more difficult.

```
    }
```

Before continuing with directories, let's make a very brief detour to errno.c. The perror above should not be called, but we want to know what it does.

Part 4.3: errno.c (source code)

This is a very simple program that demonstrates what perror does (and also introduces errno).

```
    #include <sys/stat.h>
    #include <errno.h>
    #include <stdio.h>
```

The global(-ish) variable errno tells us what went wrong in case a system call or a libc function fails. This is how perror knows what to say. I suggest you take a brief look at man 3 errno. Note that different systems put errno in different sections of the manual. You can also try whatis errno. See also man whatis. Sorry for the digression.

```
    int main()
    {
        struct stat st;

        stat( "/does/not/exist", &st );
```

Please note that errno can only be safely used immediately after the failing call; store it away because we will want to look at it later; errno is of type int.

```
        int error = errno;
```

```
        perror( "stat" ); /* prints an error based on ‹errno› */

        if ( error == ENOENT )
            fputs( "the stat error was ENOENT\n", stderr );
    }
```

Having sorted that out, let's continue with directories:

```
    $ micro directories.sh
```

Part 4.4: directories.sh (source code)

In this shell script, we will try creating and removing directories, and look at listing their content. We already know most of the commands involved, so this will be mainly review.

```
    set -x # let's ask the shell to print commands for us
```

We first create and enter a scratch directory (it is an error if it already exists)

```
    mkdir scratch ; cd scratch
```

The -p switch to mkdir tells it to create any missing parent directories. It is also not an error if the target directory already exists, in this case.

```
    mkdir -p some/subdir
    mkdir -p some
```

We also know how to list directory contents, using ls:

```
    ls some
```

The rmdir command is the inverse of mkdir: it will remove an empty directory; it is an error if the directory is not empty, see:

```
    rmdir some
```

However, those are okay:

```
    rmdir some/subdir
    rmdir some

    cd ..
    rmdir scratch
```

Let's move on to the C programs again: we'll now look at rmmkdir.c. We will skip ls for now because it's a little more complicated. But we will get to it near the end of the seminar today.

Part 4.5: rmmkdir.c (source code)

The system calls to create and remove directories are called the same as the commands we used in shell: mkdir and rmdir. Recall that system calls are described in section 2 of the manual: man 2 mkdir and man 2 rmdir.
Let's start by 'borrowing' the #include directives (adding stdio.h):

```
    #include <sys/stat.h>  /* mkdir */
    #include <sys/types.h> /* also mkdir */
    #include <errno.h>     /* errno */
    #include <unistd.h>    /* rmdir, chdir */
    #include <stdio.h>     /* perror */
    #include <stdlib.h>    /* exit */

    int main()
    {
```

We quickly scan the manpage to learn that, once again, return value 0 (from both mkdir and rmdir) means success. Let's run with that.
For mkdir, the prototype makes it clear that we need to pass in a path (of the directory to make). The second argument is a little more mysterious:

you may remember a similar thing from second assignment, when we called open with O_CREAT. This is essentially the same thing. Let's pass 0777 this time, without explanation for now.

```
    if ( mkdir( "scratch", 0777 ) != 0 )
        perror( "mkdir scratch" ), exit( 1 ); /* let's give up */
```

The C version of cd is called chdir. Details, as usual, in man chdir. Like before, 0 means success.

```
    if ( chdir( "scratch" ) != 0 )
        perror( "chdir scratch" ), exit( 1 );
```

There is no equivalent of mkdir -p in the standard library. We have to do it by hand. Hold on to your hats, errno is coming. The value for 'this file or directory already exists' is EEXIST.

```
    if ( mkdir( "some", 0777 ) != 0 && errno != EEXIST )
        perror( "mkdir some" ), exit( 1 );

    if ( mkdir( "some/dir", 0777 ) != 0 && errno != EEXIST )
        perror( "mkdir some" ), exit( 1 );
```

Now try mkdir some again. It will fail but errno will be set to EEXIST.

```
    if ( mkdir( "some", 0777 ) != 0 && errno != EEXIST )
        perror( "mkdir some" ), exit( 1 );

    if ( rmdir( "some/dir" ) != 0 )
        perror( "rmdir some/dir" ), exit( 1 );

}
```

We keep some of the stuff around, so that you can check that it actually exists by hand. You can then remove the mess by typing:

```
$ rm -r scratch
```

We learned a few things, so let's try an exercise:

```
$ cat 1.txt
```

Part 4.6: Exercise 1

1. This exercise has 2 parts. First, write a script:
   a. create directory scratch and change into it
   b. create directories dir1/sub and dir2/sub/dir
   c. compile the C program from ../remove.c (see point 2 below)
   d. run ./remove dir1 and check that it fails
   e. run ./remove dir1/sub and check that it succeeds
   f. remove the remaining directories using remove in a for loop
2. The program remove.c should do the following:
   a. remove the directory given by its first argument
   b. if this fails, it should print an error and give non-0 exit
   c. otherwise, it should print <directory> removed to stderr

That's it. Let's move to rm.sh.

Part 4.7: rm.sh (source code)

This script demonstrates the rm command, which primarily removes (unlinks) files. It can, however, also remove directories and it can remove entire directory trees recursively. First some simple cases:

```
set -x

touch file # recall that this creates an empty file
ls file    # does it exist?
rm file    # remove it
ls file    # check again
rm file    # this prints an error, since <file> no longer exists
rm -f file # not an error -- this just ensures the file is not there
mkdir dir
rm dir     # this is an error
rm -d dir  # this works (but is non-standard, even if common)

touch file
rm -d file # this also works
```

Now let's try something more complicated (and dangerous!):

```
mkdir -p dir/subdir/stuff
touch dir/subdir/stuff/file
touch dir/file
rm -d dir  # fails
rmdir dir  # also fails
rm -df dir # still fails
rm -r dir  # finally! r is for recursive
```

For extra power (and danger), you could say rm -rf. Always remember that rm does not ask before deleting all your data. If you are unsure about the command, use rm -i, like this:

```
touch some_file
rm -i some_file
```

That's about as much as we need to know about removing files and directories using user-level commands. Let's have a look at unlink.c now.

Part 4.8: unlink.c (source code)

The C interface to removing files is much simpler, and won't let you remove more than one thing at a time. We already know how to remove (empty) directories. How about files? The function is called unlink. Check the man page.

```
#include <fcntl.h>  /* open, O_CREAT */
#include <unistd.h> /* unlink */
#include <stdio.h>  /* perror */
#include <stdlib.h> /* exit */

int main()
{
```

You might recall that you can create new files using open.

```
    if ( open( "file", O_CREAT, 0666 ) < 0 )
        perror( "creat" ), exit( 1 );
```

We don't care about the file descriptor now. However, it is worth noting that our program has the file open at the moment.

```
    if ( unlink( "file" ) != 0 )
        perror( "unlink" ), exit( 1 );

    if ( unlink( "file" ) != 0 )
        perror( "second unlink" );
}
```

We know (almost) enough for another exercise. Here:

```
$ cat 2.txt
```

Part 4.9: Exercise 2

1. Write a C program that behaves just like rm -d path.
2. Assume it only takes 1 argument.
3. There is a hidden challenge in this exercise.
4. Write a shell script that compiles the program and demonstrates that it works as intended.

That said, let's look at hard links and soft links:

```
$ micro links.sh
```

Part 4.10: links.sh (source code)

Let's have some fun with hardlinks and symlinks.

```
set -x

touch file # create a test file
stat file  # let's check how it looks
```

To create a hard link, we can use the command ln (short for link). After a hard link is created, you can no longer tell which link is the original and which is 'the link': they are the same in all respects.

```
ln file hardlink # make ‹hardlink› refer to the same inode as ‹file›
stat file        # observe the link count and the inode number
stat hardlink    # same here
```

We can now unlink the original file and see what happens?

```
rm file
stat hardlink    # this should be unchanged, save for link count
mv hardlink file # get back to where we started (‹mv› renames files)

mkdir dir
ln dir dirlink # this fails, directory hardlinking is forbidden
```

Let's look at soft (symbolic) links now. They are created using ln -s (where s stands for soft/symbolic):

```
ln -s file softlink
ln -s dir dirlink
```

Both these worked, let's look at stat:

```
stat softlink
stat dirlink
```

There is one more command that is interesting with regards to symlinks:

```
readlink softlink
readlink dirlink
readlink dir # fails, ‹dir› is not a symlink

rm -df file dir dirlink softlink
```

Let's look at some weird things that can happen with symlinks too:

```
$ micro badlink.sh
```

Part 4.11: badlink.sh (source code)

Let's look at a few pathologies surrounding symlinks.

```
set -x

touch file            # just an empty file
ln -s file link       # and an innocent symlink

rm file               # get rid of ‹file›, leaving the link behind
ln -s dangling badlink # we can directly create a dangling link
ls -l link badlink    # what does ‹ls› say about those?

cat badlink           # fails, no such file...
ln -s badlink dangling # let's make a loop just for fun
ls -l badlink dangling
cat badlink           # fails in a new and interesting way

rm -d link dangling badlink
```

Here is another, slightly different pathology with loops:

```
mkdir -p dir/subdir
ln -s .. dir/subdir/link
ls dir
ls dir/subdir
ls dir/subdir/link
ls dir/subdir/link/subdir
ls dir/subdir/link/subdir/link
ls dir/subdir/link/subdir/link/subdir

rm -r dir # eww, go away ugly thing
```

Let's move on to the C view of things, in links.c.

Part 4.12: links.c (source code)

The C functions that correspond to ln are link and symlink. There is also readlink. They are all system calls and their manuals live in section 2. Let's #include things we'll need:

```
#include <unistd.h>   /* link, symlink, readlink */
#include <sys/stat.h> /* mkdir */
#include <fcntl.h>    /* open */
#include <stdio.h>    /* perror */
#include <stdlib.h>   /* exit */

int main()
{
    if ( open( "file", O_CREAT, 0666 ) < 0 ) /* make a test file */
        perror( "creating 'file'" ), exit( 1 );
    if ( mkdir( "dir", 0666 ) != 0 )         /* and a directory */
        perror( "creating 'dir'" ), exit( 1 );
```

As usual, we check the exit codes in man pages. For link and symlink, they are the usual 0 = success. But readlink behaves like read, returning the number of bytes, so we have to watch out there.

```
    if ( link( "file", "link" ) != 0 )
        perror( "creating link" ), exit( 1 );

    if ( link( "dir", "dirlink" ) != 0 )
        perror( "creating dirlink" ); /* we expect this to fail */
```

Now let's look at symlinks.

```
    if ( symlink( "file", "symlink" ) != 0 )
        perror( "creating symlink" ), exit( 1 );

    if ( symlink( "dir", "dirlink" ) != 0 )
        perror( "creating (symbolic) dirlink" ), exit( 1 );

    char buffer[ 60 ] = { 0 };
    if ( readlink( "dirlink", buffer, 60 ) < 0 )
        perror( "readlink dirlink" ), exit( 1 );

    printf( "readlink on dirlink gave %s\n", buffer );
    if ( readlink( "dir", buffer, 60 ) < 0 )
        perror( "readlink dir" );
}
```

We do not clean up so that you can check the resulting files and links with stat or ls -l. However, you will have to clean up by hand. I am sure you know how (you can use rm -d). That was links in C. Please go on to 3.txt.

Part 4.13: Exercise 3

Write a C program called destroy, that:

1. takes a single filename as an argument,
2. unlinks the file; if it is a symlink, also unlinks the target,
3. prints whether the file is gone, or some hardlinks remain.

There are a few improvements that could be done. See 4.txt.

Part 4.14: Exercise 4

Improve the solution of the previous exercise in following ways:

1. If the target of the symlink is a symlink, follow that too, until you find a non-symlink,
2. if the symlinks form a loop, say so and give up.

Solving this requires C skills on top of what we have already done. If you are new to C, feel free to skip this exercise.
In any case, the next item to look at is readdir.c.

Part 4.15: readdir.c (source code)

To do what ls does, we need a set of somewhat complicated C functions. They are like open and read but for directories, and are called opendir and readdir accordingly. Please check the man pages: they are in section 3. The readdir in section 2 is an obsolete system call on Linux, please ignore it!

```
#include <dirent.h> /* opendir, readdir */
#include <stdio.h>  /* puts */
#include <stdlib.h> /* exit */

int main()
{
```

Let's open the current working directory (cwd for short). The manual suggests that on failure, cwd will be a NULL pointer.

```
    DIR *cwd = opendir( "." );

    if ( !cwd )
        perror( "opendir ." ), exit( 1 );
```

We have cwd open and we can read the individual directory entries. The dirp below is a pointer to a single directory entry.

```
    struct dirent *dirp;

    while ( 1 ) /* forever */
    {
        dirp = readdir( cwd ); /* obtain the next directory entry */
```

If readdir returns NULL, it means that there are no more directory entries (i.e. we read all of them). In that case, let's stop the loop using break here.

```
        if ( !dirp )
            break;
```

There are a few items in the directory entry (like the inode number), but we only want to print the name for now. We can use puts for that.

```
        puts( dirp->d_name );
    }
```

That's it. This program, when compiled and executed, will print the names of all files and directories in its working directory (in no particular order).

```
}
```

Please note that in modern programs, we would most likely use readdir_r instead of readdir, because there might be threads in such programs, and readdir doesn't work very well with those. But we won't worry about that yet.
This was the last bit I wanted to show you. Only the homework remains, in hw.txt.

Part 4.16: Homework

1. Implement list.c, an extended version of readdir.c, which prints the contents of the working directory.
2. Each entry goes on a single line. The format is as follows (each column is separated from the next by a single space):
   a. one character, f for files, d for directories, l for symlinks
   b. the inode number
   c. the number of links
   d. the size of the file (0 if not applicable)
   e. the file name
   f. if the file is a symlink, ... target (otherwise nothing)

That's it for today. Have a nice day!

Part 5: Processes

1. Today, we will look at processes and executing programs.
2. We will first look from the user perspective, using shell scripts.
3. Then we will move on to the C API: fork and exec.

We will start with

```
$ micro proc.sh
```

Part 5.1: proc.sh (source code)

We will first learn how to run a process in the background and how to get its process ID (pid). The first step is running a command without waiting for the result, which is achieved by ending the command with a single ampersand, like this:

```
while true; do sleep 1; echo .; done &
```

Now the entire compound command is running in a separate process (the shell has forked): the parent processes continues to execute this script (i.e. the commands below) while the child runs the command above (it runs in the background, but it can still print to the terminal).
To do anything useful with the running process, we need to learn its process identifier, which is stored in the shell variable $!.

```
child=$!
echo $child
```

Then we wait a bit, so that the loop above can print a few lines. The sleep command, as you probably figured out by now, does nothing for a given number of seconds (5 in this case).

```
sleep 5
```

Now we request that the child process terminates. This is a 'nice' request, in the sense that the operating system notifies the process that the user asked for its termination. The process may or may not honour the request (it usually will, but it may perhaps try to clean things up before terminating).

```
kill $child
```

As usual, you can look up kill in the manual. That's it, we will look at a few more process management commands now:
$ micro ps.sh

Part 5.2: ps.sh (source code)

Now that we know how to run a process from shell, we will look at a few more commands that work with processes. So far, we have only seen kill, which, given a process ID, sends a signal to the process. You can learn more about signals in man 7 signal – unfortunately, we won't have time to get into details in the seminar.
First we will create a process and get its pid, like we did before:

```
chmod +x spin.sh
./spin.sh &

set -x

pid=$!
```

The first command that I want to show you is pgrep, which allows you to find processes by looking at the commands those processes are executing. Of course, you can look at man pgrep. Without -f, it checks only the command itself, with -f it will also match on the command line arguments. To make things a little more complicated, executable scripts sometimes have their filename as the command name, and sometimes the interpreter name, depending on the operating system. On aisa, it is the former, so the first command will work as we expect.

```
pgrep spin.sh
pgrep -f spin.sh
```

The pgrep command has a companion called pkill, which does the same thing but instead of printing the process id's of the matching processes, it kills them (sends them a 'please terminate' signal, also known as TERM). Killing processes forcibly can be done with the -KILL switch to either kill or pkill. It is usually better to not use -KILL.
The last user command for process management that you need to know is ps, which is short for 'processes' and simply prints a list of all processes. By default, only interactive processes owned by you are shown:

```
ps
```

If we want to see more details, we can add u to the options. Notice that unlike most other unix programs, the switches to ps can be specified without the dash.

```
ps u
```

But if you are more comfortable with the dash, you can have it, too:

```
ps -u # the same as the previous command
```

We can request to also see processes that do not have a controlling terminal.

```
ps -x
```

And finally, we can also request to see all the processes in the system with ps -a, but we won't run that in the script because there are many. You can try running it by hand if you like. Another command to try yourself would be top, which interactively shows current processes in the system and tells you how much CPU and memory they use.
Okay, let's wait for a bit to check our process is running. Be sure to notice our process in the process listings from ps.

```
sleep 3
kill $pid
```

Let's do an exercise, 1.txt.

Part 5.3: Exercise 1

1. Write a shell script that:
   a. creates a background process that writes a line with a dot into file.txt every second
   b. create another process that counts the lines in file.txt and prints the count to the terminal every second
   c. use ps -u and grep to print information about both processes every second for 10 seconds
   d. kill both background tasks

That is all for the user-level part. Let's move on to C:

```
$ micro fork.c
```

Part 5.4: fork.c (source code)

In the lecture, we said that new processes are created with fork. We will try that now. We will use fork to create a process and print a message in each of the processes. As usual, we need a few #include statements (see also man fork).

```
#include <unistd.h> /* fork */
#include <stdio.h>  /* printf */
```

This is almost the simplest program possible that uses fork(). We just print a message to the screen before and after the fork.

```
int main()
{
    printf( "before fork()\n" );
```

fork takes no arguments and returns a pid_t, which is an integer with platform-defined size (big enough to hold any valid process id)

```
    pid_t pid = fork();
```

The following message will appear twice, once from the parent process and once from the child process. In the parent process, fork() returns the pid of the newly created child. In the child process, it returns 0. If an error happens, it returns -1 and no process is created: hence, in this case, fork() only returns once.

```
    printf( "after fork(), result = %d\n", pid );
```

Be sure to understand the output of this program. We will use fork() in a more complicated scenario shortly.

```
}
```

Let's see one more example before moving on to exercises:

```
$ micro wait.c
```

Part 5.5: wait.c (source code)

In this program, we will learn how to check the result of the child process. Then we will move on to an exercise that will use fork().

```
#include <unistd.h>   /* fork */
#include <sys/wait.h> /* waitpid */
#include <stdio.h>    /* printf */

int main()
{
    puts( "entered main()" );
    pid_t pid = fork();

    if ( pid < 0 )
    {
```

Things went south; you can look at `man fork` to check what could cause `fork` to fail (specifically section ERRORS near the end of the manpage).

```
        perror( "fork" );
        return 1;
    }

    if ( pid == 0 )
    {
        puts( "in the child process" );
        return 13; /* terminate with exit code 13 */
    }

    if ( pid > 0 )
    {
        printf( "in the parent process, child pid = %d\n", pid );
```

The child process is running, but we don't know when it finishes or how. To find out, we can use the `waitpid` system call. It is only allowed to call `waitpid` on processes that are direct descendants (via `fork`) of the current one. As usual, see `man waitpid`. Besides waiting for the child process to finish, we can learn the exit status, or whether the process exited normally or was killed.

```
        int status;
        if ( waitpid( pid, &status, 0 ) == -1 )
        {
            perror( "waitpid" );
            return 1;
        }
```

The integer `status` passed to waitpid is filled in with information about how the process terminated. Unfortunately, it is a single integer, not a structure like `struct stat`, which makes things a little odd and complicated. We have to use macros called `WIFEXITED` and `WEXITSTATUS` to learn whether the program terminated normally and what was the exit code. The remaining macros (described in `man waitpid`) deal with signals – you can study those on your own, or perhaps in a more advanced course on POSIX programming.

```
        if ( WIFEXITED( status ) )
            printf( "child terminated with return code %d\n",
                    WEXITSTATUS( status ) );

        return 0; /* terminate with exit code 0 */
    }
}
```

Now we are ready for an exercise: check `2.txt`.

Part 5.6: Exercise 2

1. Write a C program that uses `fork()` to create two processes. The parent:
   a. creates a file called `file1.txt`
   b. loops until `file2.txt` appears – you can use `stat` and `sleep( 1 )`
   c. unlinks both `file1.txt` and `file2.txt`
   d. waits for the child process and prints its exit code
2. The child:
   a. waits until a file called `file1.txt` appears
   b. creates a file `file2.txt`
   c. exits with a code of your choosing
3. Print a message in both the parent and the child after each step (identifying which process is talking).

That's that. We are ready for the final chapter for today, and that is `exec`:

```
$ micro exec1.c
```

Part 5.7: `exec1.c` (source code)

There are multiple variants of the `exec` function (none of which is actually called `exec`). There are a few suffixes that can be given to `exec`:

- `l` is for list (of arguments)
- whereas `v` is for `varargs` (we won't need to deal with those yet)
- the suffix `p` means 'use `PATH` to find the command' and
- the suffix `e` means 'let me pass a new environment to the program'.

For now, we will just use `execl`. As is easily checked in `man execl`, we need to include `unistd.h`:

```
#include <unistd.h> /* execl */
#include <stdio.h>  /* puts */

int main()
{
```

We just print a 'hello' to make it clear we are running.

```
    puts( "hello from main" );
```

Now we use `execl` to replace the current process with another program, and pass a few arguments to that program. The first argument is the path to the program to execute, the rest of the arguments is what becomes `argv` of the new `main()` in the new program. Remember that `argv[0]` should be the name of the program itself: we need to pass that explicitly! The last argument must be NULL, to let the function know when to stop looking for more arguments. Notice that we do not pass `argc` – the arguments are counted by the system.

```
    if ( execl( "/bin/echo", "echo", "hi from echo", NULL ) == -1 )
    {
        perror( "exec" );
        return 1;
    }
```

You may have read in `man execl` that 'The exec() functions return only if an error has occurred.' This is because if they work as intended, the program executing in this process has been replaced, and is no longer this program. The following line will therefore never execute – when the new program ends, the process terminates as usual.

```
    puts( "back to main" );
}
```

Let's now look at the usual combination of `fork` + `exec`, which lets us execute a program and wait for it to finish. The code is in `exec2.c`.

Part 5.8: `exec2.c` (source code)

This program demonstrates the code for 'running' a program in the usual sense: create a new process and then use `exec` to replace the program in the process with a new one. As usual, includes first:

```
#include <unistd.h>   /* fork, execl */
#include <stdio.h>    /* printf */
#include <sys/wait.h> /* waitpid */
#include <stdlib.h>   /* exit */

int main( int argc, const char **argv )
{
    puts( "hello from main" );
    if ( argc != 2 )
        puts( "please pass in a path to the program to execute" ),
exit( 1 );
```

Let's start by creating a new process using `fork()`.

```
    pid_t pid = fork();

    if ( pid < 0 ) /* don't forget to check for errors */
        perror( "fork" ), exit( 2 );

    if ( pid > 0 ) /* parent */
    {
        puts( "hello from parent" );
        int status; /* to hold the exit status of the child */
```

We wait for the child process to terminate and get the status.

```
        if ( waitpid( pid, &status, 0 ) == -1 )
            perror( "waitpid" ), exit( 2 );
```

And print the result to the user.

```
        if ( WIFEXITED( status ) )
            printf( "child terminated with exit code %d\n",
                    WEXITSTATUS( status ) );
    }

    if ( pid == 0 ) /* child */
    {
```

As you have seen in exec1.c, we need to pass the program name twice to execl. We don't want any further arguments, so we just pass the terminating NULL.

```
        if ( execl( argv[1], argv[1], NULL ) == -1 )
            perror( "exec" ), exit( 2 );
```

We never get here.

```
    }
}
```

There are a few things we can try with this program:

```
$ ./a.out /bin/echo
$ ./a.out /bin/false
$ ./a.out /bin/true
$ ./a.out /does/not/exist
$ ./a.out true
```

Let's do an exercise with fork() and exec(): 3.txt.

Part 5.9: Exercise 3

1. Write a C program that forks into two processes.
2. The parent waits for 3 seconds and then kills the child.
3. The child executes ./spin.sh.

That wasn't too hard, was it? The homework is described, as usual, in hw.txt.

Part 5.10: Homework

1. Write a C program sync.c and a shell script sync.sh.
2. The C program should fork() & the child should then exec() the script.
3. The parent process:
   a. waits until file1.txt appears
   b. when it does, it copies its content into file2.txt
   c. waits for the script to finish and prints its exit code
   d. use POSIX read() and write() above
4. The shell script:
   a. prints its own pid (found in $$) into file1.txt
   b. waits until file2.txt appears
   c. waits a second to make sure the content is written
   d. reads file2.txt and checks that its content matches
   e. if it matches, ends with exit code 0, otherwise 1
   f. the shell built-in exit 0 and exit 1 can be used for (e)

That is all! Or is it.

Part 6: IPC (Inter-Process Communication)

1. In this lecture, we will deal with how processes can interact with each other (in ways more useful than just using kill on each other).
2. We will start with anonymous pipes, because we have already seen those.
3. There are also named pipes, although we will not get to use them in this seminar.
4. We will, however, look at sockets (both anonymous and named).
5. Before you start working on this seminar, you may want to review shell pipes as described in ../01/pipes.txt.

The first thing we will look at today is what happens to file descriptors when we fork (after that, we will look at using anonymous pipes in C):

```
$ micro forkfd.c
```

Part 6.1: pipes.sh (source code)

This script simply serves to remind you how anonymous pipes work in shell.

```
TBD
```

Part 6.2: forkfd.c (source code)

Since we want to talk about IPC, we will need multiple processes. Hence, we will need fork() to create those processes. If you don't remember how fork() works, plase check out ../05/fork.c (and the following units) again.

```
#include <unistd.h> /* fork, read */
#include <stdio.h>  /* printf */
#include <fcntl.h>  /* open */

int main()
{
```

The purpose of this C program is to demonstrate how file descriptors are inherited across a call to fork. Therefore, we will first open a file for reading.

```
    int fd = open( "intro.txt", O_RDONLY );

    if ( fd < 0 )
        return perror( "open intro.txt" ), 1;
```

And call fork, to create a second process.

```
    pid_t pid = fork();

    if ( pid < 0 )
        return perror( "fork" ), 1;
```

You might remember from the lecture that file descriptors are per-process (and this is indeed true). However, fork creates an identical copy of a process, including all its open file descriptors.
Two processes are running from this point on. The read call is performed twice, each in one of the processes, but the fd is still shared:

we will read different parts of the file.

```
    char buf[ 36 ] = { 0 };
    if ( read( fd, buf, 35 ) != 35 )
        return perror( "read" ), 1;
```

We will print the process id of each process that made it this far, by using the getpid function, which just returns the pid of the calling process. As usual, see man getpid.

```
    printf( "pid = %d, buf = %s\n", getpid(), buf );
}
```

Now that we understand how file descriptors pass from the parent to the child process during fork, we can move on to creating anonymous pipes:

```
$ micro pipe.c
```

Part 6.3: pipe.c (source code)

We reminded ourselves about fork and explained what it means for file descriptors. We will set that aside for a while, because we only need 1 process in this unit, where we explain the pipe system call. In the next part, we will combine what we know about both to build a simple IPC mechanism.

```
#include <unistd.h>  /* pipe, read, write */
#include <stdio.h>   /* printf */

int main()
{
```

To create an unnamed pipe, we will need 2 file descriptors: one for the reading end and one for the writing end. Let's prepare memory for that: what follows is an array of 2 integers.

```
    int fds[ 2 ];
```

Now the creation of the pipe: unlike open, the pipe syscall does not return a file descriptor, because it actually needs to return 2 of those. The return value only indicates success (0) or error (-1). For the file descriptors, it uses the same approach as stat: we pass a pointer to memory we prepared above.

```
    if ( pipe( fds ) == -1 )
        return perror( "pipe" ), 1;
```

The file descriptors are not interchangeable. One is the reading end (from which you can only read) and the other is the writing end (to which you can only write). A pipe is a unidirectional device – data only flows in one direction. To make the remaining code more readable, we will give names to the ends.

```
    int read_fd = fds[ 0 ], write_fd = fds[ 1 ];
```

Let's write something (the string hello) into the write end now.

```
    if ( write( write_fd, "hello", 5 ) != 5 )
        return perror( "write" ), 1;
```

We try reading from the read end to see what comes out. Knowing how pipes work, we expect hello to be the result.

```
    char buf[ 16 ] = { 0 };
    if ( read( read_fd, buf, 5 ) != 5 )
        return perror( "read" ), 1;

    printf( "we got \"%s\" from the pipe\n", buf );
    return 0;
}
```

We can go on now to actual IPC, see pipe-ipc.c.

Part 6.4: pipe-ipc.c (source code)

```
#include <unistd.h>  /* fork, pipe, read, write */
#include <stdio.h>   /* printf */

int main()
{
```

We already know how to create a pipe. Let's just repeat the process.

```
    int fds[ 2 ];

    if ( pipe( fds ) == -1 )
        return perror( "pipe" ), 1;

    int read_fd = fds[ 0 ], write_fd = fds[ 1 ];
```

Like before, we keep the pipe ends as read_fd and write_fd.

```
    pid_t pid = fork(); /* fork & check for errors */
    if ( pid == -1 )
        return perror( "fork" ), 1;

    if ( pid > 0 ) /* parent */
    {
```

The first thing we should do is close the end of the pipe that we are not going to use. This is important, because closing the write end of the pipe is the only way to indicate the end of file to the read end. And overall, it is just bad form to keep unused fd's lying around.

```
        close( read_fd );
```

Having that sorted, we just write a message for the child to read.

```
        if ( write( write_fd, "hello", 5 ) != 5 )
            return perror( "write" ), 1;

        close( write_fd ); /* indicate that we are done */
    }

    if ( pid == 0 ) /* child */
    {
        close( write_fd ); /* see parent */
```

there should be a message by the parent, let's try to read it

```
        char buf[ 6 ] = { 0 };
        if ( read( read_fd, buf, 5 ) != 5 )
            return perror( "read" ), 1;
        printf( "child obtained message: %s\n", buf );
```

That should have worked okay. Now let's check that there is no more data in the pipe (because the parent closed the writing end).

```
        if ( read( read_fd, buf, 1 ) == 0 )
            printf( "child got eof on the pipe, as expected\n" );
        else
            printf( "oops, unexpected data in the pipe!\n" );
    }

    return 0;
}
```

It is time to work out an exercise, see 1.txt.

Part 6.5: Exercise 1

1. Write a C program that creates 2 pipes.
2. Then it forks off 2 child processes.
3. Sends each of them a hello through the pipe.

4. The children both print their pid and the message.

Now let's learn about dup and dup2, so that we can replicate what shell does when we use |, but in C. That is, we will use pipe, fork and exec together. See dup.c.

Part 6.6: dup.c (source code)

The familiar preamble:

```
#include <unistd.h> /* dup, read */
#include <fcntl.h>  /* open */
#include <stdio.h>  /* printf */

int main()
{
```

The dup family of functions manipulates (duplicates) file descriptors. We will first try dup on a file and then on stdout.

```
    int fd1 = open( "intro.txt", O_RDONLY );
```

The dup call makes a copy of fd1, with a new number.

```
    int fd2 = dup( fd1 );
```

We will need two buffers for reading data.

```
    char buf1[ 22 ] = { 0 }, buf2[ 15 ] = { 0 };
```

Now we will read from the file descriptors in turn. The man page for dup says that both descriptors share the same 'open file' object (the same entry in the table of open files). This means, among other things, that the read position is shared between them.

```
    if ( read( fd1, buf1, 21 ) != 21 )
        return perror( "read fd1" ), 1;

    printf( "reading from fd1: %s\n", buf1 );

    if ( read( fd2, buf2, 14 ) != 14 )
        return perror( "read fd2" ), 1;

    printf( "reading from fd2: %s\n", buf2 );
```

However, closing one of the descriptors does not affect the other. You could read from fd2 after closing fd1, for instance. We will see that later.

```
    if ( close( fd1 ) )
        return perror( "close fd1" ), 1;
    if ( close( fd2 ) )
        return perror( "close fd2" ), 1;

    return 0;
}
```

Let's check out dup2 now, in dup2.c.

Part 6.7: dup2.c (source code)

First some includes:

```
#include <unistd.h> /* dup2 */
#include <fcntl.h>  /* open */
#include <stdio.h>  /* puts */

int main()
{
```

We start by opening an output file.

```
    int fd = open( "dup2_out.txt", O_WRONLY | O_CREAT, 0666 );
```

```
    if ( fd == -1 )
        return perror( "open" ), 1;
```

The only difference between dup and dup2 is that in the latter, you get to choose the number that will be used for the copy of the file descriptor. If that number is an existing fd, the old one will be closed and replaced by the new one. Try to think about what you would expect the following dup2 call to do.

```
    if ( !dup2( fd, 1 ) )
        return perror( "dup2" ), 1;
```

At this point, stdout (fd number 1) was replaced by a copy of fd, which points to a file open for writing. Let's use familiar puts to print something to stdout. Normally, that would go to screen, but now it should probably end up in the file dup2_out.txt?

```
    puts( "hello world" );
}
```

Now we are ready to move on to another exercise, 2.txt.

Part 6.8: Exercise 2

1. First, a hint: you will want to use execvp(argv[1], argv + 1) to execute a program along with arguments given to you on the command line.
2. If you prefer a complete example, check out execvp.c.
3. Write a C program that uses pipe and dup2 to send the string "hello\nworld\n" as the input to a command given to it as described in point 1 above.
4. Check that the program works as expected by running ./a.out cat and ./a.out wc -l.

We can now move on to sockets, see

```
$ micro socketpair.c
```

Part 6.9: socketpair.c (source code)

This program demonstrates the use of the socketpair function. The prototype is in sys/socket.h:

```
#include <unistd.h>     /* fork, read, write */
#include <sys/socket.h> /* socketpair */
#include <stdio.h>      /* printf */

int main()
{
```

Socket pairs are basically just like pipes, but bidirectional. The interface to create them is similar, but there are some new elements. However, the idea with two file descriptors is unchanged.

```
    int fds[ 2 ];
    char buf[ 6 ] = { 0 };
```

There are some extra arguments when we compare socketpair to pipe. The first is so-called address family. There are basically 3 that we could consider, those are AF_UNIX (unix domain sockets), AF_INET (internet protocol, IP) and AF_INET6 (IPv6). The first type, AF_UNIX, is a local socket, that connects programs that run on the same computer. It is the only valid option for socketpair. The other two types connect programs across a network, and we will deal with them in a later lecture.
The second new bit is socket type, of which there are two: stream sockets (SOCK_STREAM, which behave like pipes, i.e. as byte streams) and datagram sockets – SOCK_DGRAM, which are essentially a message-passing device (datagram sockets send entire packets, not individual bytes).
The third option to socketpair is protocol – the only reasonable value here is 0, which lets the system choose something appropriate.

```
    if ( socketpair( AF_UNIX, SOCK_STREAM, 0, fds ) == -1 )
        return perror( "socketpair" ), 1;
```

For socketpair, the file descriptors are equivalent – there is no distinction between a writer and a reader.

```
    int parent_fd = fds[ 0 ], child_fd = fds[ 1 ];

    pid_t pid = fork(); /* fork & check for errors */
    if ( pid == -1 )
        return perror( "fork" ), 1;

    if ( pid > 0 ) /* parent */
    {
```

Like with a pipe, we will close the end of the socket pair that is not used by this process.

```
        close( child_fd );
```

Having that sorted, we just write a message for the child to read.

```
        if ( write( parent_fd, "hello", 5 ) != 5 )
            return perror( "write" ), 1;

        if ( read( parent_fd, buf, 5 ) != 5 )
            return perror( "read" ), 1;

        printf( "parent obtained message: %s\n", buf );

        close( parent_fd ); /* clean up */
    }

    if ( pid == 0 ) /* child */
    {
        close( parent_fd ); /* see parent */
```

There should be a message by the parent, let's try to read it:

```
        if ( read( child_fd, buf, 5 ) != 5 )
            return perror( "read" ), 1;

        printf( "child obtained message: %s\n", buf );

        if ( write( child_fd, "world", 5 ) != 5 )
            return perror( "write" ), 1;

        close( child_fd );
    }

    return 0;
}
```

Let's look at a 'real' socket now. See client.c.

Part 6.10: client.c (source code)

This program will serve as the client side to an AF_UNIX socket. All it does is connect to the socket and send a simple message. The content of the message is given by the command line.

```
#include <unistd.h>     /* write */
#include <stdio.h>      /* perror */
#include <string.h>     /* strlen */
#include <sys/socket.h> /* socket, connect */
#include <sys/un.h>     /* sockaddr_un */

int main( int argc, const char **argv )
{
    if ( argc != 2 )
        return fputs( "need exactly 1 argument\n", stderr ), 1;
```

Sockets are symmetric, in the sense that the first thing we always do is create a socket. That socket can then become a server or a client, depending on what happens next. So let's create a socket: the arguments should now be familiar, because they are exactly the things that we passed to socketpair just a short while ago.

```
    int sock_fd = socket( AF_UNIX, SOCK_STREAM, 0 );

    if ( sock_fd == -1 )
        return perror( "socket" ), 2;
```

The socket itself is not very useful. To talk to some other program, we need to connect to its listening socket, and for that, we need to tell the system the address on which the server is listening. Addresses for AF_UNIX sockets are described by structures of the type sockaddr_un.

```
    struct sockaddr_un sa = { .sun_family = AF_UNIX,
                              .sun_path = "./socket" };
```

The above describes the address, now let's ask the socket to connect to it. The function to do that is called connect, and takes the socket file descriptor and the address, like this:

```
    if ( connect( sock_fd, ( struct sockaddr * ) &sa,
                  sizeof( sa ) ) == -1 )
        return perror( "connect to ./socket" ), 2;

    size_t bytes = strlen( argv[1] ) + 1;
    if ( write( sock_fd, argv[1], bytes ) != bytes )
        return perror( "write" ), 2;

    close( sock_fd );
    return 0;
}
```

Before compiling and executing this program, we will need also the server part. So let's read that first and then we can try both pieces:

```
$ micro server.c
```

Part 6.11: server.c (source code)

This is the server part of an example client-server system. It creates a socket, listens on it and prints any messages it receives from clients.

```
#include <unistd.h>     /* read */
#include <stdio.h>      /* perror, printf */
#include <string.h>     /* strcpy */
#include <sys/socket.h> /* socket, connect */
#include <sys/un.h>     /* sockaddr_un */

int main( int argc, const char **argv )
{
```

Like before, we start by creating a socket.

```
    int sock_fd = socket( AF_UNIX, SOCK_STREAM, 0 );
    if ( sock_fd == -1 )
        return perror( "socket" ), 1;
```

To become a server, we also need an address, but this time it will be the address to listen on. The code to prepare the address is, however, the same as before.

```
    struct sockaddr_un sa = { .sun_family = AF_UNIX,
                              .sun_path = "./socket" };
```

Now the differences start. Instead of calling connect, we will call bind which tells the system to turn this socket into a server socket with the given address. First, however, let's unlink any previous sockets that may have been left behind.

```
    unlink( "./socket" );
```

Now for the bind itself.

```
    if ( bind( sock_fd, ( struct sockaddr * ) &sa, sizeof( sa ) ) )
        return perror( "bind ./socket" ), 1;
```

The socket is bound and we can listen for connections on it now. The function to do that is called, unsurprisingly, listen. This is a simple function that allows clients to start connecting to the above address. The second parameter says how many clients can 'queue up' to talk to us. We'll understand why in a short while.

```
    if ( listen( sock_fd, 5 ) )
        return perror( "listen" ), 1;
```

The final ingredient is called accept, and is a little tricky, because there might be multiple programs trying to talk to us at once. We may call accept multiple times, and every time we do that, we get a new file descriptor, which behaves like one end of a socketpair (the other end being the socket that the other program called connect on). The accept call will wait for a new connection (i.e. it will not return until there is a client that wants to talk to us).
If this was an internet socket, we might be interested in the address of the connecting client. For unix sockets, this is not very interesting, and hence we pass NULL to accept (but you can see the man page for what those 2 parameters mean).

```
    int client_fd;
    char buffer[ 64 ];
    while ( ( client_fd = accept( sock_fd, NULL, NULL ) ) >= 0 )
    {
```

The return value of accept is a file descriptor, so let's try reading some data (the example client just sends us a message).

```
        if ( read( client_fd, buffer, 64 ) >= 0 )
            printf( "received: %s\n", buffer );
        else
            perror( "read" );
```

We clean up the connection and accept another client.

```
        close( client_fd );
    }
}
```

That is all for theory today, let's work out another exercise: 3.txt.

Part 6.12: Exercise 3

1. You will need the execvp trick that we used in exercise 2.
2. Connect to the server that we already have (in server.c).
3. Send the output of the program given in arguments to the server.
4. Check that things work as expected, e.g.: ./a.out ls

The homework is specified in hw.txt.

Part 6.13: Homework

1. Write a C program, listen.c, which listens on a Unix socket:
   a. the name of the socket should be ./socket
   b. repeatedly accept connections from clients
   c. each client will send a single null-terminated path
   d. the server will execute this program in a new process
   e. the path buffer should be at least 200 bytes
2. As a test client, you can use client.c from the exercise.

Part 7: Threads, Synchronisation

1. We will look at threads that run within a single process today.
2. To create and work with threads, we will use the pthread API.
3. In addition to thread creation and management, there is a number of pthread functions that implement synchronisation primitives.
4. We will try out communication using shared memory (global variables).
5. In the second half, we will try mutexes, semaphores and barriers.

Let's start by creating a thread:

```
$ micro thread.c
```

Part 7.1: thread.c (source code)

We will need a new (unfamiliar) header file: pthread.h.

```
#include <pthread.h> /* pthread_create, pthread_join */
#include <stdio.h>   /* puts, perror */
#include <unistd.h>  /* sleep */
```

Let us first declare a function which will become the 'main' function of a thread which we will create in main(). However, we want to first look at how main works, so we will define the function thread later.

```
void *thread( void * );
```

The main function is the same as always – using threads does not affect the prototype.

```
int main()
{
```

Threads have numeric identifiers, just like processes do. For a process, the numeric identifier is called a pid (as you may recall from last two weeks), and is of the type pid_t. For threads, the identifier is usually called tid (for thread id) and is of the type pthread_t.

```
    pthread_t tid;
```

We'll print out some messages to see what is going on in the program.

```
    puts( "main, before pthread_create" );
```

To create a new thread, we will use the function pthread_create (you can, as always, look into man pthread_create for details). This function is quite different from fork: it returns only once, in the main thread (i.e. the parent). The child will start executing the function passed to pthread_create as the third argument (in this case, the function thread that we declared above).
The remaining arguments to pthread_create are: an address of a variable that will hold the thread identifier of the newly created thread (this is the first argument), then come optional attributes of the thread (we will not need to set any of those) and finally a void * value (a pointer of an unspecified type), which is passed as the argument to the function given by the third argument. Since we do not need to pass anything to the thread, we will just use NULL for now.

```
    if ( pthread_create( &tid, NULL, thread, NULL ) )
        return perror( "pthread_create" ), 1;
```

Let's print some messages and sleep for a bit in between, so that we can clearly see that the newly created thread and the main thread execute concurrently.

```
    puts( "main, after pthread_create" );
    sleep( 2 );
    puts( "main also still here" );
    sleep( 1 );
    puts( "about to pthread_join" );
```

Now that we have done all that we wanted, we should synchronize with the thread, i.e. wait until it finishes execution. The pthread_join call is the thread equivalent of the waitpid syscall, which waits for a child process to finish. That is, pthread_join will wait until the thread specified by the first argument stops executing. An optional return value from the thread's main function can be obtained via the second argument. We will use this later.

```
    if ( pthread_join( tid, NULL ) )
        return perror( "pthread_join" ), 1;

    return 0;
}
```

That is all for the main thread, now let's look at the child. Its execution starts with the function thread below the moment the main thread calls pthread_create above. The thread has its own stack, but otherwise, it has access to the same memory as the main thread. We will also look at this later in more detail.

```
void *thread( void *arg )
{
```

The code in the child thread is quite boring: all it does is print a few messages and sleep some in between. Then it returns NULL, which the main thread discards anyway.

```
    puts( "thread" );
    sleep( 1 );
    puts( "thread still here" );
    sleep( 1 );
    puts( "about to end" );
    return NULL;
}
```

Unlike all the C programs until this point, this one needs a little more than just the standard C library (libc). For this reason, we need to compile it with an additional switch (this will hold for all programs that we encounter today):

```
$ cc -pthread thread.c
```

However, the resulting program can be executed as usual – ./a.out. You can examine the output for yourself, and we can move on to shared memory:

```
$ micro shvar.c
```

Part 7.2: shvar.c (source code)

The usual ensemble of header files:

```
#include <pthread.h> /* pthread_create, pthread_join */
#include <stdio.h>   /* printf, perror */
#include <unistd.h>  /* sleep, usleep */
```

From this point on, we will start our programs by writing the child thread and the main function will come at the end. However, before we do that, let's declare a global variable that we will use to demonstrate how memory is shared between threads.
We will use a volatile keyword here, even though doing so is quite incorrect. The reason for this is that until C11, which is not yet available on aisa, there is no good way to write the code we want. Therefore we use volatile and hope for the best.

```
volatile int x = 0;
```

Our thread will be very simple: it will print the value of x, sleep for a bit, then print the value of x again.

```
void *thread( void *arg )
{
    printf( "x = %d\n", x );
    sleep( 1 );
    printf( "x = %d\n", x );
    return NULL;
}

int main()
{
    pthread_t tid;
    x = 17; /* start by setting the value of x */
```

We first start the thread, the same way we did before.

```
    if ( pthread_create( &tid, NULL, thread, NULL ) )
        return perror( "pthread_create" ), 1;
```

Now let's wait for a bit, so the thread has a chance to run. This is definitely not something you would do in a normal program. The sleep here does not guarantee that the other thread will run, or that it will get past the first printf even if it does. To demonstrate how things work, however, it's an okay thing to do.

```
    usleep( 200000 ); /* 200ms */
```

Change the value of x.

```
    x = 33;
    printf( "main changed x to 33\n" );
```

And wait for the thread to finish.

```
    if ( pthread_join( tid, NULL ) )
        return perror( "pthread_join" ), 1;

    return 0;
}
```

This program has demonstrated that global variables are shared between different threads. For fun and profit, you should try to do the same thing with processes, to convince yourself how threads are different from processes (and how shared memory is different from isolated, per-process memory). In fact, this is the first exercise, in 1.txt.

Part 7.3: Exercise 1

1. Take the program in shvar.c and rewrite it to use fork and processes instead of threads.
2. Run the result to demonstrate that the variable x is not shared – the parent changing the variable will not affect its value in the child process.

We can move on to data races now, see race.c.

Part 7.4: race.c (source code)

The header files are the same as before.

```
#include <pthread.h> /* pthread_create, pthread_join */
#include <stdio.h>   /* printf, perror */
#include <unistd.h>  /* sleep, usleep */
```

This program shall demonstrate why the way we have used the global variable previously is a really bad idea. Let's try to just increment and decrement the variable from the main thread and from the child thread at the same time and see what happens.

```
volatile int x = 0;
```

The thread simply tries to increment x a million times.

```
void *thread( void *arg )
{
    for ( int i = 0; i < 1000000; ++i )
        ++ x;
    return NULL;
}

int main()
{
    pthread_t tid;
```

Let's start the thread.

```
    if ( pthread_create( &tid, NULL, thread, NULL ) )
        return perror( "pthread_create" ), 1;
```

Now let's decrement the variable x a million times. Since we increment and decrement the same number of times, we would expect the result to be 0, regardless of the order in which the increments and decrements get executed in the two threads.

```
    for ( int i = 0; i < 1000000; ++i )
        -- x;
```

We wait for the child thread to finish, so we know that both loops have executed the same number of times.

```
    if ( pthread_join( tid, NULL ) )
        return perror( "pthread_join" ), 1;
```

Here, the value of x should be zero. Let's check.

```
    printf( "x = %d\n", x );
```

Oops. Notice that the actual value is also different every time you run the program. That is quite unfortunate! Let's try to fix that.

```
}
```

We will look at mutexes, which solve this (and related) problems. See mutex.c.

Part 7.5: mutex.c (source code)

In this program, we will demonstrate how a mutual exclusion device works. Afterwards, we will combine global shared variables with a mutex, which is one of the correct approaches for communicating in a threaded program.

```
#include <pthread.h> /* pthread_create, pthread_join */
#include <stdio.h>   /* puts, perror */
#include <unistd.h>  /* sleep */
#include <stdlib.h>  /* exit */
```

First of all, we need a variable to represent the mutual exclusion device itself. The variable should be of type pthread_mutex_t and it can be either initialized by a macro like we do here, or by calling the function pthread_mutex_init. The lock and unlock methods of the mutex are implemented by the functions pthread_mutex_lock and pthread_mutex_unlock respectively. We will see an example below.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

The thread will just print messages and sleep in-between to demonstrate how mutual exclusion operates.

```
void *thread( void *arg )
{
    puts( "thread started" );
    sleep( 3 );
```

Now we lock the mutex, entering a critical section. All mutex (and semaphore, barrier and similar) functions take the address of the mutual exclusion device as their parameter. This is because the device (the mutex) must be the same mutex in all threads – passing the mutex by value would make a copy, and could not do what we want (that is, lock and unlock the shared mutex stored in the variable above).

```
    if ( pthread_mutex_lock( &mutex ) )
        perror( "pthread_mutex_lock" ), exit( 1 );
```

Now that we are in the critical section, print a message and sleep for a few seconds, so it is clear that the main thread cannot enter the critical section while we are in it.

```
    puts( "thread in the critical section" );
    sleep( 3 );
    puts( "thread spent 3 seconds in the critical section" );
```

We are satisfied with our stint in the critical section, let's unlock the mutex…

```
    if ( pthread_mutex_unlock( &mutex ) )
        perror( "pthread_mutex_unlock" ), exit( 1 );
```

… print a few more messages …

```
    puts( "thread is out of the critical section" );
    sleep( 3 );
    puts( "thread finished" );
```

… and stop

```
    return NULL;
}

int main()
{
    pthread_t tid;
```

We start the other thread first:

```
    if ( pthread_create( &tid, NULL, thread, NULL ) )
        return perror( "pthread_create" ), 1;
```

Let's try to enter the critical section repeatedly (every second) and print a message whenever we get in.

```
    for ( int i = 0; i < 10; ++i )
    {
```

The call to lock a mutex is the same in all threads.

```
        if ( pthread_mutex_lock( &mutex ) )
            return perror( "pthread_mutex_lock" ), 1;

        puts( "main entered critical section" );
```

Likewise for unlocking it.

```
        if ( pthread_mutex_unlock( &mutex ) )
            return perror( "pthread_mutex_unlock" ), 1;

        sleep( 1 );
    }
```

And wait for the thread to finish.

```
    if ( pthread_join( tid, NULL ) )
        return perror( "pthread_join" ), 1;

    return 0;
}
```

As promised, the next program will show how to use mutexes together with global variables to communicate properly. See mtxvar.c.

Part 7.6: mtxvar.c (source code)

A couple #include statements:

```
#include <pthread.h> /* pthread_{create,join} */
                     /* pthread_mutex_{lock,unlock} */
#include <stdio.h>   /* printf, perror */
```

We no longer need the volatile keyword, because we will use a mutex to guard access to the variable x, which among other things tells the compiler to generate correct code for multiple threads accessing it.

```
int x = 0;
```

Like before, we set up a global, shared mutex.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

The child thread tries to increment x a million times, but this time it protects the increment with the mutex – that is, it turns the increment into a critical section.

```
void *thread( void *arg )
{
    for ( int i = 0; i < 1000000; ++i )
    {
        if ( pthread_mutex_lock( &mutex ) )
            return perror( "pthread_mutex_lock" ), NULL;

        ++ x;

        if ( pthread_mutex_unlock( &mutex ) )
            return perror( "pthread_mutex_unlock" ), NULL;
    }
    return NULL;
}

int main()
{
    pthread_t tid;
```

First of all, let's start the thread.

```
    if ( pthread_create( &tid, NULL, thread, NULL ) )
        return perror( "pthread_create" ), 1;
```

Like in the child thread, the parent should guard the decrement on x with the same mutex (and hence the same critical section).

```
    for ( int i = 0; i < 1000000; ++i )
    {
        if ( pthread_mutex_lock( &mutex ) )
            return perror( "pthread_mutex_lock" ), 1;

        -- x;

        if ( pthread_mutex_unlock( &mutex ) )
            return perror( "pthread_mutex_unlock" ), 1;
    }
```

We wait for the child thread to finish, so we know that both loops have executed the same number of times.

```
    if ( pthread_join( tid, NULL ) )
        return perror( "pthread_join" ), 1;
```

At this point, the value of x should be zero. It wasn't before, and we hope that we fixed the problem. Let's see.

```
    printf( "x = %d\n", x );
```

Yay! The program takes appreciably longer to run, but it works.

```
}
```

Now that we know how to combine global variables and mutexes to communicate, let's try another exercise. Check out 2.txt.

Part 7.7: Exercise 2

1. Write a C program with threads. Set up a global (shared) mutex and a character array (a string) 32 bytes long.
2. The main thread should:
   a. start the child thread (see next point)
   b. sleep for half a second
   c. read the first 31 bytes of intro.txt into the global character array (in a critical section protected by the mutex)
   d. wait for the child thread to finish
3. The child thread should:
   a. print the content of the global character array
   b. sleep for a second
   c. print the content again
4. Both reads from the array in the child thread should be protected by the mutex.
5. Run the program to demonstrate it works as expected.

We can now move on to more complex synchronisation devices: semaphores and barriers. First, however, let's do one more thing with mutexes: busy waiting. See busy.c.

Part 7.8: busy.c (source code)

This program demonstrates how to busy-wait for a condition using a shared global variable and a mutex.

```
#include <pthread.h> /* pthread_create, pthread_join */
#include <stdio.h>   /* puts, perror */
#include <unistd.h>  /* sleep */
#include <stdlib.h>  /* exit */
```

Set up a global variable and a mutex to protect it.

```
int proceed = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Thread announces that it started, then waits for proceed to become non-zero and then finishes.

```
void *thread( void *arg )
{
    puts( "thread started" );
    while ( 1 )
    {
```

We enter the critical section:

```
        if ( pthread_mutex_lock( &mutex ) )
            perror( "pthread_mutex_lock" ), exit( 1 );

        if ( proceed )
        {
            puts( "thread allowed to finish" );
```

We must always unlock the mutex that we locked, otherwise we could cause deadlocks. This is very important!

```
            if ( pthread_mutex_unlock( &mutex ) )
                perror( "pthread_mutex_unlock" ), exit( 1 );

            return NULL;
        }
```

And leave the critical section again:

```
        if ( pthread_mutex_unlock( &mutex ) )
            perror( "pthread_mutex_unlock" ), exit( 1 );

    }
```

```
}
```

The main function just sets up the thread, waits for a while and then lets it proceed.

```
int main()
{
    pthread_t tid;
```

Start the thread:

```
    if ( pthread_create( &tid, NULL, thread, NULL ) )
        return perror( "pthread_create" ), 1;

    puts( "main thread sleeping for 3 seconds" );
    sleep( 3 );

    if ( pthread_mutex_lock( &mutex ) ) /* critical section */
        return perror( "pthread_mutex_lock" ), 1;

    puts( "main thread setting proceed = 1" );
    proceed = 1;

    if ( pthread_mutex_unlock( &mutex ) ) /* critical section end */
        return perror( "pthread_mutex_unlock" ), 1;
```

And wait for the thread to finish.

```
    if ( pthread_join( tid, NULL ) )
        return perror( "pthread_join" ), 1;

    puts( "all done" );
    return 0;
}
```

We can now go on to semaphore.c.

Part 7.9: semaphore.c (source code)

In this program, we will look at a semaphore, which is a generalization of a mutex. The structure of the program is the same as the one that demonstrated mutexes: mutex.c.

```
#include <pthread.h>   /* pthread_create, pthread_join */
#include <stdio.h>     /* printf, perror */
#include <unistd.h>    /* sleep */
#include <stdlib.h>    /* exit */
#include <semaphore.h> /* sem_init, sem_wait, sem_post */
```

The type for a semaphore is sem_t. Unlike pthread mutexes, semaphores can be used in multi-process programs too, but this also requires allocation of inter-process shared memory. You could try doing this as an optional exercise (you will need shm_open and mmap). In this program, we will use a semaphore with threads in a single process.

```
sem_t semaphore;
```

Unlike earlier programs, this one will run multiple threads (i.e. multiple child threads, instead of just one). All the child threads will, however, execute the same function. We will make use of the argument to tell those threads apart.

```
void *thread( void *arg )
{
    char *thread_name = arg;

    sleep( 3 );
```

Waiting for a semaphore enters the critical section. The number of threads that can simultaneously enter the critical section is configured by the parameters to sem_init, which is called by the main thread below.

```
    if ( sem_wait( &semaphore ) )
        perror( "sem_wait" ), exit( 1 );
```

Now that we are in the critical section, print a message and sleep for a few seconds, so it is clear that the main thread cannot enter the critical section while we are in it.

```
    printf( "%s in the critical section\n", thread_name );
    sleep( 3 );
```

The equivalent to unlocking a mutex is posting the semaphore, and the function to do so is called sem_post.

```
    if ( sem_post( &semaphore ) )
        perror( "sem_post" ), exit( 1 );

    printf( "%s done\n", thread_name );
    return NULL;
}

int main()
{
    pthread_t tid1, tid2, tid3;
```

Semaphores must be initialized by a call to sem_init. The function takes 3 parameters: the address of the semaphore object, whether this is an inter-process semaphore (not in our case, so we pass 0) and the number of threads that can get into the critical section at once (2, in our case).

```
    sem_init( &semaphore, 0, 2 );

    if ( pthread_create( &tid1, NULL, thread, "thread 1" ) ||
         pthread_create( &tid2, NULL, thread, "thread 2" ) ||
         pthread_create( &tid3, NULL, thread, "thread 3" ) )
        return perror( "pthread_create" ), 1;
```

Let's try to enter the critical section repeatedly (every second) and print a message whenever we get in.

```
    for ( int i = 0; i < 10; ++i )
    {
        if ( sem_wait( &semaphore ) )
            return perror( "sem_wait" ), 1;

        puts( "main entered critical section" );

        if ( sem_post( &semaphore ) )
            return perror( "sem_post" ), 1;

        sleep( 1 );
    }
```

And wait for the threads to finish.

```
    if ( pthread_join( tid1, NULL ) ||
         pthread_join( tid2, NULL ) ||
         pthread_join( tid3, NULL ) )
        return perror( "pthread_join" ), 1;

    return 0;
}
```

I think it is quite clear how a semaphore works; let's look at a barrier now, in barrier.c.

Part 7.10: barrier.c (source code)

Barriers are dual to semaphores, in the sense that semaphores allow through at most a given number of threads. Barriers let through a given minimum number of threads instead – a sufficient number of them must reach the barrier for any to be able to continue.

```
#include <pthread.h>   /* pthread_{create,join} */
                       /* pthread_barrier_{init,wait} */
#include <stdio.h>     /* printf, perror */
#include <unistd.h>    /* sleep */
#include <stdlib.h>    /* exit */
```

Like with mutexes and semaphores, there is a dedicated type for barriers, called `pthread_barrier_t`. Like with a semaphore, it must be initialized by a call to a function, `pthread_barrier_init`.

```
pthread_barrier_t barrier;
```

Like with semaphores, we will run multiple threads. They will all wait a certain number of seconds (given to them in their argument) and then proceed to synchronise on the barrier.

```
void *thread( void *arg )
{
    int n = (int) arg;

    printf( "thread %d sleeping for %d seconds\n", n, n );
    sleep( n );
```

The call to `pthread_barrier_wait` is quite simple, and unlike with mutexes and semaphores, is not paired. We don't check for errors, since the only error that could happen is that we pass an invalid barrier. Additionally, checking for the return code is more complicated, because the return value of `pthread_barrier_wait` is different in different threads. See the man page for details.

```
    pthread_barrier_wait( &barrier );
```

All threads should make it to this point at once.

```
    printf( "thread %d got past the barrier\n", n );
    return NULL;
}

int main()
{
    pthread_t tid1, tid2, tid3;
```

Barriers, like semaphores, are initialized by a call to an initialization function, in this case `pthread_barrier_init`. The arguments are: the barrier to initialize, an optional attribute object (which we do not care about) and the number of threads to synchronize on the barrier.

```
    pthread_barrier_init( &barrier, NULL, 4 );

    if ( pthread_create( &tid1, NULL, thread, (void *) 1 ) ||
         pthread_create( &tid2, NULL, thread, (void *) 2 ) ||
         pthread_create( &tid3, NULL, thread, (void *) 3 ) )
        return perror( "pthread_create" ), 1;

    printf( "all threads running\n" );
    printf( "main thread sleeping for 10 seconds\n" );

    sleep( 10 );
    pthread_barrier_wait( &barrier ); /* we also wait */
```

And wait for the threads to finish.

```
    if ( pthread_join( tid1, NULL ) ||
         pthread_join( tid2, NULL ) ||
         pthread_join( tid3, NULL ) )
        return perror( "pthread_join" ), 1;

    printf( "everything done\n" );

    return 0;
}
```

Again, it should be quite clear how barriers work. Let's try an exercise:

```
$ cat 3.txt
```

Part 7.11: Exercise 3

1. The barrier provided by the operating system is quite smart: threads that wait are sleeping, not busy-waiting, making the entire program more efficient.
2. But in principle, a barrier is very simple. Let's try to make one ourselves.
3. Write a C program which only uses mutexes and shared variables to implement a barrier.
4. The waiting threads can busy-wait.

That's the last exercise for today, the homework is, as usual, in `hw.txt`.

Part 7.12: Homework

1. In the seminar, we implemented a simple busy-waiting barrier.
2. Let's do that again, but this time with a semaphore.
3. You should create two files, `ssem.c` and `main.c`.
4. In `ssem.c` (for spin semaphore), implement the following functions:
   a. `ssem_init( int c )` which initializes the semaphore, `c` being the maximal number of threads to let in
   b. `ssem_wait()` which waits for the semaphore
   c. `ssem_post()` which unlocks the semaphore
   d. there should not be a `main` function in `ssem.c`
5. The global variables that you need should also live in `ssem.c`. Note that there is only a single global (spin) semaphore, not a data type and methods for this type. This is unlike `sem_t` from the standard library.
6. In `main.c`, write a simple program that checks that semaphore works:
   a. call `ssem_init()` so that 2 threads can enter at once
   b. create 2 new (child) threads
   c. all threads (including the main one) try to enter a critical section
   d. the critical section is guarded by `ssem_wait`
   e. each thread, upon getting into the critical section calls `in_critical()`
   f. the function `in_critical()` simply prints a message to `stdout`

Part 8: Networking

1. Today, we will learn a bit about networking.
2. In the first half, we will look at networking from the user perspective.
3. That includes a few diagnostic tools (`ping`, `traceroute`), general-purpose utilities (`nc` a.k.a. netcat and `telnet`).
4. But also clients for well-established protocols, like HTTP (`curl` and `wget`), SSH and DNS (`dig`).
5. In the second half, we will look at network (`AF_INET`) sockets.

Let's start with `ping` and `traceroute`: `cat ping.sh`.

Part 8.1: `ping.sh` (source code)

There is a number of diagnostic utilities for IP networks, mostly based on the ICMP protocol. The most basic of those is `ping`, which asks another computer whether it's alive. The packets it sends are ICMP echo request, and the target host is supposed to send an ICMP echo

reply back.
Given no switches, ping will send a packet every second or so, forever. But we don't have that much time, so we request it only sends 5 and then stops, like this:

```
ping -c 5 1.1.1.1
```

The `1.1.1.1` above is an IP address. This one is particularly easy to remember – it's a recursive DNS server operated by cloudflare. Notice the numbers that ping prints: those are statistics about how long it took for the packet to get there and back (rtt = round-trip time). Notice that it's only a few milliseconds – that means the server is not very far from us. Magic is involved (and IP addresses that belong to different computers depending on where you are)
Anyway, to prevent Google getting jealous, let's also ping their recursive DNS server, which lives at the address 8.8.8.8 (not as flashy as the cloudflare one, but not bad either):

```
ping -c 5 8.8.8.8
```

Let's just round it off with my favourite public recursive resolver, ran by Level3, living at the address 4.2.2.2 (I don't actually recommend using it for DNS... I just like to ping it):

```
ping -c 5 4.2.2.2
```

That's it for ping (with IP addresses), let's look at `traceroute` now, which is a little more interesting. The script lives in `traceroute.sh`.

Part 8.2: `traceroute.sh` (source code)

Like `ping`, `traceroute` sends out ICMP echo request packets. However, unlike `ping`, it also sets the TTL field in the IP header – short for Time To Live. This number is decremented by each router along the way, and the packet is thrown away when its TTL drops to 0. The router doing the dropping also helpfully sends back an ICMP 'time to live exceeded' packet.
Someone clever noticed that you can send ICMP echo request with TTL set to 1 to learn the address of the first router along the way to the destination, then set TTL to 2 to learn about the second closest one, and so on. This is what `traceroute` does, and it prints information about each of the hosts it encounters. Let's try it on the trio of computers we pinged earlier:
By default, `traceroute` tries to find a hostname for each IP address. But we don't know enough about DNS to worry about that yet. So we just ask it to print numeric addresses, using `-n`.

```
traceroute -n 1.1.1.1
traceroute -n 8.8.8.8
traceroute -n 4.2.2.2
```

Now that we tried both `ping` and `traceroute`, let's try some DNS queries. See `host.sh`.

Part 8.3: `host.sh` (source code)

In this script, we will first learn about the `host` utility, which does basic DNS resolution for us. For instance, we can ask what is the IP address of `aisa` like this:

```
host aisa.fi.muni.cz.
```

We can also use the shorthand, since we are in the same domain:

```
host aisa
```

This has some interesting consequences:

```
host www
```

What do you think the last query means?

Besides translating names to addresses, DNS can also go the other way: give a hostname for a numeric address. We can simply pass the IP address to the `host` utility, like this:

```
host 4.2.2.2
```

This uses some magic behind the curtain (though the magic also leaks into the user interface some). There is a special domain called in-addr.arpa, which has subdomains for all the possible IP addresses – in reverse. So when we ask `host` to find the name for `4.2.2.2`, it is in fact looking for `2.2.2.4.in-addr.arpa.` There's magic involved because the normal recursion rules don't quite work in this domain. Nonetheless, there are DNS records (of type `PTR`) and they contain the 'canonic' hostname of the computer that lives at the given IP address. You usually want the name to resolve back to the same IP address.
In fact, let's refresh our shell scripting skills with an exercise (in `1.txt`). After that, we'll look at `dig` which is another DNS tool and then we'll move on to `nc`.

Part 8.4: Exercise 1

1. Write a shell script that does the following:
   a. takes one argument, an IP address (see also below)
   b. resolves the address to a hostname
   c. resolves the hostname back to an (IPv4) address
   d. prints 'yes' if they match
   e. prints 'no' if they are mismatched
   f. prints 'error' if an error happens
2. You can run the script as `sh chkrdns.sh 4.2.2.2`.
3. The address will be available in the variable `$1`.
4. Anything other than 'yes', 'no' and 'error' should go to `stderr` (but you can print what you like there).
5. To redirect the `stdout` to `stderr` for a command, you can use `1>&2` after the command, like so: echo hello 1>&2
6. Hints: this should be quite doable with just `cut` and `$(command)`.

Let's continue to `dig.sh`.

Part 8.5: `dig.sh` (source code)

Dig is a somewhat more flexible tool for doing DNS queries from the command line. The output is also more DNS-like so to understand it, you need to know a little about DNS. For our use-case, it should be sufficient to know that there are different types of records in DNS (carrying different types of info about hostnames): `NS` stands for name server (i.e. a DNS server responsible for a given subdomain), `A` stands for IPv4 address, `AAAA` stands for IPv6 and `MX` stands for mail exchanger.
Let's try some basic queries:

```
dig aisa.fi.muni.cz -t A
```

The above performs a DNS query using the default system DNS recursive server. It does not perform recursion itself. The output is fairly long, so let's look at what's in there. Let's ignore the header, all the way until QUESTION SECTION. The question section shows us what the question we asked was. In this case, we would like the `A` record of `aisa.fi.muni.cz` filled in for us.
What follows is an ANSWER SECTION, which contains the answer to our query: it tells us that the `A` record for `aisa.fi.muni.cz` contains the address `147.251.48.1`. What follows after that is the list of name servers (`NS` records) responsible for the domain `fi.muni.cz.` and finally we get some `A` and `AAAA` records for those name servers too. In case we wanted to directly consult them.
We will actually stop the script here, so you can dig (ha-ha) through all the output ; when you are ready to continue, just hit enter.

```
read
```

Since output from dig is so long and tedious, let's only do one more. This replicates what I have shown you in the lecture (but that was somewhat redacted, to fit in the slide). Let's also ask for a different type of record for the fun of it. The +trace tells dig to perform recursion itself and print all the intermediate results too.

```
dig fi.muni.cz -t MX +trace
```

The MX record tells us who handles mail for the given domain. So when you send an email to me (using my @fi.muni.cz address), the MTA that sends email for you will do an MX lookup for fi.muni.cz to find out which computer it should contact. In this case, it'll see:

```
fi.muni.cz. 300 IN MX 50 relay.muni.cz.
```

Which basically means there's only one mail exchanger, and its DNS name is relay.muni.cz. Let's continue in this spirit with netcat.sh.

Part 8.6: netcat.sh (source code)

We will look at the client side of nc (netcat) in this script. We left off with dig when we learned that email for fi.muni.cz is handled by relay.muni.cz. Let's look at what would happen next in our hypothetical mail scenario.
The MTA would learn the destination IP address and initiate an SMTP session. I won't get into much detail, but let's talk about nc for a bit. What nc does is basically what cat does, but then sends the data over a network. It can both listen to connections (with -l) or connect to existing services. We will now use the latter mode.
In either case, whatever you send to nc, it will forward into the network, and what falls out of the network connection, nc will print to its stdout. Let's try with some SMTP:

```
{
    sleep 5; # wait a bit for the remote server to greet us
    echo HELO aisa # tell them who we are (hostname)
    echo "MAIL FROM:<not@me>" # and our mail address
    echo "RCPT TO:<yes@me>"   # and the recipient of the email
```

At this point, the SMTP server will stop talking to us because it does not want to send mail to yes@me – that is fine, we didn't want to actually spam anyone. The rest of the SMTP session would mostly involve the actual message.

```
} | nc relay.muni.cz 25 # 25 is the SMTP port
```

Let me just note that nc is not quite standardized. It's present on many/most Unix systems, but the exact behaviour and switches might differ. Be careful and read man pages. Let's try looking at nc in server mode, in listen.sh.

Part 8.7: listen.sh (source code)

```
set -x # ask the shell to print out what is going on
```

This is a simple script that demonstrates nc in listening mode. To wit:

```
port=2222
nc -l $port > file.txt &
```

Now unlike most of the things we did earlier, you may not be able to run this script. That's because port numbers are a global, shared resource, and if one person is listening on port 2222 on aisa, this port is no longer available for others. You may want to try with a different port number (must be greater than 1024, too). But since the script is brief, you might get lucky.
Now let's send something to the listening nc using another nc, this time in client mode:

```
echo "hello world on port $port" | nc localhost $port
```

localhost is a special name, reserved to mean 'this computer'. But we could have used aisa.fi.muni.cz just fine, too. Anyway, let's check the content of file.txt:

```
wait # wait for the backgrounded <nc -l> to finish
cat file.txt # print the content of the file
```

Let's do another exercise: 2.txt.

Part 8.8: Exercise 2

1. Write a shell script that:
   a. listens, using netcat, on port 2222
   b. someone (e.g. you from another terminal) will send a message to that port, in this form: host1 host2 port
   c. fetch, using http the / document from host1 (see also below)
   d. send the result to host2 on port port
2. You can use a temporary file and cut to extract the information.
3. To fetch a document using HTTP, you simply send GET / on port 80, like this: echo GET / | nc www.fi.muni.cz 80.

This is almost all for the user part. We will have a brief look at ssh and curl before we delve into C again.
Continue to:

```
$ micro ssh.sh
```

Part 8.9: ssh.sh (source code)

This is another simple script which shows that ssh can also be used for moving data across the network, like nc, but much more securely. Of course, there is some overhead involved, but in 98% of the cases, ssh is a much better choice.

```
set -x
```

First notice that ssh works non-interactively if we give it a command to execute, like this:

```
ssh aisa echo hello world
```

Of course we can pipe the output to a (local) command:

```
ssh aisa echo hello world | cut -f1 -d' '
```

We can also pipe stuff to the stdin of ssh and it appears as the stdin of the remote command, like the following. Unfortunately, we have to do double escaping, because the command is given to a shell at the remote end, but the quotes were already stripped by the local shell (when constructing argv for ssh).

```
echo hello world | ssh aisa "cut -f2 -d' '"
```

That's all for ssh. Let's move on to curl and then finally some C:

```
$ micro curl.sh
```

Part 8.10: curl.sh (source code)

While we can, technically, use nc to talk to HTTP servers, it's not the most convenient thing to do. We will look at another utility, called curl (the project name is cURL, URL being short for 'uniform resource locator'), which can, among other things, fetch data from HTTP servers. Unlike nc, it will construct headers, parse replies and do a whole bunch of other useful things, like SSL or POST request with relative ease.
Please note that curl is not standard (but can be installed on most Unix systems).

We can start by a simple curl invocation to fetch a document (the -s option tells curl to not print progress information to stderr):

```
echo fetch http://ix.io/1FWS
curl -s http://ix.io/1FWS
echo
```

Out of the box, curl prints the document to stdout – this is convenient if we want to process the output immediately using some shell magic. But if we prefer to have it stored in a file, we can use curl -O, which will select a filename for us:

```
echo fetch https://www.fi.muni.cz/index.html
test -f index.html && exit 1 # die if the file is already here
curl -s -O https://www.fi.muni.cz/index.html
test -f index.html || exit 1 # die if the file did not materialize
ls -hl index.html
rm index.html
echo
```

The last option to curl (related to GET requests) to look at is -L, for 'location'. In some instances, the HTTP server may return a code like 301 which tells the client that they should be looking elsewhere. For instance:

```
echo fetch http://www.fi.muni.cz
curl -s http://www.fi.muni.cz
echo
```

In this case, we get a 301 Moved Permanently, with the instruction to look at the encrypted version of the site (i.e. https://www.fi.muni.cz). Since we don't want to handle such matters ourselves, we can use curl -L which will automatically follow the redirect:

```
echo "fetch http://www.fi.muni.cz (with redirects)"
curl -s -L http://www.fi.muni.cz | wc -c
```

That's for GET requests with cURL, let's go on to curl-post.sh.

Part 8.11: curl-post.sh (source code)

This is a very simple script, which demonstrates how curl can be used to send data to a remote server, using so-called POST requests. This script can be really useful in case you want to send the output of a command or the content of a plain text file to someone across the internet.
We use a web service called ix.io which will hold text for us, and lets us send it using a simple POST request. Basically a no-frills pastebin.

```
echo hello world | curl -F 'f:1=<-' http://ix.io
```

The -F argument to curl is short for 'form', and uses the POST protocol in a manner similar to how a web browser would use it to send data you filled in an HTML form. f:1 is the name of the key and the <- is an instruction to read the value for that key from stdin.
After the script runs, you can try fetching (with just curl) the resulting URL. You should get back hello world.
That's all for the user-level part, let's do some C:

```
$ micro ghbn.c
```

Part 8.12: ghbn.c (source code)

This program demonstrates the use of the gethostbyname C function. It works a little like the host program: it gets a hostname as the first argument and prints the address it obtains from gethostbyname on standard output.

```
#include <netdb.h> /* gethostbyname, hstrerror, h_errno */
#include <stdio.h> /* fputs, fprintf */
```

```
int main( int argc, const char **argv )
{
    if ( argc != 2 ) /* check the number of arguments */
        return fputs( "expected 1 hostname\n", stderr ), 1;
```

The gethostbyname function is almost the simplest interface to the resolver that you could imagine. A string goes in and you get back a pointer to a structure with the address(es). If an error occurs, the pointer will be NULL.

```
    struct hostent *he = gethostbyname( argv[1] );

    if ( !he ) /* deal with a possible error */
    {
```

Unlike most C functions that we encountered so far, gethostbyname does not set errno. Instead, it has a similar variable called h_errno and a function hstrerror to turn the numeric value of of h_errno into a human-readable string. You can try calling this program on a non-existent domain to see how this works.

```
        fprintf( stderr, "error resolving host %s: %s\n",
                 argv[1], hstrerror( h_errno ) );
        return 2;
    }
```

If we got here, hostname resolution worked and we have an address in he. Thing is, one host can have more than a single IP address, so what gethostbyname gives you is a list that works a bit like argv. But we only care about a single address now, so let's just print it. See also man gethostbyname.

```
    unsigned char *addr = he->h_addr_list[0];
    printf( "host %s has address %d.%d.%d.%d\n",
            argv[1], addr[0], addr[1], addr[2], addr[3] );
    return 0;
}
```

We can now go on to http-get.c.

Part 8.13: http-get.c (source code)

This program is the C equivalent of echo GET / | nc localhost 80.

```
#include <unistd.h>     /* read, write */
#include <stdio.h>      /* perror */
#include <string.h>     /* strlen */
#include <sys/socket.h> /* socket, connect */
#include <netinet/in.h> /* sockaddr_in */
```

Before we begin, we declare some global constants that we will use as the connection parameters later on: the IP address of the remote system and the port to connect to. The port looks 'normal', but the IP address is weird. This is how I got the number (think about what it does for yourself):

```
$ printf "%02x%02x%02x%02x" 147 251 48 1
```

```
const uint32_t REMOTE_IP   = 0x93fb3001;
const uint16_t REMOTE_PORT = 80;

int main( int argc, const char **argv )
{
```

We first create a socket. The address family is INET (for IPv4 sockets) and the type is STREAM, for pipe-like operation. This pretty much implies TCP. As usual, we left the protocol to be filled in by the OS.

```
    int sock_fd = socket( AF_INET, SOCK_STREAM, 0 );

    if ( sock_fd == -1 ) /* something went wrong, bail */
        return perror( "socket" ), 2;
```

We now set up the address. The structure is slightly more complicated than the sockaddr_un that we used two weeks ago. There is a further bit of complication stemming from the fact that the networking API uses so-called network byte order – also known as big endian – to represent numbers. But the CPU in aisa (and in most contemporary computers) represent numbers the other way around, as little endian. There are a few helpful functions to convert between those formats.
We will use two: htons and htonl which are short for Host TO Network Short and Host TO Network Long and convert 16 and 32 bit numbers respectively. The opposite direction is done by ntohs and ntohl. You must always remember to use those functions when filling in sockaddr_in with native integers, otherwise bad things will happen.

```
    struct sockaddr_in sa = { .sin_family = AF_INET,
                              .sin_addr = { htonl( REMOTE_IP ) },
                              .sin_port = htons( REMOTE_PORT ) };
```

From this point, there's nothing really new. Let's connect the socket to the remote host/port.

```
    if ( connect( sock_fd, ( struct sockaddr * ) &sa,
                  sizeof( sa ) ) == -1 )
        return perror( "connect to 127.0.0.1" ), 2;
```

Send the HTTP request.

```
    const char *request = "GET / HTTP/1.0\r\n\r\n";
    int reqlen = strlen( request );
    if ( write( sock_fd, request, reqlen ) != reqlen )
        return perror( "write" ), 2;
```

Okay, I lied a little. This part is somewhat new, though we know all the ingredients. What it does is read everything that the remote side sent and print it out to stdout, in chunks of at most 32 bytes.

```
    char buf[ 32 ];
    int bytes;

    while ( ( bytes = read( sock_fd, buf, 32 ) ) > 0 )
        write( 1, buf, bytes );
```

```
    close( sock_fd );
    return 0;
}
```

That's all of the demos for today, let's do another exercise and then you can get started on the homework. The exercise is in 3.txt.

Part 8.14: Exercise 3

1. Combine gethostbyname with the socket client code to fetch a document given on the command line.
2. The host to contact is the first argument.
3. The filename part of the URL to fetch is the second argument.
4. Print the reply to stdout.
5. E.g. ./a.out ix.io /1FWS will send the following data:

```
    GET /1FWS HTTP/1.1\r\nHost: ix.io\r\n\r\n
        ^^^^^ the url            ^^^^^ hostname
```

The homework is, as usual, in hw.txt.

Part 8.15: Homework

1. Write a program, echo.c, which:
   a. connects to the TCP port 2222 of localhost
   b. reads a message of up to 20 bytes from the connected socket
   c. sends back the string "you said <message>" over the socket
   d. closes the connection and exits with status 0
2. The IP address of localhost is 127.0.0.1
3. Write a shell script echo.sh, which:
   a. compiles the program from point 1 into a.out
   b. listens on port 2222 using nc
   c. starts a.out from point 3.a
   d. sends message ping to a.out from 3.c, using the listening nc instance started in 3.b
   e. prints the message from a.out onto stdout
   f. waits until nc from point 3.b terminates

Part 9: Scripting Redux

1. Today will be slightly different than usually. We will only do exercises, since you already know most of the ingredients.
2. You can refer to examples from previous seminars, lecture slides and of course man pages.
3. In case a program you don't know could be useful, I will give you a name (so you can look it up), or a very short example.
4. If you remember a keyword, a program name or any other fragment of information from previous seminars, but aren't sure where it was, a good command to try is grep -r keyword ~/pb152.
5. There are 10 exercises today, plus the homework, in hw.txt.
6. Use NN.sh as the file name for submitting exercises, e.g. 10.sh for the solution of 10.txt. Use NN.c for C programs and NN.mk for makefiles.

That said, let's start with 01.txt.

Part 9.1: Exercise 1

Let's refresh our memory of shell scripts and pipes. Write a script, that:

1. uses ~/pb152/01/oslist.csv as the input,
2. prints the number of operating systems by Microsoft,
3. prints the year and name of all operating systems by Apple,
4. prints the list of all mono- and microkernel operating systems.

That should have been fairly easy. Let's move on to 02.txt.

Part 9.2: Exercise 2

Let's look at compiling programs. And also writing some very simple C:

1. write a C program that:
   a. prints its first argument (if present),
   b. prints no arguments if it got no arguments,
2. write a shell script that:
   a. compiles the program in steps (preprocess, compile, assemble, link),
   b. the binary name should be a.out,
   c. runs the program with all shell scripts in ~/pb152/08 as arguments,
   d. checks that ./a.out hello world prints hello,
   e. if it does, the script prints it works,
3. modify the program to print its second argument instead and rerun the script from point 3 (submit this version).

Part 9.3: Exercise 3

1. Write a script that:
   a. creates an empty file called file.txt,

   b. creates a symlink to this file, called `symlink.txt`,
   c. creates a hardlink to `file.txt` called `link.txt`,
   d. writes `foo` into symlink.txt,
   e. prints the target of `symlink.txt`.
2. Write a script that:
   a. checks if the target of `alink.txt` exists,
   b. if it does, it prints the target,
   c. otherwise, it prints `alink.txt is broken`.
3. Create a valid and a broken `alink.txt` and run the script from point 2 to check that it works.

Part 9.4: Exercise 4

1. Write a C program that prints the i-node number of a file it gets as its first argument.
2. Write a makefile that compiles this program.

Part 9.5: Exercise 5

1. Write a shell script that prints the vendor, year and name (in this order) of each operating system from `oslist.csv`, sorted by vendor first and name second.
2. Hint: try the following command:

```
$ echo 'hello,world' | sed -re 's:([^,]*),([^,]*):\2,\1:'
```

Part 9.6: Exercise 6

1. Write a script that renames each file in the current directory to only use minuscule letters. E.g. `IMG_32.jpg` would be renamed to `img_32.jpg`.
2. Create some test files and run the script to check it works.
3. Hint: try the following command:

```
$ echo Hello W0rLD | tr A-Z a-z
```

Part 9.7: Exercise 7

1. Write a script that, for each file in the current directory, moves and/or renames files like this:
   - `example_file.txt` becomes `example/file.txt`,
   - `bar.txt` stays unchanged,
   - `some_longer_name.sh` ends up as `some/longer/name.sh`.
2. Create any directories that are missing.
3. Try the following commands:

```
$ echo some_longer_name.sh | sed -e s,_,/,g
$ dirname some/longer/name.sh
```

Part 9.8: Exercise 8

Write a script that removes all broken symlinks from the current directory. For each symlink, print whether it is OK or broken (and hence is being erased).

Part 9.9: Exercise 9

Write a script that prints files in the current directory, sorted by their suffix. Print directories first (sorted by name).

Part 9.10: Exercise 10

1. Write a script that builds a program from multiple sources.
2. It takes a list of files as arguments (i.e. any number of arguments, one file name per argument).
3. If the file is a `.c` or `.s`, compile it to an object.
4. If the file is a `.o`, just keep it.
5. Link all the `.o` files (including those compiled from sources) into a single executable (named `a.out`).
6. If an argument does not exist or is not a `.c`, `.s` or a `.o` file, print an error and abort execution.
7. Make the script print the actions it takes, test it.

Part 9.11: Homework

1. Write a script, `hw.sh`, that prints names of files in the current directory.
2. The output should be sorted by i-node number in ascending order.
3. Use the program from exercise 4 to obtain i-node numbers.
4. Assume the program can be executed as `./04`.

You can also submit the following exercises to earn an activity point for this week: `02`, `04`, `06` and `10`.

Part 10: Access Control

1. In this seminar, we will look at some of the access control concepts.
2. Unfortunately, most of the interesting stuff is off limits for us, since we are mere users (and not `root`).
3. This means we can't create users or groups, or change ownership of files.
4. We can look at some of the APIs and some of the basic commands though.

Let's start with something simple: see `perm.sh`.

Part 10.1: `perm.sh` (source code)

Let's first look at how standard UNIX permissions are encoded in the user interface. We will look at the underlying numeric representation later on.

```
set -x
touch file.txt
```

We have created an empty file, so that we can look at the permissions it has. The encoding might seem odd at first, so let's explain how it works.

```
ls -l file.txt
```

The first column of the output from `ls -l` shows the permissions, and it reads `-rw-r--r--`. This is the 'human readable' encoding of UNIX permission bits. The first dash is for file type, dash meaning a regular file (`d` would mean a directory, `l` a symlink, and so on). However, what follows is more interesting for us now.
The remaining 9 letters form 3 groups of 3 letters each, like this: `rw-`, `r--` and `r--` again. Each of the letter groups encodes access rights of a particular set of users: the first belongs to the owner of the file (in this case, ourselves), the second belongs to the owning group and the last triplet applies to all users that do not belong to the owning group of the file.
The `r` indicates reading, while `w` indicates writing: the owner (us) can read and write the file, while both remaining sets of users can only read the file. The third common letter is `x` and denotes the ability

to execute the file (as a program). The letters have fixed positions: r always comes in the first position, w in the second and x in the third. A dash (-) in the corresponding position means this particular right is denied. We can check:

```
ls -l /bin/echo
```

The owner of /bin/echo is root and the owning group is also root (those are the third and fourth columns of the output above). The permission bits are rwx (the owner can read, write and execute the file), r-x (the root group can read and execute, but not write) and again r-x (in this case, this applies to us). Specifically, we are not allowed to write into /bin/echo:

```
echo hello > /bin/echo # fails with a permission error
```

However, we could read and execute it. Let's try the latter:

```
/bin/echo hello # this works okay
```

Finally, let's double-check that file.txt cannot be executed (it lacks any x bits):

```
./file.txt
```

Let's try changing some permission bits now (in chmod.sh).

Part 10.2: chmod.sh (source code)

The chmod command serves to change permission bits of files (well, i-nodes, so that covers directories, specials and so on). Let's start by creating an empty file:

```
echo creating file.txt...
touch file.txt
echo hello > file.txt # this also works
cat file.txt # this works as expected
```

We have checked that we can read the file and write into it – this is normal and expected. However, we can change the permissions (using chmod) to remove either of those access rights (or both). First, we check what the initial permissions were:

```
ls -l file.txt
```

Let's look at chmod then. What it does is manipulate the individual bits, but using a somewhat different syntax. The formatting we see in the output of ls is optimized for output, but writing out the permissions in this form could be somewhat tedious, especially if all we want is to flip one of the bits. For this reason, chmod gives names to each of the triplet: u for user (owner), g for owning group and o for others. Beware that o does not mean the owner!
What we can do now is tell chmod which bits we are interested in and what to do with them. We can either set bits with + or remove them with -. For instance, g+w would mean 'add the w bit to the group triplet', or in other words, let the users in the owning group write into this file. To remove the write permission from ourselves (the owner), we will use u-w, like this:

```
echo
echo removing the write permission and testing...
chmod u-w file.txt
ls -l file.txt
echo world >> file.txt
```

Likewise, we can remove the read permission from ourselves, using u-r. Then we won't be able to read from the file either:

```
echo
echo removing the read permission and testing...
chmod u-r file.txt
ls -l file.txt
cat file.txt # permission error
```

Multiple bits can be specified in a single group. Let's restore the original permissions:

```
echo
echo "restoring permissions (and testing again)"
chmod u+rw file.txt
ls -l file.txt
echo world >> file.txt
cat file.txt
```

If you want to specify multiple groups in a single chmod command, that can be done too: separate the groups with a comma, like this: g+r,o-w. With that out of the way, let's try an exercise (in 1.txt).

Part 10.3: Exercise 1

1. Write a script that creates files with the following permissions (using touch and chmod):
   a. r-xr-xr-x file1.txt
   b. rwxr--r-x file2.txt
   c. ---r--r-x file3.txt
   d. rw-r-xr-- file4.txt
   e. rwxr-xr-- file5.txt
   f. rwxr-xr-x file6.txt
2. Which of the above permission sets make sense? What would you say is the property that sensible permission sets have?

Let's look at encoding permissions in octal next, in octal.sh.

Part 10.4: octal.sh (source code)

In previous, we have learned how permission bits are represented in 2 related but slightly different 'human readable' formats. At the machine level, though, those bits are simply a 9-bit number (or rather, 9 bits out of a 16 bit number). Since the 9 bits are grouped into 3 groups of 3 bits each, it is convenient to represent them in octal, where each digit corresponds to 3 bits.
Let's first look at the bits that represent various human-readable triplets:

- rwx → 111
- rw- → 110
- r-x → 101
- -wx → 011

... and so on ...
The octal digits correspond to 3-bit binary numbers as follows:

- 0 → 000
- 1 → 001
- 2 → 010
- 3 → 011
- 4 → 100
- 5 → 101
- 6 → 110
- 7 → 111

The digits are written in the same order as with the human readable form: owner, group and others, from the left. Hence:

```
touch file.txt

echo -n "755: "
chmod 755 file.txt
ls -l file.txt

echo -n "750: "
chmod 750 file.txt
```

```
ls -l file.txt

echo -n "644: "
chmod 644 file.txt
ls -l file.txt
```

Let's have a quick look at how directory permissions work now: dir.sh.

Part 10.5: dir.sh (source code)

Let's first create a directory and look at its default permissions.

```
mkdir dir
touch dir/file.txt
ls -ld dir
echo

set -x # print what is going on
```

Then remove all the 'execute' bits:

```
chmod 644 dir
cd dir # permission error
ls dir # ok
echo
```

Let's now try with only the execute bits:

```
chmod 111 dir
cd dir # ok
ls # not ok
cd ..
echo
```

How about r-x------? i.e. deny writing to ourselves

```
chmod 500 dir
rm dir/file.txt # not ok
echo
```

Go back to where we started and clean up.

```
chmod 755 dir
rm dir/file.txt
rmdir dir
```

Next up: umask.sh.

Part 10.6: umask.sh (source code)

Unfortunately, creating files and then changing their permissions is a race condition: some other user could come at the wrong moment, and open the file while it had incorrect permissions (between creation and chmod). In most cases, it is not a huge risk, but it is a risk and in certain circumstances can lead to serious security problems.
There are multiple mechanisms that can be used to close this loophole, one of them being umask. The other is specifying the permissions at the time the file is being created, but this is mainly an option for C programs. On the user side of things, not many tools allow the user to specify the permissions they want for a particular new file.
This is where umask comes in: the operating system maintains a per-process permission mask (called umask) which is subtracted from permissions of each newly created file. Most shells allow the user to manipulate this mask, and all programs executed by the shell inherit the mask from the shell (since it is inherited across fork).
When a file is created, all the bits that are set in the umask are cleared:

```
echo umask 777
umask 777
touch umask.txt
ls -l umask.txt
rm -f umask.txt
```

The clearing of permission bits from umask happens atomically: a file with the permission bits set does not exist, not even temporarily. Of course, 777 makes little sense as an actual umask, but it does demonstrate what is going on. A more reasonable (though also somewhat paranoid) option would be 077, which means new files are only accessible to the owner (i.e. the creator of the file, i.e. us):

```
echo
echo umask 077
umask 077
touch umask.txt
ls -l umask.txt
rm -f umask.txt
```

Let's try to create a file that is marked executable by the tool that creates it. A compiler is a good example:

```
echo
echo umask 077 with cc
cc empty.c
ls -l a.out
rm -f a.out
```

A somewhat less reasonable umask (again only useful for demonstration purposes) would deny any executable bits:

```
echo
echo umask 133 with cc
umask 133
cc empty.c
ls -l a.out
rm -f a.out
```

And finally, a fairly common default would be 022: this prevents anyone but the owner from writing into new files, but anyone else can read and/or execute them if they wish so.

```
echo
echo umask 022
umask 022
touch file.txt
cc empty.c
ls -l a.out file.txt
rm -f a.out file.txt
```

Possibly useful: if you run umask without arguments, it will print the currently active umask. That said, let's look at creating files in C now, and specifically setting their permissions upon creation. See perm.c.

Part 10.7: perm.c (source code)

Let's look at how to set permissions when we create files in C.

```
#include <fcntl.h>    /* open */
#include <sys/stat.h> /* chmod, fchmod */
#include <unistd.h>   /* close */

int main()
{
```

You might remember that we used open to create new files. At the time, the mysterious third argument remained without a proper explanation. As you can surely see now, this is nothing else but an octal representation of permissions. In C, a leading 0 in a numeric literal means the number is written out in octal (just like 0x means what follows is hexadecimal). Please keep in mind that the operating system will apply umask to the permissions given here in the third argument.

```
    int fd1 = open( "create1.txt", O_CREAT | O_RDWR, 0711 );
    int fd2 = open( "create2.txt", O_CREAT | O_RDWR, 0600 );
    int fd3 = open( "create3.txt", O_CREAT | O_RDWR, 0600 );
    int fd4 = open( "create4.txt", O_CREAT | O_RDWR, 0600 );
```

Besides setting the permission bits at file creation time, we can change them using two system calls that resemble the chmod program. However, in C, we always must supply the permission bits as a number (and for convenience, we usually stick to octal literals). The chmod syscall is straightforward: it takes a file path and a number. See man 2 chmod. Of course it might fail, but we currently don't care.

```
        chmod( "create3.txt", 0 );
```

The fchmod function is a little different, because it does not use a file path to change the permission bits. Rather, it takes a file descriptor, which corresponds to an i-node reference and changes the i-node via this fd. Again, the function may fail.

```
        fchmod( fd4, 0100 );
```

That's it, let's clean up.

```
        close( fd4 );
        close( fd3 );
        close( fd2 );
        close( fd1 );
    }
```

With this sorted out, let's look at another exercise, this time in C: 2.txt.

Part 10.8: Exercise 2

1. Write a C program that creates an empty file with permissions given on the command line. The name of the file comes first, then comes a single octal number representing the permissions.
2. To convert the number from a string to an actual number, use strtol with base set to 8 and endptr set to NULL. See also man strtol.
3. You will want to call the C function umask, which works the same as the shell builtin we have seen before.

This is pretty much all about permissions / access modes. Let's have a look at users using shell commands for a bit – we will look at C APIs later on. Next stop: users.sh.

Part 10.9: users.sh (source code)

While we can't manipulate users, there is a bunch of commands that print information and which are available to us. Let's have a look. The first command to check out is called simply w. Note that the command head as used here discards all but the first 5 lines of output.

```
echo output from w
w | head -n 5
```

The command prints information about users who are logged on, along with commands that they are currently executing. A more basic (and more standard) variant is a command called who, which only prints a subset of the info provided by w. Most notably, the currently executed command is missing:

```
echo
echo output from who
who | head -n 5
```

Feel free to experiment with those commands yourself. To learn your own username, you can use the command whoami (who am i?), like this:

```
echo
echo info about the current user
whoami
```

To print the list of groups attached to the active session, we can use groups. There can be more than one.

```
groups
```

Finally, the info from the two previous commands, along with some extras, can be obtained from id. Unlike groups, this tells us which group is primary and also gives numeric identifiers for each piece of information. The output of id is meant to be machine readable.

```
id
```

The last two commands can be also invoked on other users (i.e. not yourself). This works as follows:

```
echo
echo info about the user kontr
groups kontr
id kontr
```

This is about all the user- and group-related commands that we will look at. Let's have a quick look at passwd.txt before going on to C.

Part 10.10: User Database

A simple, local database of users and groups is, on UNIX computers, stored in text files in the /etc directory. This database is usually present even when it is supplemented by a network-based database like LDAP (which is the case on aisa and on school computers in general).
You can look at the files with a text editor or just cat them, and quite easily understand the format from just looking. More details can be found in man 5 passwd.

```
$ cat /etc/passwd
$ cat /etc/group
```

Next up: getid.c.

Part 10.11: getid.c (source code)

```
#include <unistd.h> /* getpid, getuid, getgid */
#include <stdio.h>  /* printf */
```

There are various identifier numbers associated with each process. We have seen the process id before, but we can obtain the user identifier (owner of the process) and the group identifier (primary group of the process) just as easily, with getuid and getgid.

```
int main()
{
    printf( "pid: %d\n", getpid() );
    printf( "uid: %d\n", getuid() );
    printf( "gid: %d\n", getgid() );
}
```

Next: getgroups.c

Part 10.12: getgroups.c (source code)

```
#include <unistd.h> /* getgroups */
#include <stdio.h>  /* printf */
```

Each process belongs to exactly one user, but it can have multiple active groups attached to it. This is mainly because users can belong to multiple groups, and they gain permissions from each of them.

```
int main()
{
```

The getgroups function will give us the group id's of all the groups for which we have permissions. But there is more than one, so it cannot come in the return value: like with stat before, we need to prepare some memory for getgroups to write the information into. Let's make space for 10 gids.

```
    gid_t grp[ 10 ];
```

When calling getgroups, we need to first tell it how much space we prepared – how many gids it can write into the memory – and then the memory itself. If the actual number of groups exceeds the space, the function fails. Otherwise, it returns the number of groups that we are actually in, and fills in their numbers in first n fields of grp.

```
    int i, n = getgroups( 10, grp );
```

We now print out the valid identifiers. Beyond n, the memory is garbage!

```
    for ( i = 0; i < n; ++i )
        printf( "%d\n", grp[ i ] );
}
```

Next: getpwent.c

Part 10.13: getpwent.c (source code)

The first thing we need to do is set a so-called feature macro, because out of the box, the getpwent function is not visible. Just consider this to be a magic line that lets us use getpwent.

```
#define _POSIX_C_SOURCE 201800
#define _BSD_SOURCE /* needed for getpwent */

#include <pwd.h>   /* getpwent */
#include <stdio.h> /* printf */

int main()
{
```

Each time getpwent is called, it returns the record corresponding to the next line of /etc/passwd. We will call it 3 times and hence print out 3 user records.

```
    int i;

    for ( i = 0; i < 3; ++i )
    {
        struct passwd *pw = getpwent(); /* obtain the next entry */
```

We have seen structures like this before (e.g. struct stat). The structure has various fields, the most interesting of which we will print. There isn't much else to this.

```
        printf( "name: %s\n", pw->pw_name );
        printf( "uid: %d\n", pw->pw_uid );
        printf( "gid: %d\n", pw->pw_gid );
        printf( "home: %s\n", pw->pw_dir );
        printf( "shell: %s\n", pw->pw_shell );
        printf( "\n" );
    }
}
```

Next: getpwuid.c.

Part 10.14: getpwuid.c (source code)

```
#define _POSIX_C_SOURCE 201800 /* same as before */
#include <pwd.h>   /* getpwuid */
#include <stdio.h> /* printf */

int main()
{
```

If we are interested in the passwd entry of a particular user, we don't have to find it ourselves: the function getpwuid (and getpwnam) can find it for us. In this case, getpwuid will search by the numeric user id. Let's then look up our own entry.

```
    struct passwd *pw = getpwuid( getuid() );
    printf( "name: %s\n", pw->pw_name );
    printf( "uid: %d\n", pw->pw_uid );
    printf( "gid: %d\n", pw->pw_gid );
    printf( "home: %s\n", pw->pw_dir );
    printf( "shell: %s\n", pw->pw_shell );
    printf( "\n" );
}
```

Next: getpwnam.c.

Part 10.15: getpwnam.c (source code)

```
#define _POSIX_C_SOURCE 201800 /* as before */
#include <pwd.h>   /* getpwnam */
#include <stdio.h> /* printf */
```

This program prints the basic information about a user whose login name is given to it as a command-line argument.

```
int main( int argc, const char **argv )
{
    if ( argc != 2 ) /* check command line arguments */
        return fputs( "expected 1 username as an argument\n", stderr
), 1;
```

Like getpwuid, the getpwnam function looks up user records, but this time by their login (user name).

```
    struct passwd *pw = getpwnam( argv[1] );
    printf( "name: %s\n", pw->pw_name );
    printf( "uid: %d\n", pw->pw_uid );
    printf( "gid: %d\n", pw->pw_gid );
    printf( "home: %s\n", pw->pw_dir );
    printf( "shell: %s\n", pw->pw_shell );
    printf( "\n" );
    return 0;
}
```

Next: getgrent.c.

Part 10.16: getgrent.c (source code)

```
#define _POSIX_C_SOURCE 201800 /* still the same */
#include <grp.h>   /* getgrent */
#include <stdio.h> /* printf */
```

This program prints info about groups (which are stored in /etc/groups). The functions to access group information are analogous to those for accessing user (passwd) information. This program basically prints info from lines 10-13 of /etc/groups.

```
int main()
{
    int i, j;
```

We first skip the first 10 entries because they are quite boring.

```
    for ( i = 0; i < 10; ++i )
        getgrent();

    for ( i = 0; i < 3; ++i )
    {
        struct group *grp = getgrent();
```

Groups have names and identifiers, just like users… let's print those first.

```
        printf( "name: %s\n", grp->gr_name );
        printf( "gid: %d\n", grp->gr_gid );
```

However, users only have a fixed number of fields attached to them. The situation with groups is somewhat different, because they contain a list of all users which belong to the group. We don't know beforehand how many such entries there are. The information is stored just like argv: an array of strings. We know we hit the end when we encounter a NULL string (pointer). Let's print all the member users then:

```
        for ( j = 0; grp->gr_mem[j]; ++j )
            printf( "member: %s\n", grp->gr_mem[j] );
        printf( "\n" );
    }
}
```

Next: 3.txt.

Part 10.17: Exercise 3

1. Write a C program that prints the groups of the current user (not the current process, i.e. not what getgroups does).
2. You can either use getgrent and iterate, or you can use getgrouplist. However, you should know that the latter is not part of POSIX.
3. You will also need getuid and getpwuid.
4. Recall that strings in C are compared using the function strcmp.

That's all for today. Follows homework, in hw.txt.

Part 10.18: Homework

1. Write a C program which does the following:

```
a. if given an argument, that argument is a user name
b. print the numeric id of that user
c. print the numeric id of the primary group of the user
d. print the name of the primary group of the user
e. if no argument is given, print the name of the current user,
   along with c. and d.
```

2. You can use any of: getpwent, getpwnam, getpwuid, getgrent, getgrnam and getgrgid.
3. Print each piece of info on a separate line, with no extra text.

Part 11: C API Redux

Like we did two weeks ago with shell, we will review some of the C programming that we did over the course. Since C is somewhat harder and more time-consuming than shell, there are only 8 exercises this week.

Part 11.1: Exercise 1

Implement a C program that does what ln does: take 2 arguments and create a hardlink of a file given by the first under the path given by the second.

Part 11.2: Exercise 2

Implement a small C program that does what tee does: copy its standard input onto its standard output, while also copying it into a file given to it as an argument (unlike standard tee, your program only needs to write into a single file).

Part 11.3: Exercise 3

Implement a C program that runs another program given in the first argument (passing all other args to this program). Recall execvp from seminar 6. Print for how many seconds this other program was running (see also man 3 time).

Part 11.4: Exercise 4

Implement a C program that limits the time for which another program may run. The first argument is the number of seconds for the time limit (recall atoi), the remaining arguments are the program to run along with arguments.
It is OK if the program always runs for exactly the number of seconds given. If you wanted to quit as soon as the command ends, you could use waitpid(pid, &status, WNOHANG) (see the man page for more details).

Part 11.5: Exercise 5

Write a C program that does what nc does, but only in one direction and on a UNIX domain socket:

1. connect to a UNIX domain socket with a path given by the first argument,
2. send any data from stdin into the socket.

You can use nc -U -l socket for testing.

Part 11.6: Exercise 6

Do what nc -l does, but only in one direction and using UNIX domain sockets:

1. listen on a socket given by the first argument,
2. when a client connects, copy anything it sends to your standard output,
3. exit when the client disconnects.

You can use nc -U socket to test, or you can use your own program from the previous exercise.

Part 11.7: Exercise 7

Write a C program that prints all the directories under the current working directory, in pre-order (i.e. after you print a directory name, recurse into the directory). Print one directory name per line.

Part 11.8: Exercise 8

Write a C program which prints the sum of the sizes of all the regular files in the current directory.

Part 11.9: Homework

Write a C program, lndir.c that takes 2 arguments, first of which is an existing directory. The program will:

1. create a directory given by the second argument. If it already exists (or is not a directory), bail (exit with a non-zero exit status),
2. for each regular file in the directory given by the first argument, create a hardlink to this file in the newly created directory (under the same name as the original).

You can also submit the following exercises to earn an activity point for this week: 02, 03, 05 and 07.