Shell Scripting

Videos

Video	Length	Slides
Permissions	20 minutes	slides
shell scripting 1	17 minutes	slides
shell scripting 2	21 minutes	slides

Exercises

- File permissions
- Shell scripting

File permissions

Log in to your Debian VM for the following exercises.

Create a user and a group

Create a new user with `sudo adduser NAME` - I'm going to be using `brian` as an example name in these notes. When it asks for a password, you can just use `brian` or something; it will complain about the password being too short but it will create the user anyway. You can skip the GECOS information asking for a full name and phone number---it's just to help an admin contact you if needed.

Check the user and group files with `tail /etc/passwd` and `tail /etc/group` to check that the new user has been created - `tail` displays the last 10 lines of a file by default; `tail -n N FILE` would display the last N lines. Your new user `brian` (or whatever you called them) should appear in both files. Also check with `ls -l /home` that the home directory for Brian exists and is set to the correct user and group.

Time to change user: `su brian` and enter the password. Notice that the prompt has changed to `brian@debian12:/home/vagrant$` (at least if you started off in that folder). So the user has changed, and because `/home/vagrant` is no longer the current user's home directory, it gets written out in full. Run `cd` to go home followed by `pwd` and check that you are now in `/home/brian` or whatever you called your new user.

Next, create a user `nigel` (or some other name) add both your two new users, but not `vagrant`, to the group `users` (which already exists) using the command `sudo addgroup USERNAME GROUPNAME`, where group and username are changed accordingly. Note: `brian` cannot use sudo, so you have to exit his terminal to get back to one running as vagrant for this.

Explore file permissions

As user `brian` (or whatever you called your first new user), set up your home directory using what you learnt in the videos so that

- You can do everything (rwx).
- Members of the `users` group can list files and change to your home directory, but not add/remove files. You will need to change the group of your home directory to `users` for this, using the command `chgrp -R GROUPNAME DIRECTORY`.
- Everyone else cannot do anything with your home directory.

Create a file in your home directory, e.g. `nano readme.txt` then add some content.

Check, by using `su USERNAME` to log in as the different users, that:

- `nigel` can view Brian's home directory but not create files there;
- `nigel` can view but not edit Brian's readme file;
- `vagrant` cannot list files in or enter Brian's home directory at all. What happens when you try?

Of course, vagrant can use sudo to get around all these restrictions. Permissions do not protect you from anyone who can become root.

Also as `brian`, make a `private` subdirectory in your home folder that no-one but you can access (read, write or execute). Create a file `secret.txt` in there with `nano private/secret.txt` as user `brian` from Brian's home directory, and put something in it. Do not change any permissions on `secret.txt` itself.

Check as Nigel that you can see the folder itself, but not cd into it nor list the file. Check that even knowing the file name (`cat /home/brian/private/secret.txt`) as Nigel doesn't work.

Using `ls -l` as Brian in both `~` and `~/private`, compare the entries for the files `~/readme.txt`, `~/private/secret.txt` and the folder `~/private`. Why do the groups of the two files differ?

Note that, even though the secret file has read permissions for everyone by default, Nigel cannot read it. The rule is that you need permissions on the whole path from `/` to a file to be able to access it.

This is another reminder that if you want to store private files on a lab machine, then put it in a folder that is only accessible to you. Other students can read your home directory by default, and they would be able to look at your work. This has led to plagiarism problems in the past, but good news: we keep logs and can usually figure out what happened! `:-)` .

Altenatively you could remove permissions from everyone else on your home directory there, but this prevents you from being able to share files in specific folders that you do want to share with other students.

Setuid

We are going to create a file to let Nigel (and others in the users group) send Brian messages which go in a file in his home directory.

As Brian, create a file `message-brian.c` in your home directory and add the following lines:

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

const char *filename ="/home/brian/messages.txt";

int main(int argc, char **argv) {
  if (argc != 2) {
    puts("Usage: message-brian MESSAGE");
    return 1;
  }
  FILE *file = fopen(filename, "a");
  if (file == NULL) {
    puts("Error opening file");
    return 2;
  }
  int r = fputs(argv[1], file);
  if (r == EOF) {
    puts("Error writing message");
    return 2;
  }
  r = fputc('\n', file);
  if (r == EOF) {
    puts("Error writing newline");
    return 2;
  }
  fclose(file);
  return 0;
}
```

Compile it with `gcc -Wall message-brian.c -o message-brian` (you should not get any warnings) and check with `ls -l`, you will see a line like

```
-rwxr-xr-x    1 brian     brian         19984 Oct 28 13:26 message-brian
```

These are the default permissions for a newly created executable file; note that gcc has set the three `+x` bits for you. Still as Brian, run `chmod u+s message-brian` and check the file again: you should now see `-rwsr-xr-x` for the file permissions. The `s` is the setuid bit.

As Nigel (`su nigel`), go into Brian's home directory and run `./message-brian "Hi from Nigel!"`. The quotes are needed here because the program accepts only a single argument.

Now run `ls -l` and notice that a `messages.txt` has appeared with owner and group `brian`. Check the contents with `cat messages.txt`. Although Nigel cannot create and edit files in Brian's home directory himself (he can't edit `messages.txt` for example, although he can read it), the program `message-brian` ran as Brian, which let it create the file. Nigel can send another message like this (`./message-brian "Hi again!"`), which gets appended to the file: try this out.

This shows how setuid programs can be used to allow other users to selectively perform specific tasks under a different user account.

Warning: writing your own setuid programs is extremely dangerous if you don't know the basics of secure coding and hacking C programs, because a bug in such a program could let someone take over your user account. The absolute minimum you should know is the contents of our security units up to and including 4th year.

A general task for a security analyst might be finding all files with the setuid bit set on a system. You can try this yourself, but return to a vagrant shell first so that you're allowed to use sudo:

```
sudo find / -perm /4000
```

You might get some errors relating to `/proc` files, which you can ignore: these are subprocesses that find uses to look at individual files.

Apart from `message-brian`, you'll find a few files by default: `sudo`, `mount`, `umount` and `su`. The first one you already know; look up what the next two do and think about why they are setuid. Specifically, what kinds of (un)mounting are non-root users allowed to do according to the manual pages?

Look up the `passwd` program in the manual pages. Why might that program need to be setuid?

Sudo

Make sure your terminal is running as `brian` and try a `sudo ls`. You will see a general message, you will be asked for your password, and then you will get the error `brian is not in the sudoers file.  This incident will be reported.` (This means that an entry has been logged in `/var/log/messages`.)

So, `brian` can currently not use sudo. Switch back to `vagrant` and run the command `sudo cat /etc/sudoers`. Everything is commented out except `root ALL=(ALL) ALL` and the last line `#includedir /etc/sudoers.d` (this is not a comment!) which contains a single file `vagrant` with the line `vagrant ALL=(ALL) NOPASSWD: ALL` which is why vagrant can use sudo in the first place.

However, note the commented lines such as

```
# %wheel ALL=(ALL) NOPASSWD: ALL
# %sudo ALL=(ALL) ALL
```

If uncommented, the first one would let everyone in group wheel run commands using sudo (this is the default on some other linux distributions), whereas the second one would allow everyone in the group `sudo` to do this, but would prompt for their own password beforehand.

Let's allow people in the users group to reboot the machine. Open a root shell with `sudo su` as vagrant; this is so we don't get locked out if we break sudo.

Edit the sudoers file with `visudo` as root, and add the following line:

```
%users ALL=(ALL) /sbin/reboot
```

and save the sudoers file.

Warning:: Never edit `/etc/sudoers` directly and *always* use `visudo` instead. If you make a mistake and add a syntax error to the file then `sudo` will refuse to work. If your root account doesn't have a password (some people don't like that as a security precaution) then you'll have to spend the next half-hour figuring out how to break into your own computer and wrestle back control. There is almost always a command to check a config file before replacing the current one: the same advice also applies to the ssh config files. If you break them you might have to travel to wherever the server is with a keyboard and a monitor.

You can now switch back to `brian` (check the prompt to make sure you are Brian) and do `sudo reboot`. After asking for Brian's password, the virtual machine will now reboot, which you notice because you get kicked out of your ssh connection. Another `vagrant ssh` after a few seconds will get you back in again.

Advanced note

After rebooting, your `/vagrant` shared folder might not work. In this case, log out and do `vagrant halt` then `vagrant up` and `vagrant ssh` again on the host machine.

When vagrant boots your VM, it automatically sets up the shared folder, but this doesn't always work if you reboot the VM yourself.

Shell Scripting

Shell scripting is a huge topic - the shell is a full programming language after all. In the video for this activity you have only seen a very brief introduction.

Compile helper exercise

Write a shell script in a file called `b` (for build) that does the following:

- Your script should run under any Bourne-compatible shell (e.g. not just `bash`), and it should be written so that you can call it with `./b`.
- `./b compile NAME` should compile the file of the given name, so for example `./b compile hello` should run `gcc -Wall -std=c11 -g hello.c -o hello`.
- However, your script should accept both `./b compile hello` and `./b compile hello.c` as input, and do the same thing in both cases, namely compile `hello.c`. The output file for gcc in both cases should be called just `hello`.
- If the source file you provided as an argument does not exist (adding `.c` if necessary) then the script should print an error message and return a nonzero exit status - *not* invoke the C compiler.
- `./b run NAME` should run the program, assuming it exists in the current folder, so both `./b run hello` and `./b run hello.c` should run `./hello`. If it does not exist, again print an error message and exit with a nonzero status, don't try and run the program.
- `./b build NAME` should first compile the C source file, and then if the compile was successful it should run the program. If the compile failed, it should not try and run the program.
- If you call `./b` without any parameters, or `./b COMMAND` with a command other than compile or run or build, it should print some information on how to use it. If you call `./b compile` or another command with no filename at all, then the script should print an error message and exit with a nonzero exit status.

You now have a useful tool for when you are developing C programs. Of course you can add other features of your own like a `debug` command that compiles the file and launches it in `gdb`.

If you already know about makefiles, you might wonder why we don't just use `make` for this. You are correct, but this is specifically a shell scripting exercise. We will learn about makefiles soon.

Strict Mode

Some programming languages have an optional *strict mode* which treats some constructs as errors that people often do by mistake. It is similar in spirit to `-Werror` in C that treats all warnings as errors. This page suggests using the following line near the top of your shell scripts: `set -euo pipefail`. (It also talks about IFS to improve string handling with spaces, but that's a separate matter.)

You might want to use these yourself if you get into shell scripting. `set` is a shell internal command that sets shell flags which controls how commands are run.

- `set -e` makes the whole script exit if any command fails. This way, if you want to run a list of commands, you can just put them in a script with `set -e` at the top, and as long as all the commands succeed (return 0), the shell will carry on; it will stop running any further if any command returns nonzero. It is like putting `|| exit $?` on the end of every command.
- `set -u` means referencing an undefined variable is an error. This is good practice for lots of reasons.
- `set -o pipefail` changes how pipes work: normally, the return value of a pipe is that of the *last* command in the pipe. With the `pipefail` option, if any command in the pipeline fails (non-zero return) then the pipeline returns that command's exit code.

A couple of notes on `set -u`: if you write something like `rm -rf $FOLDER/` and `$FOLDER` isn't set, then you don't accidentally end up deleting the whole system! Of course, most `rm` implementations will refuse to delete `/` without the `--no-preserve-root` option, and you should not have that trailing slash in the first place. There was a bug in a beta version of Steam for linux where it tried to do `rm -rf "$STEAMROOT/"*` to delete all files in a folder (which explains the slash), but the variable in some cases got set to the *empty string*, which `-u` would not protect against. This was an installer script, so it ran as root which made things even worse.

Exercise: think of an example in a shell script where `pipefail` makes a difference, that is where the last command in a pipe could succeed even if a previous one fails. As a counter-example, `cat FILE | grep STRING` would fail even without `pipefail` if the file does not exist, because grep would immediately get end-of-file on standard input.

Build Tools 2

Videos

Video	Length	Slides
Build Tools 1	17 minutes	slides
Build Tools 2	12 minutes	slides

Exercises

- C
- Python
- Java
- Spring

Build tools: C

In this exercise you will practice the traditional way of building C projects from source. We are going to use the sqlite database as an example project to build.

Download the source file https://sqlite.org/2021/sqlite-autoconf-3340100.tar.gz into your VM with `wget` or similar and extract it with `tar -zxvf FILENAME`. This creates a subfolder, do a `cd` into it.

You can see a file called `INSTALL` which you can open in a text editor to find the standard instructions:

> Briefly, the shell commands `./configure; make; make install` should configure, build, and install this package.

Configure

If you look at the first line of the configure script, it starts `#! /bin/sh` as that path should be valid on just about any vaguely posix-compatible system. The whole thing is just over 16000 lines, so you don't want to read all of it.

Run the script with `./configure`. You can see that it does a lot of checking, including things like:

- Whether your system has a C compiler.
- Whether your C compiler is gcc.
- Whether your C compiler actually works.
- Whether standard headers like `string.h` or `stdlib.h` exist.
- Whether the readline library is installed on your system.

Your configure script should run through and print `creating Makefile` on one of its last lines.

The configure script is basically a collection of tests for every single bug and oddity found on any system known to the autoconf developers that could break the build. For example, someone once reported a bug in a build on Sun OS 4 (released in 1988), so in lines 2422 and following of the configure script we read

---

```
# Use test -z because SunOS4 sh mishandles braces in ${var-val}.

# It thinks the first close brace ends the variable substitution.
```

```
test -z "$INSTALL_PROGRAM" && INSTALL_PROGRAM='${INSTALL}'
```

Make

Type `make` to build sqlite. If it's not installed, `sudo apt install make` will fix that.

Some of the compiler commands might take a while to run. While they're running, note the number of configuration variables (everything passed with a `-D`) involved; some of them turn on/off features (for example readline support is off if it can't find the header files for it on your system) and some of them set options specific to the operating system and compiler, for example `-DHAVE_STRING_H` defines whether `string.h` exists on your system.

These translate to `#ifdef` commands in the source files, for example in `shell.c` starting at line 121 we include readline, if the header files exist:

```
#if HAVE_READLINE
# include <readline/readline.h>
# include <readline/history.h>
#endif
```

The last command run by the makefile is

```
gcc [lots of options] -g -O2 -o sqlite3 sqlite3-shell.o sqlite3-sqlite3.o -
lreadline -lcurses
```

This should build an executable `sqlite3` that you can run (use `.q` to quit again).

If you want to, you can now type `sudo make install` to copy the executable to `/usr/local/bin`.

Advanced note

What do you do if it says it can't find a `.h` file, or can't link it to a library file (a `.so`)? C predates modern languages with package managers, so it probably means you haven't installed a library the code depends on. Luckily `apt-file` can be really helpful here: run `apt-file search <name of file>` to find out which package provides the file you're missing and install it.

I was trying to build a package that was complaining it couldn't find a library `libffi.so`: what package might have provided it?

Try not to panic if the software you're building won't build cleanly! Read the error message and fix the bug. Normally installing a library, or altering a path in the source code is enough to fix it. Being able to fix simple bugs yourself is what makes Linux (and other OSs) really powerful!

Build tools: Python

The Python programming language comes with a package manager called `pip`. Find the package that provides it and install it (**hint**: how did we find a missing library in the C exercise?).

We are going to practice installing the mistletoe module, which renders markdown into HTML.

- In python, try the line `import mistletoe` and notice that you get `ModuleNotFoundError: No module named 'mistletoe'`.
- Quit python again (Control-D) and try `sudo pip3 install mistletoe`. You should get a success message (and possibly a warning, explained below).
- Open python again and repeat `import mistletoe`. This produces no output, so the module was loaded.

Create a small sample markdown file as follows, called `hello.md` for example:

```
# Markdown Example

Markdown is a *markup* language.
```

Open python again and type the following. You need to indent the last line (four spaces is usual) and press ENTER twice at the end.

```
import mistletoe
with open('hello.md', 'r') as file:
    mistletoe.markdown(file)
```

This should print the markdown rendered to HTML, e.g.

```
<h1>Markdown Example</h1>\n<p>Markdown is a <em>markup</em> language.</p>
```

Advanced note

Python version 3 came out in 2008 and has some syntax changes compared to Python 2 (`print "hello world"` became `print("hello world")`). Version 2 is now considered deprecated; but the transition was *long* and *extremely painful* because changing the syntax of a thing like the print statement leads to an awful lot of code breaking and an awful lot of people preferring not to fix their code and instead just keep an old version of Python installed.

So whilst we were dealing with this it was typical for a system to have multiple versions of Python installed `python2` for the old one and `python3` for the newer on (and even then

these were often symlinks to specific subversions like `python2.6`), and then `python` being a symlink for whatever your OS considered to be the "supported" version.

Different OSs absolutely had different versions of Python (MacOS was particularly egregious for staying with Python 2 for far longer than necessary) and so a solution was needed, because this was just breaking things while OS designers bickered.

The solution is that for *most* dependencies (except for compiled libraries) we generally use a programming language's own package manager and ignore what the OS provides. For Python that means `pip` (occasionally called `pip3` or `pip2`).

Sometimes you'll see things telling you to install a package with `sudo pip install` but don't do that! It will break things horribly eventually. You can use pip without sudo, by passing the `--user` option which installs packages into a folder in your home directory (`~/.local`) instead of in `/usr` which normally requires root permissions.

Sometimes you'll still need to install a package through the OSs package manager (`numpy` and `scipy` are common because they depend on an awful lot of C code and so are a pain to install with `pip` as you have to fix the library paths and dependencies manually) but in general try and avoid it.

Python used to manage your OS should be run by the system designers; Python used for your dev work should be managed by you. And never the twain shall meet.

Scipy

We often use `scipy` for statistics, so you may as well install that too. Unfortunately, `pip` will not help you here because scipy depends on a C library for fast linear algebra. You could go and install all the dependencies (and you might have to do this if you need a specific version of it), but it turns out Debian has it all packaged up as a system package: Try searching for it with `apt search scipy`.

The following commands show if it is correctly installed, by sampling 5 times from a Normal distribution with mean 200 and standard deviation 10:

```
from scipy.stats import norm
norm(loc=200, scale=10).rvs(5)
```

This should print an array of five values that are not too far off 200 (to be precise, with about 95% confidence they will be between 180 and 220 - more on this in Maths B later on).

Avoiding sudo

If you need to install libraries you might be tempted to install them for all users by using `sudo pip` but this can lead to pain! If you alter the system libraries and something in the system depends on a specific version of a library then it can lead to horrible breakage and things not working (in particular on OSs like Mac OS which tend to update libraries less often).

Python comes with a mechanism called venv which lets you create a virtual python install that is owned by a user: you can alter the libraries in that without `sudo` and without fear of mucking up your host system. Read the docs and get used to using it---it'll save you a world of pain later!

Advanced note

`pip freeze | tee requirements.txt` will list all the packages your using and what version they are and save them in a file called `requirements.txt`.

`pip install -r requirements.txt` will install them again!

This makes it *super easy* to ensure that someone looking at your code has all the right dependencies without having to reel off a list of *go install these libraries* (and will make anyone whoever has to mark your code happy and more inclined to give you marks).

Build tools: Java

In the Java world,

- The `javac` compiler turns source files (`.java`) into `.class` files;
- The `jar` tool packs class files into `.jar` files;
- The `java` command runs class files or jar files.

A Java Runtime Environment (JRE) contains only the `java` command, which is all you need to run java applications if you don't want to do any development. Many operating systems allow you to double-click jar files (at least ones containing a special file called a `manifest`) to run them in a JRE.

A Java Development Kit (JDK) contains the `javac` and `jar` tools as well as a JRE. This is what you need to develop in java.

`maven` is a Java package manager and build tool. It is not part of the Java distribution, so you will need to install it separately.

You can do this exercise either in your VM, or on your own machine where you have probably already installed Java for the OOP/Algorithms unit, and you can use your favourite editor. The exercises should work exactly the same way in both cases, there is nothing POSIX-specific here.

Installing on Debian

On Debian, install the `openjdk-17-jdk` and `maven` packages. This should set things up so you're ready to go but if you have *multiple versions* of Java installed you may need to set the `JAVA_HOME` and `PATH` variables to point to your install.

For example:

```
export JAVA_HOME='/usr/lib/jvm/java-17-openjdk'
export PATH="${PATH}:${JAVA_HOME}/bin"
```

Advanced note

Debian also has a special command called `update-alternatives` that can help manage alternative development environments for you. Read the manual page!

Installing on your own machine

Use whatever package manager your OS comes with. If you can't and have to install it manually:

- download the OpenJDK distribution
- unzip it somewhere
- add the binaries folder to your `PATH`
- set the `JAVA_HOME` variable to point to the folder where you unzipped the JDK.

To install maven, follow these instructions which again involve downloading a ZIP file, unzipping it somewhere and then putting the `bin` subfolder on your `PATH`.

Note: `JAVA_HOME` must be set correctly for maven to work.

Running maven

Open a shell and type `mvn archetype:generate`. This lets you *generate an artifact from an archetype*, which is maven-speak for create a new folder with a maven file.

If you get a "not found" error, then most likely the maven `bin` folder is not on your path. If you're on a POSIX system and have used your package manager, this should be set up automatically, but if you've downloaded and unzipped maven then you have to `export PATH="$PATH:..."` where you replace the three dots with the path to the folder, and preferably put that line in your `~/.profile` too.

|||advanced On Windows, if you must user it, search online for instructions how to set up the path variable, or you can drag-and-drop the `mvn.cmd` file from an Explorer window into a Windows CMD terminal and it should paste the full path, then press SPACE and enter the arguments you want to pass. |||

The first time you run it, maven will download a lot of libraries.

Maven will first show a list of all archetypes known to humankind (3046 at the time of counting) but you can just press ENTER to use the default, 2098 ("quickstart"). Maven now asks you for the version to use, press ENTER again.

You now have to enter the triple of (groupId, artifactId, version) for your project - it doesn't really matter but I suggest the following:

```
groupId: org.example
artifactId: project
version: 0.1
```

Just press ENTER again for the following questions, until you get a success message.

Maven has created a folder named after your artifactId, but you can move and rename it if you want and maven won't mind as long as you run it from inside the folder. Use `cd project` or whatever you called it to go inside the folder.

If you're in a POSIX shell, then `find .` should show everything in the folder (in Windows, `start .` opens it in Explorer instead):

```
.
./src
./src/main
./src/main/java
./src/main/java/org
./src/main/java/org/example
./src/main/java/org/example/App.java
./src/test
./src/test/java
./src/test/java/org
./src/test/java/org/example
./src/test/java/org/example/AppTest.java
./pom.xml
```

This is the standard maven folder structure. Your java sources live under `src/main/java`, and the default package name is `org.example` or whatever you put as your groupId so the main file is currently `src/main/java/org/example/App.java`. Since it's common to develop Java from inside an IDE or an editor with "folding" for paths (such as VS code), this folder structure is not a problem, although it's a bit clunky on the terminal.

The POM file

Have a look at `pom.xml` in an editor. The important parts you need to know about are:

The artifact's identifier (group id, artifact id, version):

```
<groupId>org.example</groupId>
<artifactId>project</artifactId>
<version>0.1</version>
```

The build properties determine what version of Java to compile against (by passing a flag to the compiler). Unfortunately, the default maven template seems to go with version 7 (which for complicated reasons is called 1.7), but version 8 was released back in 2014 which is stable enough for us, so please change the 1.7 to 1.8 (there are some major changes from version 9 onwards, which I won't go into here):

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

The dependencies section is where you add libraries you want to use. By default, your project uses `junit`, a unit testing framework - note that this is declared with `<scope>test</scope>` to say that it's only used for tests, not the project itself. You do not add this line when declaring your project's real dependencies.

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

The `<plugins>` section contains the plugins that maven uses to compile and build your project. This section isn't mandatory, but it's included to "lock" the plugins to a particular version so that if a new version of a plugin is released, that doesn't change how your build works.

The one thing you should add here is the `exec-maven-plugin` as follows, so that you can actually run your project:

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>exec-maven-plugin</artifactId>
    <version>3.0.0</version>
    <configuration>
        <mainClass>org.example.App</mainClass>
    </configuration>
</plugin>
```

The important line is the `mainClass` which you set to the full name (with path components) of your class with the `main()` function.

Compile, run and develop

`mvn compile` compiles the project. The very first time you do this, it will download a lot of plugins, after that it will be pretty fast. Like `make`, it only compiles files that have changed since the last run, but if this ever gets out of sync (for example because you cancelled a

compile halfway through) then `mvn clean` will remove all compiled files so the next compile will rebuild everything.

The `App.java` file contains a basic "Hello World!" program (have a look at this file). You can run the compiled project with `mvn exec:java` if you've set up the plugin as above. After you've run it the first time and it's downloaded all the files it needs, lines coming from maven itself will start with `[INFO]` or `[ERROR]` or similar, so lines without any prefix like that are printed by your program itself. You should see the hello world message on your screen.

The development workflow is now as follows: you make your edits, then run `mvn compile test exec:java` to recompile, run your tests, then run the program. (Like `make`, you can put more than one target on a command, separated by spaces.)

`mvn test` runs the tests in `src/test/java`. There is an example test already created for you (have a look).

`mvn package` creates a jar file of your project in the `target/` folder.

I assume that you will be storing your Java projects in git repositories. In this case, you should create a file `.gitignore` in the same folder as the `pom.xml` and add the line `target/` to it, since you don't want the compiled classes and other temporary files and build reports in the repository. The `src/` folder, the `pom.xml` and the `.gitignore` file itself should all be checked in to the repository.

Exercise: make a change to the Java source code, then recompile and run with maven.

Adding a dependency

Thymeleaf is a Java templating library. It lets you write a template file or string for example (depending on the syntax of your library)

```
Hello, ${name}!
```

which you can later render with a particular name value. This is one of the standard ways of creating web applications, for example to display someone's profile page you would write a page template that takes care of the layout, styles, links etc. but uses template variables for the fields (name, email, photo etc.) which you render when someone accesses the profile page for a particular person. You will see this in more detail in your SPE project next year.

To use Thymeleaf or any other library, you first have to add it to your pom file. Go to mvnrepository.org and search for Thymeleaf, then find the latest stable ("release") version. There is a box where you can copy the `<dependency>` block to paste in your pom file. The next `mvn compile` will download thymeleaf and all its dependencies.

Next, make a template file called `unit` in the folder `src/main/resources/templates` (you will have to create the folder first), and put the following lines in it:

```
Unit: [(${name})]

In this unit, you will learn about:

[# th:each="topic: ${topics}"]
  - [(${topic})]
[/]
```

This is thymeleaf "text" syntax, where the first line renders the value of a variable and the third-from-last line is the template equivalent of a 'for' loop that renders its contents once for each element in a list (or other collection data structure).

Thymeleaf needs to know where to find its template files, and in this example we are going to demonstrate loading resources from the classpath because that is the correct way to work with resources in a java application (there are special considerations for web applications, but they usually end up using the classpath in the end anyway).

In your Java source file, you can now do the following. First, the imports you will need:

```
import java.util.List;
import java.util.Arrays;

import org.thymeleaf.TemplateEngine;
import org.thymeleaf.context.Context;
import org.thymeleaf.templatemode.TemplateMode;
import org.thymeleaf.templateresolver.ClassLoaderTemplateResolver;
```

And the code:

```
ClassLoaderTemplateResolver resolver = new ClassLoaderTemplateResolver();
resolver.setTemplateMode(TemplateMode.TEXT);
resolver.setPrefix("templates/");

TemplateEngine engine = new TemplateEngine();
engine.setTemplateResolver(resolver);

Context c = new Context();
c.setVariable("name", "Software Tools");
List<String> topics = Arrays.asList("Linux", "Git", "Maven");
c.setVariable("topics", topics);
String greeting = engine.process("unit", c);

System.out.println(greeting);
```

Compile and run this, and you should see:

```
Unit: Software Tools

In this unit, you will learn about:

  - Linux
  - Git
  - Maven
```

Let's look at how the code works.

1. A template resolver is a class that finds a template when you give it a name (here: "unit"). In this case we use a resolver that loads off the classpath, so we just have to put the template files somewhere under `src/main/resources`; we tell it that we want the template files treated as text (e.g. not HTML), and that the template files are in a subfolder called `templates`.
2. The template engine is the class that does the work of rendering the template, once the resolver has found the source file.
3. To render a template, you need a template name for the resolver to look up, and a context - an object on which you can set key/value parameters. In this case we're setting the key "name" to "Software Tools" and the key "topics" to a list of three topics. The names and types of keys obviously have to match what's in the template file.

Exercise: rewrite this example to be a bit more object-oriented by creating a unit class:

```
public class Unit {
    private String name;
    private List<String> topics;
    public Unit(String name, List<String> topics) {
        this.name = name;
        this.topics = topics;
    }
    public String getName() { return this.name; }
    public List<String> getTopics() { return this.topics; }
}
```

You will still need one single `setVariable` call, and in the template the syntax `[(${unit.name})]` should translate into a call to the getter.

Advanced note

More recent releases of Java have wonderful things called `records` that make your life *a lot* easier. All that above code translates to just:

```
public record Unit(String name, List<String> topics) {}
```

Unfortunately support for more recent Java releases is a bit spotty (and worse in the *real* world). You'll need to get rid of the `maven.compiler.target` and `maven.compiler.source` bits you added in your pom.xml and replace it with a new:

```
<maven.compiler.release>17</maven.compiler.release>
```

Spring

The Spring framework helps you develop modern web / cloud / microservice / serverless / *(insert buzzword here)* applications in Java. Netflix, for example, uses it.

Advanced note

Jo is cantankerous and old and so doesn't use it and prefers to either use CGI scripts, or a rather weird old thing called inetd. They're out of fashion nowadays, but worth reading up on---especially inetd which can save you a bunch of code for a simple webapp.

Vagrant preparation

Web applications listen to a port (normally TCP port 80 for HTTP, 443 for HTTPS in production; 8000 or 8080 while in development). If you are following these exercises on your host OS, you can skip this preparation. If you are in a VM on vagrant, then although you can quickly get an application to work on port 8080 inside the VM, you need one extra step to make it accessible from your browser outside the VM.

Add the line

```
config.vm.network "forwarded_port", guest: 8080, host: 8080
```

to your Vagrantfile just below the `config.vm.box` line, then restart the VM by logging out and doing `vagrant halt` then `vagrant up`. Your firewall may pop up a warning and ask you to approve this.

Now, while the VM is running, any connection to port 8080 on your host OS will be sent to the VM (this also means that if you're doing web development on the host, you can't use port 8080 for anything else while the VM is up - if you need port 8080, just pick another port number for the `guest` part).

Spring set-up

Spring wants to make setting up a project as easy as possible, so go to start.spring.io and you get a graphical interface to create a spring/maven project.

- Pick "Maven Project" and "Java language".
- Enter a group id and artifact id (you can use `org.example` and `project`).
- Make sure "packaging" is on JAR and "Java" (version) is on 17 (or whatever you have installed).

- Under *Dependencies* on the right, click "Add Dependencies" and add "Spring Web".
- Click "Generate" at the bottom, this downloads a ZIP of a project (complete with pom.xml file and folders) that you can unzip either on the host or in the VM.

To unzip the file in the VM, place it in the same folder as your Vagrantfile on the host, then inside the VM it will appear in the `/vagrant` folder. The command `unzip` is your friend. Install it if not already installed.

This project uses Spring Boot, a library and plugin to help with building and running Spring applications, so the only maven command you need is

```
mvn spring-boot:run
```

which will recompile if necessary, and then run the application. Once it's running, you can go to localhost:8080 in a browser on your host OS (whether the Spring application is running on the host, or inside the VM as long as you've set up the port forwarding as described above). You will see an error message as there's no pages yet, but the application is running. Stop it with Control+C.

Development

Open the source file under `src/main/java/...`. I called mine "project" so the class is `ProjectApplication`, but the name doesn't matter.

- Add the `@RestController` annotation to the application class.
- Create a method as follows:

```
@GetMapping("/")
public String mainPage() {
    return "Hello, Software Tools!\n";
}
```

- Add the imports for the two annotations you've just used; they're both in the package `org.springframework.web.bind.annotation`.

Now you can re-run the project with maven, go to `localhost:8080` on your browser and you should see the message. Congratulations - you've built your first Java/Maven/Spring web application!

You can also access the web page from the terminal on your host machine with `wget localhost:8080 -q -O /dev/stdout` which should print "Hello, Software Tools!" on your terminal. If you just do `wget localhost:8080` it will save the output to a file with the default name `index.html`.

The `@GetMapping` means, when a HTTP GET request comes in for a URL with the path in the annotation (such as a browser would send), then run this function. For example, accessing `localhost:8080/pages/index.html` would look for a mapping `/pages/index.html` where as the `/` mapping covers when you type no path at all. You can start to see that URL paths on the web are modelled on POSIX paths as you learnt in this unit, with forward slashes and a concept of a "root folder" `/` which is what gets returned when you type a website's name in your browser without a path on the end.

You've seen that build tools like maven automate the process of downloading the libraries your project needs and compiling your project. The Spring Framework automates (as far as possible) the process of running these libraries when your application starts, and getting them all to work nicely together (there is a *lot* going on in the background from a web request arriving to your function being called to get the page contents). The obvious next step would be to use thymeleaf to render HTML templates to make proper pages. This is indeed something you'll learn about later in this unit, but there are a few more steps you'll need to know to do this - and that's quite enough material for one workshop.

Build Tools 1

Videos

Video	Length	Slides
Debugging	34 minutes	slides

Exercise

- Debugging exercise

Debugging exercise

Clone the repository `git@github.com:cs-uob/COMSM0085` if you have not done so already and open the folder `code/debugging`.

There is a program `stackcalc.c` that attempts to implement the specification in `stackcalc.txt` for a Reverse Polish Notation calculator, but it does not work correctly. For example, `1 2 +` should produce `3.0000` but produces `Error, operator on empty stack`. (Read the notes in the text file about how to compile the program with the required library.)

Your exercise is to debug and fix the program, making as few changes to the general structure as possible (so don't just rewrite the whole thing from scratch).

Bonus POSIX Activity

This activity is optional, and non-examinable. We recommend that you watch the videos anyway.

Videos

Video	Length	Slides
inodes	18 minutes	slides
The TTY	22 minutes	slides

Exercises

- inodes and system calls
- Concurrent programming in POSIX
- Pipes in C
- Input/Output in C
- Input/Output in POSIX
- The final challenge

Inodes and System Calls

In this exercise we will look under the hood of the `stat` system call, which returns information about an inode.

Note: for this exercise it's even more important than usual that you are using Debian within Vagrant, as you will get different results if you try it on Windows Subsystem for Linux or on a Mac for example.

A system call is a way for a linux user or program to interact with the kernel, and there are usually at least three ways of calling each one:

1. Execute the system call directly in assembly.
2. Use the wrapper function provided by your C library.
3. Use a command-line program provided by your distribution.

Preparation

Have a look at the manual page `man stat` for the `stat` system call. The abbreviated headers are:

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

int stat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
```

`stat` is the main system call: you give it a pathname and it fills a `struct stat` for you with information about the inode associated with this pathname, however if the pathname is a symbolic link then it follows the link and returns information about the target inode. If you do not want this behaviour, you can use `lstat` instead.

`fstat` takes an open file descriptor instead of a file name: this is fine for getting the inode information (in the kernel, a file descriptor contains an inode number) but you will not be able to get a file name back from this - remember, *files don't have names; names have files*.

Later in the manual page, it explains the `struct stat` contents. Let's have a look at the sources directly though:

- `nano /usr/include/sys/stat.h` shows you the header file and the function definitions, including the bitmasks for the mode bits e.g. `#define S_IRUSR 0400` is the "user can read" bit and the file type bits e.g. `#define S_IFDIR 0040000`. Note, in C, numbers with a leading 0 are octal!

- The definition of the `struct stat` is in another file included from `sys/stat.h`, namely `bits/stat.h`; open that in your editor too and have a look at it.
- The types of the fields in the structure (`dev_t`) etc. are yet in another file - `bits/alltypes.h` if you're curious - but eventually they get defined as `long`, through intermediate definitions of `_Int64` etc. Basically, on a 64-bit machine, most of these fields are 64 bits.

Run the following short C program to check the size of your `struct stat`; I get 144 bytes but note down if you get something different:

```
#include <sys/stat.h>
#include <stdio.h>
int main() {
    printf("Size: %lu bytes\n", sizeof(struct stat));
    return 0;
}
```

The assembly level

Create a file with the following content - the convention for assembly files is usually to end in `.s`, so I've called mine `dostat.s`:

```
.section .text
.global  _start

_start:
    mov $6,  %rax
    lea str, %rdi
    lea buf, %rsi
    syscall

    mov $60, %rax
    mov $0,  %rdi
    syscall

str: .asciz "/dev/stdin"

.section .data
buf: .skip 144
```

Change the 144 in the last line if you got a different size for your structure.

Let's see what's going on here:

1. `.section .text` is an assembly directive to say *the following is code*, more precisely *it goes in the code section (which we named 'text' for obscure historical reasons)*.
2. `.global _start` says to export the label `_start` to the linker, which is the assembly version of `main`: in fact, C programs really start at `_start` too as you can check by

setting a breakpoint on this label in a debugger, this function is part of the C library and it sets a few things up and then calls `main`. When you return from `main`, then `_start` does some cleanup and exits the program cleanly with the correct return value.

3. The way you invoke a system call is you put the system call number in the `rax` register, and parameters according to the platform convention - on Intel/AMD 64 bit, the first parameter goes in the `rdi` register, the second one in the `rsi` register. System calls and their parameters are documented in this table for example. Return values from system calls end up in the `rax` register again. Looking ahead a bit in our assembly code, system call 60 is `sys_exit` and takes an exit code in `rdi`, so the lines `mov $60, %rax; mov $0, %rdi; syscall` are the equivalent of `return 0;` in a main function of a C program or `exit(0);` anywhere else (indeed, that is the last thing the C library `_start` will do when `main` returns to it).
4. System call 6 is `sys_lstat`, so the lines `mov $6, %rax; lea str, %rdi; lea buf, %rsi; syscall` call `sys_lstat(str, buf)` where both `str` and `buf` are pointers. (`lea` stands for *load effective address* and is similar to the `&var` address-of operator in C).
5. `syscall` is an assembly instruction that hands control over to the operating system. It is comparable to a software interrupt as you might have learnt in Computer Architecture, but it is an optimised version (since many programs do a lot of system calls) that doesn't have the full overhead of the older interrupt mechanism on x86.
6. `.asciz` is a zero-terminated string in C style (which is what the kernel expects). `.section .data` says *the following goes in the data section*, which we need because the buffer variable needs to be written to and the code section is read-only when a program is running. `.skip` reserves a block of the given size in bytes, similar to `byte buf[144];` in C (you can read `char` for `byte` if you want).

Assemble the program with

- `as dostat.s -g -o dostat.o`
- `ld dostat.o -o dostat`

The first command is the assembler itself, which produces and object file (C compilers usually do this too, but you don't see it unless you ask for it). `-g` is the same as for gcc, it includes debug information. `ld` is the linker which produces an executable.

You can run the program with `./dostat`, but it will simply exit with status 0. What we want to do is debug it with `gdb dostat` (install `gdb` with `apk` if you don't have it installed already), then do the following:

- `break _start` to set a breakpoint.
- `run` to start running (and hit the breakpoint).
- `si` steps a single assembly instruction. Do this until you have passed the first `syscall` and land on the line `mov $60, %rax`.

- The memory at `buf` now contains filled-in `struct stat` for `/dev/stdin`, the standard input file. Look at this with `x/40xb &buf` (memory dump, show 40 hex bytes starting at the buffer).

Based on what you know about the `struct stat` memory layout, what is the inode number and mode of `/dev/stdin`? Note down the inode number in decimal, and the low 16 bits of the mode word in binary. Note that the memory layout is most likely little-endian, and you are working with 64-bit long integers.

From this information, and the bit patterns in `/usr/include/sys/stat.h`, decode the file type and permissions of `/dev/stdin`.

You can then quit gdb with `q`, and answer yes when it askes whether you want to terminate the program.

The C level

We will now do the same in C. Create this program, I've called it `exstat.c`, then compile and run it:

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

int main() {
    struct stat buf;
    char *str = "/dev/stdin";
    int r = lstat(str, &buf);
    if (r != 0) {
        puts("An error occurred.");
        return 1;
    }
    printf("The inode number is %lu.\n", buf.st_ino);
    if (S_ISDIR(buf.st_mode)) { puts("It's a directory.");}
    if (S_ISCHR(buf.st_mode)) { puts("It's a character device.");}
    if (S_ISBLK(buf.st_mode)) { puts("It's a block device.");}
    if (S_ISREG(buf.st_mode)) { puts("It's a regular file.");}
    if (S_ISFIFO(buf.st_mode)) { puts("It's a FIFO.");}
    if (S_ISLNK(buf.st_mode)) { puts("It's a soft link.");}
    if (S_ISSOCK(buf.st_mode)) { puts("It's a socket.");}

    return 0;
}
```

Here we can see that we:

1. Set up the buffer structure and execute the system call.
2. Check the return value! If it's not 0, then an error occurred - the file `/usr/include/bits/errno.h` contains a table of error codes, although the system call

will return the negative error code in register `rax`. The `man stat` manual page explains the meaning of each error code for this particular system call.
3. Print the inode number (in decimal) and the file type.

Check that you get the same inode number and file type as you did with the assembly version.

Symbolic links

The point of checking the file type is that `/dev/stdin` is a symbolic link. To find out where it points, you can use this function:

```
#include <unistd.h>
ssize_t readlink(const char *pathname, char *buf, size_t bufsiz)
```

This is another system call wrapper (readlink is system call 89) which takes the pathname of a symbolic link and writes its contents (e.g. the file it points at) in a buffer. However, be aware of the following:

- `readlink` does not zero-terminate its buffer! That is your responsibility as caller.
- The returned value (yet another unsigned long) indicates what happened:
  - A positive value indicates the number of bytes written, so you know where to put the zero byte at the end.
  - If the return value is equal to the buffer size, then your buffer was too short, and the buffer may contain a truncated string.
  - If the return value was negative, then an error occurred.

In the assembly version, the negative error code would land directly in `rax`. This is why system calls return negative error codes, to distinguish them from successful return values as a successful call will never write a negative number of bytes.

However, the C wrapper is different as you can read in `man readlink`: it always returns -1 on error, but puts the error code (this time positive again) in a global variable called `errno`. You can match this against the codes in `errno.h` as before, and then check the manual page for an explanation of each one for this particular system call.

Exercise: write a C program that, starting with a filename you pass in `argv[1]`:

1. `lstat` s the file and prints the inode number and file type of the file.
2. If the file is a symbolic link, calls `readlink` to get the link target and repeats from 1. for the target file.

Since this is systems programming, make sure you check the return value of system calls, and correctly zero-terminate strings. If a system call fails, your program should print an

error message and exit, and *never* look at the buffer or string the system call wrote to, as it might not contain valid data.

Call your program for `/dev/stdin` and `/dev/stdout` to follow the chain of soft links for these files. Also try it on a few standard files and directories (including soft links).

On the command line

To check the results of your program, the `stat` command line program (`/bin/stat` , in fact yet another soft link to busybox) calls stat and prints the output to the terminal. You can see with `stat --help` that it offers lots of custom formats.

Note that the `stat` command line tool calls the `lstat` system call by default, e.g. it does not follow soft links. `stat -L` gets you the link-following version.

To see how the command line version works, we are going to have a look at its sources.

Clone the busybox repository with `git clone git://busybox.net/busybox.git` (if that doesn't work for some reason, try the https version). Inside the cloned folder, the source file is `coreutils/stat.c` - open that and have a look:

- The comments at the start are read by a custom build tool. The `//applet` line (currently line 38) says to build a command called `stat` .
- `file_type` (line 123 at the time of writing) is the code for turning mode bits into strings, note there are lots of `#ifdef` s depending on what options you compile busybox with. Also, if you `stat` a file that does not match any known type, you get the string "weird file".
- Most of the source file is the kind of "plumbing" that you need in any real C program. The interesting part is `do_stat` (line 588 at the time of writing):

First, it allocates a `struct stat` buffer like we did before.

The key line is currently line 605 and following:

```
if ((option_mask32 & OPT_DEREFERENCE ? stat : lstat) (filename, &statbuf) != 0)
{
    bb_perror_msg("can't stat '%s'", filename);
    return 0;
}
```

Based on the value of `OPT_DEREFERENCE` (the `-L` command line flag), we call either the `stat` or `lstat` system call wrapper in the C library, and complain and exit if we don't get a success return value - remember, in case of errors, `struct stat statbuf` could be corrupted so we shouldn't look at it.

The rest of the function is basically setting up one giant `printf` statement to output the results in the correct format. Note here that the `struct stat` still doesn't know anything about the file's name, as that's not in the inode, but the command-line program does because you gave the name as an argument.

If the file is a soft link, then we call a version of readlink - currently `xmalloc_readlink_or_warn` in line 713 - to display the link target. This function is implemented in `libbb/xreadlink.c` where it currently delegates to `xmalloc_readlink` on line 20, which calls `readlink` in a loop with increasing buffer sizes until it finds one that is big enough for the target - have a look at how this is implemented.

The main function for this utility is `stat_main`, currently on line 757. All this does is parse the command line arguments with `getopt32`, call the `do_stat` function in a loop for each command line argument (lines 787 and following in the current version) and then return success if all files were successfully processed, otherwise failure.

If nothing else, the learning point of this activity is that system programming in C and calling syscalls directly is a lot more involved than you may think! Please don't be that kind of programmer who ignores system call error values, and makes terrible things happen.

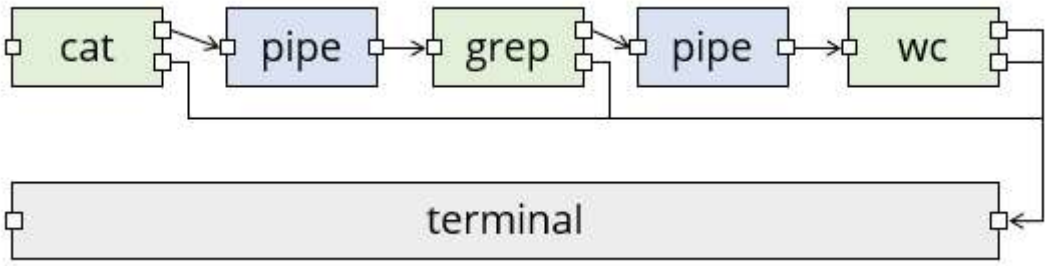
If you want to explore this code further, you can build your own busybox - install `ncurses` and `linux-headers`, then run `make menuconfig` to bring up a configuration menu (this is the part that uses ncurses, which is the menu system) and just select exit (with TAB then ENTER). Then you can `make` the whole thing.

Concurrent programming in POSIX

In the undergraduate degree at Bristol, students study concurrent programming using the go programming language, which provides an abstraction called *channels* that different *actors* can use to talk to each other. In the case of go, the actors are threads, but the same principle applies to a concurrent system with different processes, in which case channels are a kind of *inter-process communication (IPC)*. In POSIX, pipes are a kind of channel.

- If one process reads from a pipe, then this blocks the process until another process writes to it.
- If one process writes to a pipe, then this blocks the process until another process reads from it.
- If one process reads from a pipe and another process writes to it (it does not matter who goes first) then the data is transferred from the reader to the writer, and both processes continue.

For example, when you use a command like `cat logfile | grep Error | wc -l` to count the number of lines containing the word "Error", then there are three processes (not counting the terminal) and two pipes involved:



In this diagram, green boxes are processes with standard input on the left and standard output (top) and standard error (bottom) on the right. The little squares all represent file descriptors; a pipe has two of them, one each for reading and writing.

This is not yet a fully concurrent system though, as data is only flowing in one direction: there are no loops.

As a little example of something that does have a loop, we are going to use the `bc` calculator program, which (once installed) reads a line from standard input, evaluates it as a formula, and writes the result to standard output:

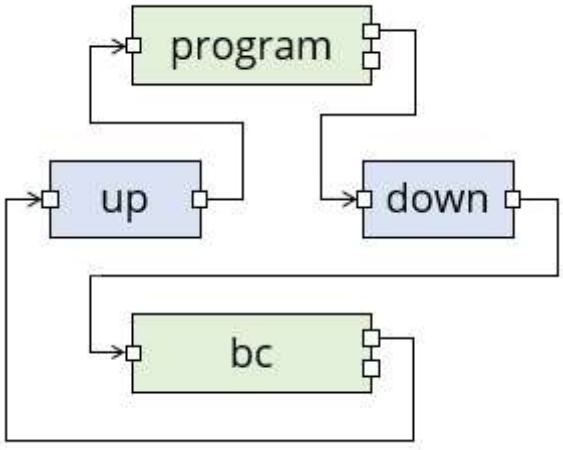
```
vagrant@debian12$ bc
(you type ) 2+3
(bc prints) 5
(you type ) 1+1
(bc prints) 2
```

`bc` reads in a loop, so type `^D` to close its standard input, then you get back to the terminal.

As a diagram, the interaction with bc looks like this:

Here the terminal's right file descriptor is the one that the terminal reads, and the left one is where the terminal writes your keyboard input. But there's still a human in the loop here.

We are going to write a program that can talk to `bc`. This means we have two processes, our program and `bc`, and they need to communicate with each other. We will use pipes for this, giving the following diagram where we call the pipes *up* and *down*:



In this diagram, data flows through pipes from left to right, but logically the "down" pipe is for data flowing from our program downwards to bc, and the "up" pipe is for data flowing back up again. Standard error is not shown (it still goes to the terminal).

The `bc` program does not need to know about any of this: it still reads expressions from its standard input, evaluates them, and writes them to standard output.

Notice that when we used a pipe character `|` in the shell, it automatically set up the pipes for us so each pipe had one reader and one writer. When we set the pipes up ourselves, we will have to take care of this ourselves too.

The first program

Study the following program. It tries to add the numbers from 1 to 10 by writing the sums (1+2 etc.) to standard output, and reading the result back from standard input. You can compile it with `gcc -std=c99 -Wall ipc1.c -o ipc1` and then run it yourself: it prints `1+2`

and if you reply `3`, it follows up with `3+3` and so on. After ten steps, it prints the result and exits.

```
/* ipc1.c */
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>

void check(int ok, char *where) {
  if (ok < 0) {
    fprintf(stderr, "Error in %s: %s\n", where, strerror(errno));
    exit(1);
  }
}

int evaluate(char *buffer, int len) {
  fprintf(stderr, "> %s", buffer);

  int ok = printf("%s", buffer);
  if (ok < 0) {
    return -1;
  }
  char* p = fgets(buffer, len, stdin);
  if (p == NULL) {
    return -1;
  }
  fprintf(stderr, "< %s", buffer);
  return 0;
}

int main() {
  char buffer[100];
  int ok;

  setbuf(stdout, NULL);
  setbuf(stdin, NULL);

  strcpy(buffer, "1+2\n");

  ok = evaluate(buffer, sizeof(buffer));
  check(ok, "I/O");

  for (int i = 3; i <= 10; i++) {
    sprintf(buffer + strlen(buffer) - 1, "+%u\n", i);
    ok = evaluate(buffer, sizeof(buffer));
    check(ok, "I/O");
  }
  fprintf(stderr, "The result is %s", buffer);
  return 0;
}
```

The program also prints a copy of all its input and output to standard error so you can see what data is being transferred. The transfer happens in `evaluate`:

1. First, it prints the buffer to standard error.
2. Then, it prints the buffer to standard output with printf. Checking the return value of printf is extremely paranoid, but we're about to use the program in a situation where standard output is not the terminal, so it could potentially fail.
3. Next, it reads a line from standard input with fgets. Checking the return value here is good practice even if you're not paranoid.
4. Finally, it prints the received value to standard error - note that if `fgets` had returned NULL, then it would not be safe to access the buffer.

Once you are familiar with how the program works, you can comment out the lines that print to standard error if you like and run it again to see exactly what happens on standard input/output and nothing else. Then, uncomment the lines again.

Advanced note

You could also use another trick to disambiguate standard output/error (for programs that don't, like this one, indicate which is which: the lines starting `>` or `<` are standard error).

```
escape=$(printf '\033')
./ipc 2> >(sed -e "s/\(.*\)/${escape}[32m\1${escape}[0m/")
```

This might mess up the last line on your terminal a bit, but you can reset it with Control+L. The first command sets a shell variable to the ANSI escape character. The second line redirects standard error (`2>`) to a subprocess (`>(...)`) which calls the stream editor `sed` to replace each line with the line surrounded by `\e[32m` (set colour to green) and `\e[0m` (reset colour). This will make the standard error lines appear in green.

Incidentally, some versions of sed support the `\e` escape sequence directly, or at least the version `\x1b` that creates a character from its ASCII code in hexadecimal, but some versions do not, so you need the shell variable trick with a fall back to octal (033) notation!

And now for the interesting part:

- Make two named pipes (FIFOs) with `mkfifo up` and `mkfifo down` .
- You will need two terminals open for the following. Either ssh into your Debian box a second time or, better still, use tmux (`Control+B, %` opens a second terminal beside the first one; you can use `Control+B, Left` and `Control+B, Right` to switch between them).
- Run the program with `./ipc1 > down < up` to redirect standard input and output to the pipes. This blocks (exercise: which statement is it currently blocking on?).
- In the second terminal, run `bc < down > up` .

Both processes should now terminate, and although you won't see the pipes directly, standard error of your program is printing a copy to the terminal, where `>` is data sent to down pipe and `<` is data received on the up pipe:

```
> 1+2
< 3
> 3+3
< 6
> 6+4
< 10
...
> 45+10
< 55
The result is 55
```

Note that the program printed the "The result is ..." line to standard error too, otherwise you would not see it here.

Pipes in C

Types of inter-process communication

You can use any of the following for one process to talk to another:

- Pipes in the shell, e.g. `cat FILE | grep STRING`. We are going to learn how to manage these pipes ourselves from a C program.
- Named pipes (FIFOs), which are pipes that have filenames.
- `popen`, a C function that launches a new process with a pipe either for reading or writing, but not both.
- TTYs, as explained in the lectures. The C interface for using TTYs starts with `posix_openpt()` and is fairly complicated.
- Sockets. This includes both network sockets (TCP) and also UNIX sockets, which are another type of special file. Unlike pipes, sockets provide bidirectional communication.

We are going to be using pairs of pipes, since a single pipe only works in one direction.

C functions

We will need the following functions for our task, all from `unistd.h`:

- `int pipe(int fd[2])` creates a pipe. It takes an array of two integers and creates two file descriptors in them, one for the reading and one for the writing end of the pipe. Details in `man 2 pipe`, and like all system calls, you need to check the return value and look at `errno` if it is negative otherwise bad things *will* happen.
- `int dup2(int oldfd, int newfd)` changes `newfd` to point to the same file descriptor as `oldfd`. This is the system call version of the shell redirect.
- `pid_t fork()` creates a copy of the current process, and is the starting point for launching another process. Once you are sure that it has not returned an error, then the child process will see return value 0 and the parent process will see a return value >0, which is the process id of the child process.
- `int execve(const char *path, char *const argv[], char *const envp[])` replaces the current process by the indicated process, this is the C version of typing a shell command except that (1) the shell does fork-then-exec and (2) there is no shell involved, so you cannot use shell features like `*` expansion or builtin commands like cd here.

Here is the basic way to launch a program from another, keeping the original program running:

```
/* launch.c */
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

char* command = "/bin/echo";
char* argv[] = {"echo", "Hello", NULL};
char* envp[] = {NULL};
/* both argv and envp must be NULL-terminated arrays,
   also argv[0] has to be the program name (busybox cares about this)
 */

int main() {
    int ok = fork();
    if (ok < 0) {
        printf("fork() failed: %s\n", strerror(errno));
        return 1;
    }
    /* ok, fork succeeded and the following code will now be running TWICE */
    if (ok == 0) {
        /* This is the child process */
        ok = execve(command, argv, envp);
        if (ok < 0) {
            printf("execve() failed: %s\n", strerror(errno));
            return 1;
        }
        /* we will never get here as if execve succeeded, we're gone */
    }

    /* if we got here, then we're the parent process */
    printf("Launched a child process, it has pid %u.\n", ok);

    return 0;
}
```

If you run this several times in a row, you might see the PID of the child increasing by 2 each time, why is this?

And here is the basic way to work with pipes:

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

typedef int fd;
typedef fd Pipe[2];
fd Reader(Pipe p) { return p[0]; }
fd Writer(Pipe p) { return p[1]; }

void check(int ok, char *where) {
  if (ok < 0) {
    fprintf(stderr, "Error in %s: %s\n", where, strerror(errno));
    exit(1);
  }
}

int main() {
  int ok;
  Pipe p;
  ok = pipe(p);            check(ok, "opening pipe");
  ok = close(Reader(p));   check(ok, "closing reader");
  /* here we can write to p */
  ok = close(Writer(p));   check(ok, "closing writer");
  return 0;
}
```

Let's go through this step by step.

- A file descriptor is simply an integer. so we make a typedef for this.
- A pipe is a pair of file descriptors for reading and writing, implemented as an array of length 2. The second typedef is C syntax for defining `Pipe` with a capital P to be `fd[2]` We use a capital P because lowercase `pipe` is already in use for the function that sets up a pipe.
- The functions `Reader` and `Writer` are just for convenience.
- In main, we declare a variable of type `Pipe` and open it with `pipe()`. Like all POSIX functions this returns a negative number in case of errors, and we have to check for this: it's not safe to use the pipe if it was not opened correctly. The pipe does not need a name as it's local to our program, but you can print the value of the integers if you like, it should be something like (3, 4) as 0-2 are already in use for standard input, output and error.
- In this example we want to use a pipe for writing to, so we close the reading end first. This is important when sharing a pipe between two processes: only one process should have each end open. (There are scenarios where you might want both ends of a pipe open in the same process, but they are more advanced than what we are doing here.)
- In the line where the comment is, we can write to the pipe - you will see how soon.
- Finally, before returning, we close the writing end of the pipe to ensure the process on the other end knows we're done: if we don't do this, they could get a "broken pipe" (EPIPE) error.

We still need to learn how to write to a file descriptor, though. And this pipe won't do anything useful until we can connect something to the other end.

Advanced note

If you look at the pattern for checking return values

```
ok = pipe(p); check(ok, "create pipe");
```

you might be wondering why we don't just combine it into one:

```
check(pipe(p), "create pipe");
```

This is absolutely fine except if we want to debug the function being called, as it makes using the debugger's step features more complicated (more on this in future weeks of this unit). It's just a personal preference of mine not to combine the two.

Of course, if you want to use line-based features of debuggers like breakpoints a lot, you might even want to put the check on a separate line.

You might have wondered if the problem with the all-in-one pattern is that it "swallows" the return value. That is not a problem: we could adapt `check` to return its first parameter as the return value if it's not an error, then where you do need the value you can do things like

```
int pid = check(fork(), "trying to fork");
if (pid > 0 ) { /* parent */ }
else { /* child */ }
```

This is one of several patterns for error handling you will see in C, although this particular one is commonly implemented as a macro so it can also print the file and line where the error occurred:

```
#define check(ok, where) _check(ok, where, __FILE__, __LINE__)
int _check(int ok, char *where, char *file, int line) {
  if (ok < 0) {
    fprintf(stderr, "Error at %s on line %i while %s: %s\n",
            file, line, where, strerror(errno));
    exit(1);
  }
  return ok;
}
```

Since C99 you can also use `__func__` to get the name of the current function.

You will find various patterns like this a lot if you start looking into real C code bases. Of course, most real programs will try to handle an error where possible rather than just exit the whole program.

Input/Output in C

Reminder: C file handling

You should already be familiar with the following C functions from `<stdio.h>`, though you can look up details with `man 3 FUNCNAME` on a lab machine:

- `FILE *fopen(const char *path, const char *mode)` opens a file, returning a file handle if successful and NULL otherwise. On some systems, it makes a difference whether you read a file in text mode (`r`) or binary mode (`rb`), the difference being that the former activates an extra "line discipline" in the C library. For example on Windows, text mode translates a lone `\n` character to `\r\n`. *On POSIX systems, there is no difference between text and binary mode.*
- `int fclose(FILE *file)` closes a file, and flushes any pending writes. It returns 0 on success and nonzero in case of errors.
- `int fflush(FILE *file)` flushes a file: for output, it makes sure any buffered characters are written out. For input, it throws away any unread input in the c library input buffer. It returns 0 on success and nonzero in case of errors. `fflush(NULL)` flushes all open files.
- `int feof(FILE *file)` returns nonzero if the file in question has reached end-of-file, and zero otherwise. This is the only correct way to test for an end-of file condition.
- `int ferror(FILE *file)` returns nonzero if the file in question is in an error state. For example after a `fread` that did not return the expected number of blocks, exactly one of `feof` and `ferror` will return nonzero, indicating what happened. In case of an error, the global variable `errno` will contain the error code.
- `size_t fread(void *dest, size_t size, size_t num, FILE *file)` reads up to `num` blocks of `size` bytes into the buffer at `dest` and returns the number of blocks read (for a block size of 1, this is the number of bytes). If the return value is less than `num` then either an error occurred or the end of the file was reached.
- `size_t fwrite(const void *buffer, size_t size, size_t num, FILE *file)` writes up to `num` blocks of `size` bytes from the `buffer` to the `file` and returns the number of blocks written. If this is less than `num`, then an error occurred during the writing.

A number of utility functions build upon these:

- `int fprintf(FILE *stream, const char *format, ...)` writes formatted output to a file, usually by calling `fwrite` behind the scenes. `printf` is a shortcut for `fprintf(stdout, ...)`. These functions to not print a newline unless you ask for one. It returns the number of characters written, or a negative number if an error occurred.
- `int fputs(const char *string, FILE *file)` writes a zero-terminated string to the file, not including the zero terminator. `puts` is a shortcut that writes to `stdout` *and*

adds a newline at the end.

- `int fputc(int c, FILE *file)` writes a single character (`c` is cast to an `unsigned char` first) and `putchar` is a shortcut for writing to `stdout` (but does not add a newline, unlike `puts`).
- `int fgetc(FILE *file)` reads a single character from a file (but returns it as an int, however it's safe to cast to char). `getchar` is a shortcut that reads from standard input.
- `char *fgets(char *buffer, int size, FILE *file)` reads up to `size-1` bytes into the buffer, stopping early if it finds a null byte, newline or end-of-file character. It then adds a zero-terminator to the string. If it stopped early because of a newline, the newline is included in the buffer; if it stopped due to an end-of-file then this is not included. Older versions of C included a `gets` version that reads from standard input and does not take a size argument; this is insecure as it can produce a buffer overflow and it should never be used. `fgets` is safe if the length of the buffer is at least `size` bytes. It returns a pointer to buffer on success and NULL if an error occurred. End-of-file counts as an error for this purpose if no characters could be read at all.

Fread (and fwrite) can return in one of three different states:

1. The return value is equal to the number of items you asked to read/write (third argument). This means that the read/write was successful.
2. The return value is not equal to the number of items, `ferror` returns nonzero on the file: an error occurred. The return value indicates how many items were successfully read or written before the error occurred. `errno` contains more information about what happened.
3. The return value is not equal to the number of items, `ferror` returns zero on the file: end of file. Calling `feof` on the file will return nonzero. End of file when reading means exactly what it says; end of file when writing to something with a program on the other side (pipe, socket, pty) means the other side has closed the connection.

Exercises

For our next exercise, we investigate how the C library file functions interact with the terminal. Compile this program:

```c
// program: input1.c //
#include <stdio.h>
#include <string.h>
#include <errno.h>

// Utility function to print an error message and return from main,
// use: return error(code, text); // in main
// prints text and the current errno value, then returns code.
int error(int ret, char* text) {
  int e = errno;
  printf("Error %s: %s (code %i).\n", text, strerror(e), e);
  return ret;
}

int main(int argc, char **argv) {
    if (argc < 2) { printf("Use: %s FILENAME\n", argv[0]); return 1; }
    printf("Opening [%s]\n", argv[1]);
    FILE *file = fopen(argv[1], "r");
    if (file == NULL) { return error(2, "opening file"); }

    char c, d;
    size_t result = fread(&c, 1, 1, file);
    if (result < 1) {
      if (ferror(file)) {
        return error(2, "reading first character");
      } else {
        puts("No first character - end of file?");
        return 2;
      }
    }
    printf("Read character: [%c].\n", c);
    result = fread(&d, 1, 1, file);
    if (result < 1) {
      if (ferror(file)) {
        return error(2, "reading second character");
      } else {
        puts("No second character - end of file?");
        return 2;
      }
    }
    printf("Read another character: [%c].\n", d);

    int e = fclose(file);
    if (e) { return error(2, "closing file."); }
    return 0;
}
```

The essence of the program are the four lines

```c
fread(&c, 1, 1, file);
printf(..., c);
fread(&d, 1, 1, file);
printf(..., d);
```

which read two characters from the file indicated in its first argument, then print them out. Note that we print the first character before reading the second, as this will become relevant.

The rest of the program is error handilng, and although some tutorials leave this off to make the code look easier, this sets you up for terrible habits - in the real world, errors happen and you need to handle them properly. So this program also shows the absolute minimum of error handling when you work with a file.

- Make a file `file` containing some characters such as `abc` and run `./input1 file`. Convince yourself that the program prints the first two characters (first two bytes, to be precise).
- Next, run `./input1 /dev/stdin`, to make it read from standard input. Notice that it blocks. Type a few characters such as `abc` and notice that the program prints nothing, until you press ENTER at which point both the "Read character" and "Read another character" output appears at once. **What is the reason for this?**
- Restart the program, and when it is waiting for input, type `ab[BACKSPACE]c[ENTER]`. **Explain what you see.**
- What happens if you enter only one character and then press ENTER? What if you press ENTER and haven't typed anything at all?
- If you want, verify that using fgetc / getchar produces the same behaviour on standard input: the program does not proceed past the first read command until you press ENTER. You can even try debugging with gdb to make sure.
- Insert the line `setbuf(file, NULL);` just after the `fopen` line to make sure it's not the C library buffering.

Have you answered the questions in bold above before reading on?

- Next, on your terminal, execute `stty -icanon`, which turns off the "icanon" setting. (The convention for `stty` is that an option name in an argument turns it on, except if prefixed with a minus which turns it off.)
- Rerun the program (`./input1 /dev/stdin`) and see what happens now.
- Turn "icanon" back on with `stty icanon`.
- Research what `icanon` does, for example in `man stty` on a lab machine or online.
- **Why does BACKSPACE and `^U` work in your bash shell, even with icanon turned off?**

Next, we are going to experiment with the terminal in fully raw mode:

- `stty -g` prints the current terminal setting in a format that you can save and load again. Take a look, then store them with the shell command `state=$(stty -g)`.
- Execute `stty raw`, then run your program again. **The terminal looks a bit "messed up" - can you tell what is happening?** Try another command like `ls` in raw mode too if that helps.
- Restore your terminal settings with `stty $state`, reading them back from the variable where you stored them. You can then reset the window with `^L`.

There is an important warning here if you are ever benchmarking a program that produces (standard) output - the total running time will be the sum of the program's running time and the connected terminal's running time to process the output. For example, printing a sequence of newline characters is typically slower than printing a sequence of 'a's, even if both programs are doing `putc` in a loop, because the terminal has extra work to do on each newline.

Here is another C program to experiment with:

```
// input2.c //
// Reads characters in a loop and prints their hex codes.
// Quit with 'Q'.

#include <stdio.h>

int main() {
  unsigned char c;
  int r;
  setbuf(stdout, NULL);
  while(1) {
    r = fread(&c, 1, 1, stdin);
    if (r < 1) break; // bail out on error or eof
    if (c == 'Q') break;
    printf("(%02X) ", c);
  }
  return 0;
}
```

- What do you expect to happen if you run this in a "normal" terminal and type a few characters?
- Run it in a terminal with icanon turned off and type a few characters. Notice that you can still cancel it with `^C`.
- Run it in a terminal in raw mode and type a few characters. What happens now when you press `^C`?

Luckily, we built in another way to quit the program with Q!

- In raw mode, try a few "special" keys such as ENTER, ESC, the arrow keys, function keys (F1, F2) etc.
- We turned off buffering for standard *output*. Why is this important - what happens otherwise and who is doing the buffering?

What actually happens when you press Control-C in a terminal in "cooked" mode is it sends the signal `SIGINT` (interrupt) to the connected program - the default behaviour for this signal is to terminate the program. Putting the terminal in raw mode just passes the control code for Control-C on to the program, though it would still quit if you sent it a SIGINT another way (e.g. with the `kill` command from another terminal). However, you could also write a signal handler in your own program that reacts to SIGINT and does something else, for example just ignores the signal.

When you type say an `a`, even in raw mode, the character appears on your screen even though there's no print command for it. This is the terminal echo; you can turn even this off with `stty -echo` but you are now typing blind! You can still start your program with `./input2` followed by ENTER, and it will still print the hex codes of everything you type from then on until you press Q but it will no longer show you the characters themselves. If you saved the terminal state before, then typing `stty $state` and ENTER will get you your echo back.

Input/output in POSIX

Both the C library and the POSIX standard library contain file I/O functions. Usually, the C library ones build on the POSIX ones if you are on a POSIX system.

The POSIX I/O functions are declared in `fcntl.h` and `unistd.h`. The main ones are:

Open and Close

- `int open(const char *pathname, int flags, [mode_t mode])` opens a file and returns a file descriptor. The flags are the boolean OR (`|`) of one or more of the following:
  - `O_CREAT` : if the file does not exist, create it. In this case the *optional* mode parameter can be used to specify the permissions for the newly created file. If the `O_CREAT` flag is omitted and the file does not exist, `open` returns an error.
  - `O_EXCL` : return an error if the file already exists (only really useful together with `O_CREAT`).
  - `O_PATH` : don't really open the file, just make a file descriptor (that you can use to e.g. `stat` the file). In this case you do not need one of the three flags mentioned in the next point.
  - `O_RDONLY` , `O_WRONLY` , `O_RDWR` : exactly one of these mutually exclusive flags must be set to indicate whether you want read, write or both access modes on the file.
  - `O_TRUNC` : if the file already exists, then its previous contents are overwritten if you open it to write.
  - `O_NONBLOCK` : this is the most interesting option and the main reason why you might want to use `open` over `fopen` . More details on this in the next activity.
  - There are a lot more flags that you can see with `man 2 open` .
- `int close(int fd)` : close a file descriptor. Returns 0 on success and -1 (setting errno) on error.

You might wonder how a C function call can have an *optional* parameter. The function is actually declared in `fcntl.h` as `int open(const char*, ...)` where the ellipsis means *any number of parameters of any type* just like `printf` uses.

If `open()` fails, it returns a negative number that indicates the error code: see `errno.h` for the names of the codes and the manual page (`man 2 open`) for what each one means for this function.

(Non)blocking IO

When you read from a regular file, there is a short delay while the kernel does the actual reading for you - possibly it has to fetch the data from disk. This is not what is meant by blocking. However, when you read from a pipe (e.g. you call `a | b` in the shell and `b` reads from standard input) or from a named pipe (FIFO), then if no process is connected to the writing end, your system call blocks until someone writes to the pipe.

The Posix functions offer an option `O_NONBLOCK` that makes a call fail instead of block. This is one of the reasons that you might use `read` directly rather than `fread`. While this is an important topic for concurrent programming (and especially network programming), that can wait until second year.

Read and Write

- `ssize_t read(int fd, void *buf, size_t count)` : read up to `count` bytes from the file descriptor `fd` into the buffer. In case of an error, this returns -1 and puts the error code in `errno`. If the end of file is reached, then the return value might be less than count (possibly zero) but this does not count as an error.
- `ssize_t write(int fd, const void *buf, size_t count)` : the same but for writing to a file (descriptor).

Before you use any of these functions, there are two warnings. The small one is that unlike the C library which provides *buffered* input/output, the POSIX functions do not - so while it's ok to read a large file one character at a time with `fread` due to buffering, this becomes really inefficient if you use the basic `read`.

The big warning is that checking return values is even more important than usual because these functions can return two kinds of errors: fatal ones (e.g. file not found) and temporary ones, which basically means *try again*, so in some cases the correct pattern to use these functions is to call them in a loop that repeats until the function either succeeds or returns a fatal error. Specifically, `read()` can return any of the following:

- -1, `errno = EINTR` : the call was interrupted, try again.
- -1, `errno = EAGAIN` or `errno = EWOULDBLOCK` : you have requested non-blocking IO, and the call would block.
- -1, any other errno: a fatal error occurred and `errno` gives more information. You must not retry the call.

Exercise

In the sources for busybox (`git clone git://busybox.net/busybox.git`), use your shell skills to find the source file and line where the function `safe_read` is defined, and study the pattern used there to operate `read` in a loop.

The final challenge

The final challenge for this part of the unit is to write a C program that uses pipes to interact with bc to add the numbers from 1 to 10, as an example of concurrency and inter-process communication in action. For this, you will need to put together items from the previous pages for this activity.

The general ideas are:

1. Create two pipe variables called `up` and `down`.
2. Fork the process. This shares the pipes between both copies of the process.
3. The parent process will write to the down pipe and read from the up pipe, so it should close the down reader and the up writer.
4. The child process closes the down writer and the up reader, redirects its standard input to the down reader and its standard output to the up writer, and then execs `bc`. This keeps the redirected standard input and output.
5. The parent process now writes its sums starting with `1+2` to the down writer with `write` and then uses `read` on the up reader to get the result. You can use a function `evaluate()` like in the previous example that handles the reading/writing and prints debug information to standard output (we don't need standard error as the pipe is a separate file descriptor).
6. The parent cleanly closes the pipes when it is done, and writes the result to standard output.

Two constraints here:

- All calls to Posix or file I/O functions *must* be checked for error return values. It is ok to terminate the program in case of fatal errors rather than try and fix the cause, but not to just carry on ignoring the return value. You can reuse the `check()` function for this.
- Calls to `read` and `write` must be assumed to be able to fail with `EINTR`, in which case they need to be retried, so these functions must be called from within loops as you saw in the busybox sources.

The one new function you need to know is `int dup2(int src, int dest)` which redirects the destination to be a duplicate of the source. For example if `dr` is a file decriptor for the down reader, then `ok = dup2(dr, 0)` redirects standard input (file descriptor 0) to read from the down reader. Of course, you have to check the return value `ok`!

Advanced note

One way of using `dup2` is to redirect your own standard output to a file, then any calls to `printf` or other functions from then on should end up in the file. If you later on want to restore the original standard output, you can first call `int dup(int fd)` which makes a copy of the file descriptor - for example, if a file is open on file descriptor 3, then

```
saved = dup(1);       check(saved, "duplicating standard output");
ok = dup2(3, 1);      check(ok, "redirecting standard output");
/* any printf etc here will go to the file on fd 3 */
ok = dup2(saved, 1); check(ok, "restoring standard output");
ok = close(saved);    check(ok, "closing duplicate");
```

shows how you can restore the original standard output by making a copy. If no other file descriptors are open, then `saved` will likely have a value of 4, as file descriptors are just numbers to communicate with the kernel and the kernel usually gives you the next consecutive unused one when you open or duplicate a file descriptor.