HIGH PERFORMANCE PROGRAMMING
UPPSALA UNIVERSITY
SPRING 2017
LAB 1: LINUX & C BASICS

The aim of this lab is to familiarise students with the Linux/Unix environment, the build process, and those aspects of C that are the most usual source of confusion and bugs. Before the next lab, you should feel comfortable executing the tasks in this lab, understand the steps of the build process, and have some experience in understanding and manipulating C code.

Before you start! Search on the web for a Linux/Unix quick reference. There are many to choose from — find one you like!

1. Getting Started

Log into a system and download the lab tar-ball `Lab01_Linux.tar.gz` from Studentportalen. Save it to your desktop.

Open a terminal. You'll see a bash prompt (a.k.a. shell) like

username@hostname:~ $

where `username` is your username and `hostname` is the name of the computer you are logged in to.

You can type commands at the prompt, and bash will attempt to execute them. It does this by looking through your PATH until it finds the right program. PATH is an *environment variable* that lets the system find the things you need, no matter where they might be found.

Important: Linux is case-sensitive. That means that there's a difference between PATH, Path, and path.

Type `echo HEJ hej` to see how the `echo` command works — simply echoing back the thing you gave it as input.

Type `echo $PATH` to see what PATH expands into — a colon-delimited list of directories.

Type `which echo` to see where the `echo` command is located.

A single dot, "`.`", denotes the current working directory, where you're located right now (which should be your home directory). Similarly, "`..`" denotes the parent directory.

Type `pwd` to see your current working directory.

Date: January 17, 2017.

Type `ls` to see the contents of your current working directory.

Type `cd ..` to change directory to the parent directory.

Type `cd -` to return to the previous directory.

Create a directory for the course: Type `mkdir HPP` and cd into it (`cd HPP`).

Create a directory for this lab, cd into it.

Move the lab file to this directory (note that `~` always denotes your home directory):

```
mv ~/Desktop/Lab01_Linux.tar.gz ./
```

Decompress the tar-ball with the following command:

`tar -xvzf L<TAB>`, where `<TAB>` is the "tab" button. After you hit TAB, the command should read `tar -xvzf Lab01_Linux.tar.gz`.

That's right, hitting the TAB button autocompletes directories and filenames, so you don't have to type everything and, *even more importantly*, avoid typos. *Always use tab completion as you navigate directories and write file names. This will save your life!*

`tar` is an archiving program. The cryptic-looking string of letters `-xvzf` are known as *flags*. Flags control what the program is going to do.

Use the command `man tar` to look at the manual page for tar and read about what each flag does. Note that 'q' quits the man, space scrolls down a page, and typing e.g. `/Word` searches for 'Word' in the text. You can also use arrow keys to scroll up/down, or pageup/pagedown keys to scroll a whole page at a time.

Many programs also take the `--help` option, which usually prints out a shorter, more understandable summary of the command than man.

The `tar` command can also be used to create a tar-ball (this is how the `Lab01_Linux.tar.gz` file was created). Use `man tar` to figure out how this is done, and create your own tar-ball package from some files or directories. Hint: the `-c` and `-f` options can be useful (and `-z` if you want to use gzip compression).

The `tar` command will be useful later when you are going to submit code and/or reports for the assignments in this course. Then you will be asked to package your submission into a single .tar.gz and submit that in the Student Portal.

By convention, a tar file (a.k.a. tarball) created without compression is given the extension .tar while a file created without compression is given the extension .tar.gz or .tgz.

2. The build process

The process of transforming a written program into an executable in Linux/Unix takes 4 distinct steps, see Figure 2.1. For convenience, some or all of these steps can be performed with a single command. First, we will look at how to compile a simple program with a single command, and then we will look at each build step individually.

Task 1: Cd into the Task-1 directory. Compile the first program with the following command:

```
gcc -o first first.c
```

Here, we are using the gcc compiler. The "-o first" flag tells the compiler to call the output file "first". The compiler then recognises the .c file ending, denoting a C source file, which it puts through the entire build process.

You can run the program by typing `./first`.

Now compile again giving another name, e.g. "-o haha" instead of "-o first". Run the new "haha" and check that it works in the same way as "first". You can also skip the -o flag and simply compile like this:

```
gcc first.c
```

What happens then? Use `ls` to see if any file was created. Can you run it? (The somewhat boring name a.out is usually the default executable name, if -o is not specified.)

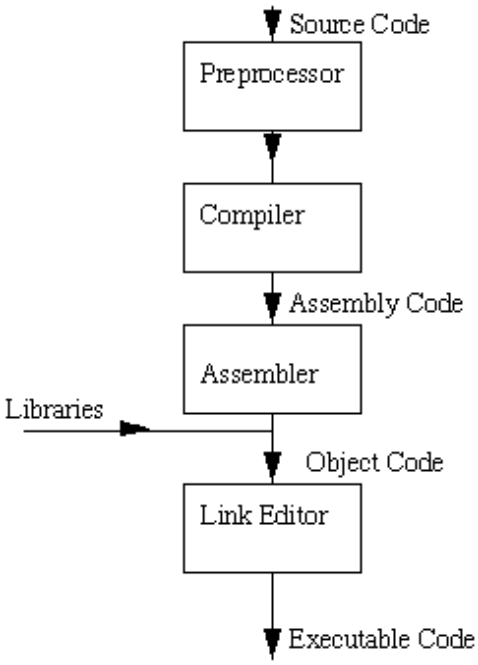

FIGURE 2.1. An overview of the build process. The "Link Editor" that converts object code to executable code is often simply called a "Linker"

2.1. **Preprocessing.** The preprocessor takes your C code as input and generates another C code as output. Preprocessing is used to combine different source files together, introduce code from an external source (e.g. a library), and to customise a code so it suits system or user requirements.

A list of common preprocessor directives:

- `#include` inserts code into a file. Typically, a C program code is separated into two sets of files. Header files (ending in .h) contain function prototypes, and source files (ending in .c) contain the function definitions (more on function prototypes and definitions in Section 2.2). In first.c, the include directive is used to include the function definitions from the standard library for I/O (input/output).
- `#define` defines a macro. Macros can be used to store predefined constants like "`#define PI 3.14`" or short code snippets like "`#define SQR(x) ((x)*(x))`". The preprocessor will then look for PI or SQR(...) and replace those tokens with 3.14 or ((...)*(...)), respectively.
- `#if, #ifdef, #ifndef, #endif` are used to add or remove code depending either on a boolean condition or whether a keyword is defined.

Task 2: Cd into the Task-2 directory, type `gcc -E -o preprocess.i preprocess.c` in the terminal and compare the output file with the .c and .h files.

Useful predefined macros, handy for debug prints or to annotate output:

- `__LINE__` Line of the current file
- `__FILE__` Name of the current file
- `__DATE__` Current date
- `__TIME__` Current time

2.2. **Compilation.** The compiler takes C code as input and generates assembly code as output.

If we want, we can examine the assembly code to see what the program is *actually* doing. For example, this can be useful if we suspect that the compiler has a bug in it. Later in the course, being able to look at the assembly code will be useful when we want to check if and how the compiler has optimized the code.

You can ask gcc to generate assembly code with `gcc -S *.c`, and it will output a .s file for every .c file.

The compiler does the bulk of the work when building a program. The compiler therefore gives you a wealth of options to tweak its behaviour, a few of which are listed below. Read the `man` page or the option summary online at gcc.gnu.org/onlinedocs/gcc/Option-Summary.html for more details.

2.2.1. *Useful basic compiler options, warnings, etc.*

- `-Imyincludedir` Add myincludedir to search path for include files.
- `-Wall` Turn on all warnings. Make this one a habit!
- `-g` Produce debugging information that a debugger (like `gdb`) can use.
- `-O` Optimisation settings.
  - `-O0` Turn off all optimisation (default). Use this when debugging!

- `-O` or `-O1` Turn on optimisations to reduce code size and execution time that don't take much compilation time.
- `-O2` Optimize even more. Perform nearly all optimisations that don't involve a space-speed tradeoff.
- `-O3` Turn on maximum optimisation. You'll experiment with these optimisation levels in the next lab.
- `-Os` Optimise for size.

2.3. **Assembly.** The assembler takes assembly code as input and generates object files as output. Object files are binary files that can either be linked with start-up code or wrapped into a library. Each source file will typically result in a .o file.

You can tell gcc to stop at the assembly step with the `-c` flag. This is commonly done in projects when you want explicit control over the linking step.

Task 3: Cd into the Task-3 directory and compile all the .c files with the command `gcc -c *.c`. Also build the `singleFile_calculator` executable.

The `nm` program can parse an object file, library, or executable and report on its contents. This is useful if you need to know what symbols are in a library or if you're having trouble linking object files produced by different compilers. The `nm` output contains one line for each symbol found in the object file.

Use `nm` to examine the contents of the `singleFile_calculator` executable and compare it to the contents of the .o files. You should be able to find the names of symbols and whether they are defined. Undefined symbols cannot be used unless a definition is linked in via a library or object file!

2.4. **Linking.** The linker combines object files and external libraries and generates an executable or library as output. There are two types of libraries: *static* and *dynamic*.

The static library is just an archive of .o files. You use it by linking it into your executable in the same way as your own .o files. The linker only loads the functions of the library that your code references. This moderately increases the size of the executable, but the running process will be small.

The dynamic library is prepared differently. When you link to a dynamic library, the linker just puts references in the executable, which results in a small executable file. When the running executable encounters function that is linked in from a dynamic library, the function loads dynamically (hence the name). The dynamic linker tends to load a large portion of the library, even if only a few functions are used, which increases the memory footprint of the running process.

We are going to create libraries for the add and subtract functions of the calculator program.

Create a static library from the .o files by writing

```
ar crs libstaticfunction.a add.o subtract.o
```

Now generate a new executable file using the static library and the calculator.o file:

```
gcc -o static_calculator calculator.o -L. -lstaticfunction
```

The `-L` flag tells the linker in which directory to look for libraries. The `-l`*name* flag tells the linker to link with static libraries with the name `libname.a` or dynamic libraries with the name `libname.so`.

Create a dynamic library with these commands:

```
gcc -o libdynfunction.so -shared add.o subtract.o
```

And link a corresponding executable:

```
gcc -o dyn_calculator calculator.o -L. -ldynfunction
```

If you get an error that the system cannot find libdynfunction.so when running `dyn_calculator`, you have to update the search path for dynamic libraries:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
```

Tip: Use `ldd dyn_calculator` to ask the executable which dynamic libraries it is linked to.

(note) – The intrepid student can read about performance issues of dynamic libraries in Section 14.11 in *Optimizing software in C++*.

Use `nm` to examine the structure of your executables and libraries.

3. Makefiles

When working with multiple compilation units (.c files), compiler flags, library paths, and so much more, it becomes necessary to make the build process repeatable and smooth. This is done with `make`. This program reads in and executes *makefiles*.

A makefile can contain a number of rules, defined like so:

```
target: dependency1 dependency2
<TAB>command-to-create-target
```

White space is important. There must be a TAB in front of the command. When `make` creates a target, it first looks at the specified dependencies. If the dependencies are files, then it only executes the command if any of those files have changed or the target is not present. If the dependencies are other targets, then it attempts to create those targets before continuing.

Task 4: Cd into Task-4 and type `make` to build the `sorting` executable. By default, `make` will run the file called `makefile` (or `Makefile`) and build the first target (and therefore also any other targets that are dependencies of the first target). Type `make clean` to delete all the .o files in preparation for the next step.

Read `makefile` and understand how it works.

The `-w` flag turns off warnings. This is not good. Instead we want to turn on all warnings (`-Wall`), but you'll notice that this makefile does not make this super-easy. Each time we want to change compiler flags we have to make changes to 3 or 4 lines of code.

The file `makefile-1` is more useful to us. By using variables to collect common elements in the commands, we can make changing the build process much more efficient.

Turn on all warnings by editing the `CFLAGS` variable in `makefile-1` .

Type `make -f makefile-1` to build the program again. (Do you get compiler warnings now?)

In practice, one usually does not write a makefile from scratch. We highly recommend that you save makefile-1 to an easy-to-remember directory and use it as a template in the future.

4. Debugging

Finding bugs in C programs in Linux without an IDE usually involves four things, presented roughly in order of decreasing importance (of course `-Wall` should be used as well):

(1) Good design. A careful program design with clearly defined and well documented/commented parts helps a lot.

(2) Using `assert` statements

(3) Using the `gdb` debugger

(4) Using `printf` statements

Task 5: Cd into the Task-5 directory. In this task, the program is supposed to add the diagonal elements of a square matrix.

Note: if you are new to C programming this task may be difficult. If so, don't worry, you will learn more about C programming during the next lecures and labs. Do your best now, and then you can go back and look at this task again after the C lectures.

The program named `trace.c` consist of four functions:

- `initialization` (initialize the square matrix)
- `fill_vector` (fill a vector with random numbers from +10 to -10)
- `print_matrix` (display the matrix)
- `trace` (sum the diagonal values and return the sum)

Your task is to debug the program.

Turn on warnings. Or don't, if you want more of a challenge.

Use assert statements of the form `assert( bug_condition && "Description of error")` to trap for bugs.

For example, if at a certain point in the code you want to verify that the variable q has a positive value, you could use a line like this:

```
assert( q > 0 && "checking that q is positive");
```

or simply

```
assert( q > 0 );
```

Keep in mind that asserts are used to help finding programming errors during development, *not* for error-checking in a final release version of a code, so they are not the way to handle issues like bad user input.

All assert statements can be disabled by defining a macro with the name `NDEBUG` at some point before including assert.h. So, for a final release version of a program you can add a line like "`#define NDEBUG`" before assert.h is included and then all assert statements will be skipped by the compiler. This is a convenient way of making sure the release version of the code will not be slowed down by any assert statements.

Use GDB to locate segmentation fault bugs: First, compile with the `-g` flag to save debug info within the executable. Then, initialize the debugger with the `trace` executable inside by typing `gdb ./trace`. Type `run` or just `r` to start the program. `q` will exit gdb.

If you want to follow the program's execution with print statements, you can distribute a few `printf("%s: %d\n",__FILE__,__LINE__)` in the code using copy-and-paste.

5. Monitoring a program while it is running

The `top` command can be useful for checking if a program is running and huw much CPU time it is using.

Task 6:

Cd into the Task-6 directory.

Compile and run the small test code there. Change the value of N in the code so that it takes $\sim$ 10-20 seconds to run.

Run it, and while it is running, run the `top` command in another terminal window. Do you see your program running? The "%CPU" column shows how much CPU time each process is currently using. A single-threaded program running by itself on one core would get 100% while a threaded program running e.g. 3 threads on 3 cores would get 300%. In `top`, use q to quit.

6. Extra part

In this task, you will practice using bitwise operators. The directory "Task-Extra" contains the file `bitwise.c`. This code defines two variables and prints out the corresponding bitstring of each, as well as the results of some bitwise operations. Your task is to write a function "`count_bits`" that counts the number of ones in a

variable of type `n_type`. Print the number of ones in variables `a` and `b` and of the products of the given operations.