Chapter 2 – Compiling and Debugging a C program

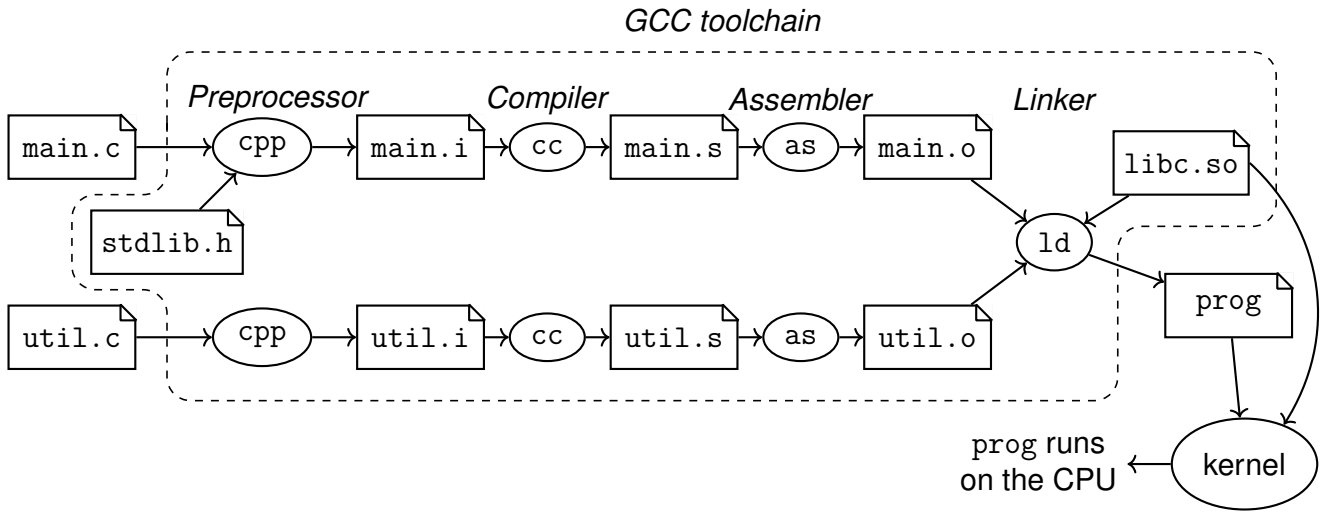
The role of the **processor** aka the CPU (e.g. Intel i9, ARM Cortex) is to execute machine language instructions, one after the other. But a modern computer typically runs many programs "at the same time" on the same CPU, a feature known as **multi-tasking**. How is this possible ?
Multi-tasking is an illusion, implemented by the operating system through time-sharing: at each instant the CPU is still executing instructions from a single program ; however many times per second, the system somehow switches from running one program to running another.[1] This paradigm is called **concurrent execution** and is completely transparent for applications: each "running program" (aka each **process**) is isolated from others and executes as if it had a private CPU.

But not all programs are processes. The **kernel** (e.g. Linux) is a special program which acts as a "orchestra conductor" for processes: its role is to load **executable files** from disk into memory, and to somehow have the CPU execute them all concurrently. The kernel also provides input-output services to applications: when a process wants to display something on the screen, write data to a file, or send a message over the internet, it actually can't do it "by itself", but rather it must delegate those actions to the kernel, by calling special functions we refer to as **system calls**.

As a programmer, you very often interact with the kernel through these system calls. As an end-user, however, you are more used to interact with an entire **operating system** (e.g. Ubuntu, Android), which includes not only a kernel, but also a whole collection of programs (graphical user interface, shell, etc) that work well enough together to make the machine usable by humans.
An important feature provided by the OS is the ability to **compile programs**, i.e. translate **source code** that is understandable by the programmer, to machine code (binary) that is understandable by the CPU. In this chapter, you will learn the ropes of using such a compiler toolchain.

1 The "GNU Compiler Collection" toolchain

There are several steps necessary to build an executable program from source files, but modern toolchains hide this complexity behind the scenes. The diagram below illustrates what happens when we type `gcc -o prog main.c util.c` to build executable `prog` from source files `main.c` and `util.c`, and then `./prog` to run the program.



[1] Details of the *context switching* mechanism are beyond the scope of this course, but you can read `https://en.wikipedia.org/wiki/Context_switch` if you want to learn more.

Exercise 1 Create a file named `hello.c` and type in the program below. Compile it with command `gcc -o hello hello.c`, then run it with `./hello`. Note: function `puts()` (i.e. "put string") writes some **character string** to the terminal, followed by a newline.

```c
#include <stdio.h>

int main() {
    puts("Hello, world");

    return 0;
}
```

Exercise 2 By default, GCC deletes all intermediate files (i.e. `.i`, `.s`, `.o`, etc) when they are not needed any more, so you won't be able to see them. But we can ask the toolchain to stop after a particular step:

- `gcc -E` runs the **preprocessor**[2], but does not compile.
- `gcc -S` runs the preprocessor and then compiles the result, but does not assemble.
- `gcc -c` preprocesses, compiles, and assembles, but does not run the linker.
- `gcc` (with no options) does all the steps and produces an executable.

Use these commands (with `-o filename`) to produce `hello.i`, `hello.s`, and `hello.o` respectively.

Exercise 3 Open `hello.i` in a text editor and browse the code until you locate the `main()` function. Everything above comes from file `/usr/include/stdio.h` (open this one too) and was copy-pasted here by the preprocessor. Notice that the comments from `stdio.h` have been removed. In `hello.c`, try and add some *`/* comments */`* anywhere and run the preprocessor again.

Exercise 4 So-called **object files** (like `hello.o`) store machine code as binary data, not as ascii-encoded text, so we cannot open them directly in a text editor. Type `xxd hello.o > hello.xd` to perform a "hex dump" then open the resulting text file in an editor. It will look more or less like this:

```
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0100 3e00 0100 0000 0000 0000 0000 0000  ..>.............
00000020: 0000 0000 0000 0000 5802 0000 0000 0000  ........X.......
00000030: 0000 0000 4000 0000 0000 4000 0e00 0d00  ....@.....@.....
00000040: f30f 1efa 5548 89e5 488d 0500 0000 0048  ....UH..H......H
00000050: 89c7 e800 0000 00b8 0000 0000 5dc3 6865  ............].he
00000060: 6c6c 6f2c 2077 6f72 6c64 0000 4743 433a  llo, world..GCC:
00000070: 2028 5562 756e 7475 2031 332e 332e 302d   (Ubuntu 13.3.0-
00000080: 3675 6275 6e74 7532 7e32 342e 3034 2920  6ubuntu2~24.04)
00000090: 3133 2e33 2e30 0000 0400 0000 1000 0000  13.3.0..........
000000a0: 0500 0000 474e 5500 0200 00c0 0400 0000  ....GNU.........
000000b0: 0300 0000 0000 0000 1400 0000 0000 0000  ................
```

The first column indicates the position in the file (address), the middle columns are the contents of the file in hexadecimal (two bytes per column) and the right column is the same contents but interpreted as ASCII. For instance, 0x45 0x4C 0x46 are the ASCII codes of letters "E", "L" and "F", which together indicate that this file is encoded in Executable and Loadable Format. Locate the *`"hello world"`* string in the hex dump, and write down the ASCII codes for each letter.
Open the tool's man page to look for help: type `man xxd`.

[2] `https://en.wikipedia.org/wiki/Preprocessor`

Exercise 5 What is the meaning of `00000000`, `00000010`, `00000020`, etc in the left-most column of the hex dump ?

Exercise 6 Reading hex dumps is very crude. Fortunately GCC offers the `objdump` tool, which purpose is to **disassemble** machine code into a readable listing. Type `objdump -D hello.o > hello.lst` then open the resulting file. It will look more or less like this:

```
hello.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # f <main+0xf>
   f:   48 89 c7                mov    %rax,%rdi
  12:   e8 00 00 00 00          call   17 <main+0x17>
  17:   b8 00 00 00 00          mov    $0x0,%eax
  1c:   5d                      pop    %rbp
  1d:   c3                      ret
```

- Notice how bytes are shown *grouped* into what `objdump` believes are machine instructions[3]:
  - 0x55 is the machine encoding for `push %rbp`
  - 0x48 0x89 0xe5 (three bytes) is the machine encoding for `mov %rsp,%rbp`
  - 0xF3 0x0F 0x1e 0xfa (four bytes) is the machine encoding for instruction `endbr64`, etc.
- Notice that some memory addresses (for instance, the destination address of the `call` instruction) are not known at this stage. They will be filled in by the linker.
- Notice how certain parts of the listing don't make sense as instructions: where are the bytes that encode our *`"Hello, world!"`* string ?

Exercise 7 Use GCC to produce the executable, then disassemble it with `objdump`. Notice how more code got included (which we will ignore for now) and how the `call` instruction now does have an explicit destination.

2 The GDB Debugger

One of the reasons programming in C is hard is because the language provides very few abstractions or guarantees against programming errors. In some respects, the C langage is little more than a structured syntax to write portable assembly. For this reason, it is often useful to look at a program at the machine code level, like we did with `objdump`. The GCC toolchain also offers the interactive debugger `gdb`, which allows to execute the program step-by-step and inspect memory contents interactively.

Note: GDB only works on Linux (including WSL) but not on macOS. If you are using a mac, you can learn to use `lldb`, the macos debugger, instead. The syntax is not exactly the same but the concepts are equivalent. Ask us for help ! Alternatively, you can skip this whole section entirely (after all, who needs a seatbelt when driving a car...) and move on to the next chapter.

[3] On Intel processors, instructions don't always occupy the same number of bytes

Exercise 8 Compile `hello.c` once again, this time with `gcc -g` so as to keep all **debug info**[4] in the executable: `gcc -g -o hello ./hello.c`. Then, launch the program from inside the debugger:

```
$ gdb -q ./hello
Reading symbols from ./hello...
(gdb)
```

Note that you are now in a new command-line tool (gdb). It accepts new commands (see below) and unix shell standard commands are not understood:

```
(gdb) ls
Undefined command: "ls".  Try "help".
```

Play with these gdb commands:

- `list` shows source code
- `help CMD` gives some help about a command (try it with all commands !)
- `disassemble /r FUNCNAME` displays a disassembled view of the code
- `break LINENUM` or `break FUNCNAME` inserts a **breakpoint** at specified location
- `start` adds a temporary breakpoint on `main` and executes until there
- `run` starts the program from the beginning
- `continue` lets the program run until it terminates or hits a breakpoint
- `help` prints a list of available commands

Exercise 9 Create a file named `fact.c` and type in the program below. Compile it with `gcc -g`.

```
#include <stdio.h>

int factorial(int n) {
    if (n<=1)
        return 1;

    return n * factorial(n-1);
}

int T[10];

int main() {
    for(int i=0; i<10; i++)
    {
        T[i] = factorial(i);
    }

    return 0;
}
```

Run this program inside gdb and play with these commands:

- `step` executes one line of source code, **stepping *into*** function calls
- `next` executes one line of source code, **stepping *over*** function calls
- `stepi`/`nexti` are similar to `step`/`next` but they work in terms of machine instructions
- `until` executes until the program reaches the following line in the source code (stepping over calls and loops)
- `where` displays the call stack
- `finish` executes until the current function returns

Advanced usage: `step N`, `next N`, `until LOCATION`. Use `help` to learn more about these commands, and read the documentation at `https://sourceware.org/gdb/current/onlinedocs/gdb.html/Continuing-and-Stepping.html`

[4] debug info = a two-way mapping between memory addresses in the executable and line numbers in the source file

Exercise 10 Make the program execute until the loop is finished and show the contents of the array with command `print T`.

Exercise 11 In the loop header, replace `i<10` with `i<100`. Compile and run the program, and observe that it does not crash, even though we now write well past the end of the array. Then repeat with `i<1000`. Somehow, it still runs "fine".
To understand why, we need to learn two things:

1. The C language is performance-oriented and not correctness-oriented. In other words, the toolchain makes no effort to detect incorrect programs. This responsibility falls entirely to the programmer.
2. The kernel provides each process with a virtualized, isolated execution environment which looks like a simplified von neumann machine (CPU+memory). As long as we stay within this sandbox, the kernel does not care about the fine-grained memory layout of our variables !

Because of these two properties combined, an incorrect program might well run (and produce wrong results) for a long time without being detected.

Exercise 12 Repeat the exercise with `i<10000`. Execution should now end in a **crash** with a "`Segmentation fault`" error message (otherwise, please ask us for advice). This is because we now go so far in memory that we exceed the space allocated to our process. The kernel detects this incorrect behaviour and reacts by killing the faulty process.
Execute the program in `gdb` to see what line of source code causes the crash. What is the value of `i` at that point ?

Exercise 13 (hard, optional) In gdb, use the `x` command to "examine" memory and observe values which have been produced before the crash.
This command is spelled `x/nfu ADDR` where `n`, `f`, and `u` arguments are optional.

- `ADDR` is a memory address or a symbol name
- `n` is the desired length, in number of values (default is 1 value)
- `u` is the size of one value: `b` (bytes), `h` (2 bytes), `w` (4 bytes, default) or `g` (8 bytes)
- `f` is the desired display format: `s` (text string), `i` (machine instructions) or `x` (hexadecimal, default)

For more info, type `help x` or read the GDB user manual at `https://sourceware.org/gdb/current/onlinedocs/gdb.html/Memory.html`
What is the "biggest" correct value in the resulting array ?