Lecture 21: Case Study: C Standard Library

COMP26020 Part 1 (C) Lecture Notes

Pierre Olivier

These notes summarise the important points mentioned in the lectures. They are supposed to be a help for revising and not a way to avoid attending the live lectures and watching the videos. In other words, live lectures and videos may include examinable content that is not present in these notes.

The slides for this lecture are available here:
https://olivierpierre.github.io/comp26020-lectures/21-libc-case-study.

Videos and recordings of live sessions can be found on the video portal: https://video.manchester.ac.uk/lectures.

The C Standard Library

The C standard library, also abbreviated libc, is the library providing `stdio.h`, `stdlib.h`, etc., that we have been using since the beginning of the course. It is written mostly in C, with a few bits of assembly. The default library coming with Linux distribution such as Ubuntu/Debian or Fedora is the GNU Libc, Glibc[1]. It is very complex and difficult to study, so in this le cture we will rather have a look at a simpler C standard library named Musl[2]. Musl is a production ready C standard library, and is widely used: for example the Alpine Linux container-oriented distribution ships with Musl as libc.

Using Musl

It is very easy to try out Musl. It can be downloaded, built, and used to build a small hello world C program as follows:

```
# Download musl
$ git clone https://github.com/ifduyue/musl.git

# Compile it and install it locally
$ cd musl
$ ./configure --prefix=$PWD/prefix
$ make
$ make install

# Write a hellow world and compile it against Musl rather than Glibc
# Don't forget the -static
$ ./prefix/bin/musl-gcc hello-world.c -o hello-world -static
$ ./hello-world
$ hello world!
```

In the rest of this document we take a brief look at how the libc works, and why it makes sense for it to be written in C. The main job of the C library is to interface the application with the operating system. To study this, let's see how Musl implements the function we have been using the most, `printf`. But before that, we need to understand how the C Library request services from the OS, through **system calls**.

[1] https://www.gnu.org/software/libc/
[2] https://www.musl-libc.org/

System Calls

Letting application access the hardware directly is too dangerous. So applications use system calls (abbreviated syscalls) to ask for services from the kernel, which is the only privileged entity that can directly manipulate the hardware to do things like reading and writing from/to disk, sending and receiving packets to/from the network card, printing to the console, etc. Syscalls are rarely made directly by the user code: the programmer rather calls functions from the libc, which in turn takes care of invoking syscalls. For example, to print to the console, the programmer calls `printf`, and if we look at its implementation within the libc code, we can see that it calls the `write` syscall (or a variant of it like `writev`). Therefore, the libc provides wrappers to the user code around system calls. Some wrappers have the same name as the system call in question such as `write`.

When a syscall is made, what happens under the hood is a bit more complex than a simple function call. There is a *world switch* of the CPU execution state, from user mode to privileged (kernel) mode. There is a particular convention on how this switch should happen, i.e. what are the machine instruction that user mode should issue to indicate to the kernel what syscall to run, and with what parameters.

The System Call Application Binary Interface

On Intel x86-64, a system call is realised with the following machine instructions:

1. First the syscall *number* is put in the `%rax` register. This number identify uniquely a given syscall, you can see the list of Linux syscalls and their numbers here: https://filippo.io/linux-syscall-table/.

- The syscall parameters are put in `%rsi`, `%rdx`, `%r10`, `%r8`, and `%r9` (in order).
- Then the "syscall" instruction should be executed, this triggers a trap to the kernel and the OS will process the syscall.
- When done the kernel place the syscall return value in `%rax`.

This is a convention put in place for communication between 2 entities: the program (that includes the libc) invoking the system call, and the kernel receiving the call and processing it. These 2 entities are compiled separately, hence for interacting they cannot use application programming interfaces (APIs), i.e. language level constructs such as functions, like one would normally do within a single program or between a program and a library it compiles against. They rather use directly machine instructions: it's an **application binary interface**, ABI.

Musl's Implementation of `printf`

`printf` is implemented in Musl's sources in the `src/stdio/printf.c` file. It calls `vfprintf`, which calls `printf_core`, which itself calls `out`. All of these functions are in `src/stdio/vfprintf.c`. Their job is to expand the token in the string passed to `printf` (e.g. `%d`) with the value of the corresponding variables. For example, they transform `"hello: %d, %lf", 12, 12.0"` into `"hello: 12, 12.00000"`.

`out` calls `__fwritex` which is in `src/stdio/fwrite.c`. In `__fwritex` there is a call to a function pointer: `f->write(f, s, i);`. We do not describe function pointers in details here[3], just know that this will lead at run-time to a call to `__stdout_write(f, s, i)`. That function is located in `src/stdio/__stdout_write.c` and calls `__stdio_write` in `src/stdio/__stdio_write.c`, which itself calls `syscall(SYS_writev, f->fd, iov, iovcnt)`.

`syscall` is a macro expansion, defined in `src/internal/syscall.h`. It calls `syscall3` as follows:

```
__syscall_3(SYS_writev, f->fd, iov, iovcount);
```

This will lead to a call to the `writev` system call[4] with parameters `f->fd`, `iov`, and `iovcount`. `writev` realises several write operations at once in a file descriptor, each characterised by a memory address to write from, and the amount of bytes to write: in other words it is equivalent to a series of calls to `write` on the same file descriptor. The series of operation is described in an array of `iovec` data structures:

```c
struct iovec {
    void  *iov_base;    /* Starting address */
    size_t iov_len;     /* Number of bytes to transfer */
};
```

[3]For more information please see: https://en.wikipedia.org/wiki/Function_pointer#Example_in_C
[4]https://linux.die.net/man/2/writev

`writev` takes as parameter the file descriptor to write to, (in the call above, `f->fd`), the array of `iovec` structs (`iov`), and its size (`iovcount`). At runtime `f->fd` will be `1`, a special file descriptor representing the standard output. Indeed, in UNIX-like operating systems such as Linux, *everything is a file*[5], and printing to the console is realised by writing to a (pseudo-) file representing the standard output.

`syscall3` is a function defined in `arch/x86_64/syscall_arch.h`:

```
static __inline long __syscall3(long n, long a1, long a2, long a3) {
    unsigned long ret;
    __asm__ __volatile__ (  "syscall" :  // 5) the syscall instruction
                            "=a"(ret) :  // 6) we'll get the return value in rax
                            "a"(n),      // 1) syscall number in rax
                            "D"(a1),     // 2) argument 1 in rdi
                            "S"(a2),     // 3) argument 2 in rsi
                            "d"(a3) :    // 4) argument 3 in rdx
                            "rcx", "r11", "memory");
    return ret;
}
```

This function will be inlined by the compiler wherever it is called, as indicated by the `__inline` keyword, hence it is fine to have its body implemented in the `syscall_arch.h` header file. We will not go into the details of how inline assembly works here, but know that this generates the code adhering to the ABI previously described. In assembly and in the proper order, instructions will look approximately like this:

```
mov $20, %rax
mov $1, %rdi
mov <address of the iovec array>, %rsi
mov $1, %rdx
syscall
```

The steps are:

1. The system call identifier of `writev` (`n`, i.e. `SYS_writev`, i.e. `20`) is placed in `%rax`.
2. The 3 parameters (file descriptors, array of struct `iovec`, size of the array) are placed in `%rdi`, `%rsi`, and `rdx`.
3. The `syscall` instruction is invoked. At that point the kernel starts to run to process the system call. We will discuss what happens there further in the next lecture.
4. When the kernel returns to the application from the system call execution, the return value of the system call (an `unsigned long`) is present in `%rax`. In `__syscall3` it is placed in the `ret` variable, and returned up the call stack.

Hello World without Libc

Now that we know how to issue a system call to print on the console, an interesting exercise is to achieve that without the libc, i.e. without `printf`. This can be done with the following program:

```
// nolibc.c

// Print "hello!" to the standard output without the C library, directly making a
// write syscall to stdout file desccriptor (by convention 1)
// Notice the abscense of '#include': we don't want to use the libc

/* Without libc the default entry point is _start */
void _start() {
    unsigned long ret;

    /* Write syscall */
    __asm__ __volatile(
            "syscall" :             // the syscall instruction
            "=a"(ret) :             // we'll get the return value in rax
```

[5] https://en.wikipedia.org/wiki/Everything_is_a_file

```
            "a"(1),                // syscall number (1 for write)
            "D"(1),                // argument1: file descriptor (1 for stdout)
            "S"((long)"hello!\n"), // argument2: char array to print
            "d"(7) :               // argument3: number of bytes to write
            "rcx", "r11", "memory");

    /* exit syscall to quit properly */
    __asm__ __volatile( "syscall" : :  // syscall instruction
            "a"(60),                   // exit's syscall number
            "D"(0) :                   // exit parameter: 0
            "rcx", "r11", "memory");
}
```

This program issues 2 system calls: one `write` to the standard output, followed by `exit` to terminate the program. It can be compiled without the libc and executed as follows:

```
$ gcc -nostdlib nolibc.c -o nolibc
$ ./nolibc
hello!
```