COMP26020 Programming Languages and Paradigms -- Part 1

Case Study: C Standard Library

The C Standard Library

- Provides and implement all the functions from `stdio.h`, `stdlib.h`, etc.
- C + a tiny bit of assembly
- Default libc in most Linux distribution Glibc
  - Complex and hard to study
  - https://www.gnu.org/software/libc/

- Simpler: Musl

  - https://www.musl-libc.org/

Using Musl

Simple to try it out:

```
# 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!
```

- Let's have a look at **how Musl implements `printf`**

System Calls

- Application request services from the OS through **system calls**
- Generally not made directly by user code but rather the libc
- Upon syscall: user/kernel "world switch"
- System calls have a **calling convention**
  - Machine instructions used to trigger a syscall

System Calls

On Intel x86-64, this convention is as follows:

1. Syscall *number* in `%rax`
   - List here: https://filippo.io/linux-syscall-table/
2. Syscall parameters in order in `%rsi`, `%rdx`, `%r10`, `%r8`, and `%r9`
3. Execute the `syscall` instruction
4. Return value in `%rax`

Musl's `printf` Implementation

```
printf("value: %d %lf\n", int_val, double_val);
       ^^^^^^^^^^^^^^^^^  ^^^^^^^^^^^^^^^^^^^
          fmt                    ...
```

- In [src/stdio/printf.c]():

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

int printf(const char *restrict fmt, ...)
{
    int ret;
    va_list ap;                         // variable argument list management
    va_start(ap, fmt);
    ret = vfprintf(stdout, fmt, ap);  // stdout: data structure representing the std output
    va_end(ap);
    return ret;
}
```

Musl's `printf` Implementation

In src/stdio/vfprintf.c:

```
/* lots of logic for format string parsing ... */

int vfprintf(FILE *restrict f, const char *restrict fmt, va_list ap) {
/* ... */
    ret = printf_core(f, fmt, &ap2, nl_arg, nl_type);
/* ... */
}

static int printf_core(FILE *f, const char *fmt, va_list *ap, /* ... */) {
/* ... */
    if (f) out(f, a, l);
/* ... */
}

static void out(FILE *f, const char *s, size_t l)
{
     if (!(f->flags & F_ERR)) __fwritex((void *)s, l, f);
}
```

All these transforms something like `"hello: %d, %lf", 12, 12.0` into `"hello: 12, 12.00000"`

Musl's `printf` Implementation

In [src/stdio/fwrite.c]():

```
size_t __fwritex(const unsigned char *restrict s, size_t l, FILE *restrict f)
{
/* ... */
    size_t n = f->write(f, s, i);   /* this is actually a call to __stdout_write(f, s, i); */
/* ... */
}
```

In [src/stdio/__stdout_write.c]():

```
size_t __stdout_write(FILE *f, const unsigned char *buf, size_t len)
{
    struct winsize wsz;
    f->write = __stdio_write;
    if (!(f->flags & F_SVB) && __syscall(SYS_ioctl, f->fd, TIOCGWINSZ, &wsz))
        f->lbf = -1;
    return __stdio_write(f, buf, len);
}
```

Musl's `printf` Implementation

In [src/stdio/__stdio_write.c]():

```
size_t __stdio_write(FILE *f, const unsigned char *buf, size_t len)
{
/* ... */
    cnt = syscall(SYS_writev, f->fd, iov, iovcnt);
/* ... */
}
```

- In C, printing to the standard output corresponds to a `write` (or `writev`) syscall on a file descriptor representing the standard output

Musl's `printf` Implementation

In src/internal/syscall.h:

```
/* ... */
#define __SYSCALL_NARGS_X(a,b,c,d,e,f,g,h,n,...) n
#define __SYSCALL_NARGS(...) __SYSCALL_NARGS_X(__VA_ARGS__,7,6,5,4,3,2,1,0,)
#define __SYSCALL_CONCAT_X(a,b) a##b
#define __SYSCALL_CONCAT(a,b) __SYSCALL_CONCAT_X(a,b)
#define __SYSCALL_DISP(b,...) __SYSCALL_CONCAT(b,__SYSCALL_NARGS(__VA_ARGS__))(__VA_ARGS__)

#define __syscall(...) __SYSCALL_DISP(__syscall,__VA_ARGS__)
#define syscall(...) __syscall_ret(__syscall(__VA_ARGS__))
/* ... */
```

```
// This is a set of macros that transforms "syscall(SYS_writev, f->fd, iov, iovcnt); we saw
// earlier into:
__syscall_3(SYS_writev, f->fd, iov, iovcount);
```

Musl's `printf` Implementation

We now reach the architecture-specific part, let's look at the x86-64 code, 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;
}
```

```
  401bae:    41 be 14 00 00 00        mov    $0x14,%r14d
 ...
  401beb:    48 63 7b 78              movslq 0x78(%rbx),%rdi
  401bef:    49 63 d5                 movslq %r13d,%rdx
  401bf2:    4c 89 f0                 mov    %r14,%rax
  401bf5:    48 89 ee                 mov    %rbp,%rsi
  401bf8:    0f 05                    syscall
```

Printing Without Libc

- Now that we know how to print on stdout, we can make a syscall directly from our program without involving libc

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

    /* write(1, "hello!\n", 7); */
    __asm__ __volatile(
            "syscall" :            // the syscall instruction
            "=a"(ret) :            // we'll get the return value in rax
            "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(0); */
    __asm__ __volatile( "syscall" : :  // syscall instruction
            "a"(60),                   // exit's syscall number
            "D"(0) :                   // exit parameter: 0
            "rcx", "r11", "memory");
}
```

21-libc-case-study/nolibc.c

Printing Without Libc

```
0000000000001000 <_start>:
    1000:    55                       push   %rbp
    1001:    48 89 e5                 mov    %rsp,%rbp
    1004:    48 8d 35 f5 0f 00 00     lea    0xff5(%rip),%rsi # argument 2, address of "hello!"
    100b:    b8 01 00 00 00           mov    $0x1,%eax   # syscall number, 1 == write
    1010:    bf 01 00 00 00           mov    $0x1,%edi   # argument 1, fd 1 == stdout
    1015:    ba 07 00 00 00           mov    $0x7,%edx   # argument 3, 7 bytes to write
    101a:    0f 05                    syscall
    101c:    48 89 45 f8              mov    %rax,-0x8(%rbp)
    1020:    b8 3c 00 00 00           mov    $0x3c,%eax # syscall number, 60 == exit
    1025:    ba 00 00 00 00           mov    $0x0,%edx  # argument 1: 0
    102a:    89 d7                    mov    %edx,%edi
    102c:    0f 05                    syscall
    102e:    90                       nop
    102f:    5d                       pop    %rbp
    1030:    c3                       retq
```

Feedback form

https://bit.ly/37uNB0m