CS-382 Computer Architecture and Organization Fall 2022

Lab 11 · System Calls (Part I)

Lecturer: Shudong Hao *Date: See Canvas*

This is the first part of the System Call lab, where we are going to use system functions (instead of C functions) to receive inputs from the keyboard and print out numbers to the screen. The System Call lab will be useful as a transition to CS-392 next semester.

For Part I, we need to call `read` to receive user inputs as **strings**, convert that string into an integer, and save that integer to a register.

1 Requirements

- When your program starts, you don't need to print any prompt;
- Then you should type an integer, and hit Enter. The program will finish at that time;
- The number you typed is a string, so you need to convert that string into an integer number, and **save it to `X0`**;
- In your PDF report, submit a screenshot of your program, and use `gdb` to verify that the value is indeed inside the register. You can have two screenshots, one for the execution of the program, and the other for `gdb`;
- You are not allowed to use **any** C library functions such as `printf` and `scanf`. Code with the use of C library functions will receive zero credit.

Deliverable

One `.s` file and one PDF report.

A System Call

Other than functions provided in C library, we can use system functions directly to perform similar operations (just a bit more work). In assembly, we don't branch and link to system functions as in `scanf`; instead, we follow system calling conventions by passing arguments to corresponding registers, and invoke an interrupt (in x86 machines) or a supervisor call (in ARM chips).

As an example which we have used and seen countless times, to exit a program, here's what we need to do:

```
MOV   X0, 0
MOV   X8, 93
SVC   0
```

`X0` is the first and the only argument for system function `exit()`, while `X8` is the system call number. After the parameters are set up, we use `SVC 0` to invoke supervisor call. You can verify this by looking at the row 93 of the calling convention table. The calling convention for ARMv8 can be found in: https://chromium.googlesource.com/chromiumos/docs/+/HEAD/constants/syscalls.md#arm64-64_bit.

B Receive Input from `read()`

The prototype of `read` is defined as follows:

```
ssize_t read(int fd, void* buf, size_t nbyte);
```

where the first argument `fd` is a file descriptor, the second `buf` is the address where the received string will be stored, and `nbyte` is the number of bytes of the string. The `read` function returns the number of bytes successfully written into the file, which may at times be less than the specified `nbyte`. It returns `-1` if an exceptional condition is encountered.

The file descriptor is an integer representing the file of the operation. We usually have values 0, 1, 2, for standard input, standard output, and standard error, respectively. For `read()`, we use standard input (`fd == 0`).

We usually have no idea how many characters will be received from the user, and so the variable `nbyte` cannot be decided statically. We can use a loop to address this, where in each iteration we only receive one character, and we break the loop when we receive a `'\n'`.

Reminder: don't forget that some registers can't be trusted during a procedure call.

C Converting a String to an Integer

It is quite straightforward to convert a string to an integer. For example, if the string received is `9082`, the number is just $9 \times 10^3 + 0 \times 10^2 + 8 \times 10^1 + 2 \times 10^0$. Just remember that the characters in a string are stored as its ASCII value, not the real digit.