COM 341, Operating Systems

Project #1 Part #2: System Calls

Note, that you will have to repeat the process below for every CPU architecture.

Initial Steps

1. Move into the directory with the Debian hard drive image.

```
cd 'syscall-project/debian/<arch>'
```

2. Start the Debian Linux in QEMU.

```
./start.sh
```

3. Start a new shell instance.
4. Copy the `ish` directory from your machine to the emulated system with `scp` or git.
5. Log in into the Debian system through SSH. For MIPS and RISC-V Debian versions the login should be root. In other cases it should be the login that you have specified during the installation process.

```
ssh -p 2222 <login>@127.0.0.1
```

7. Go into the directory `ish`.

   In this task, you need to perform a number of system calls to the Linux kernel without the help of any standard libraries.

   The calls are

```
read
chdir
exit
stat
open
creat
dup2
close
fork
execve
waitpid
write
```

   You have sources of a simple shell that performs the calls above with the help of a system library.

   Your task is to implement functions

```
ish_read
ish_chdir
ish_exit
ish_stat
```

```
ish_open
ish_creat
ish_dup2
ish_close
ish_fork
ish_execve
ish_waitpid
ish_write
```

in `ish_syscalls.<arch>.c` to perform the system calls to the kernel directly on your own. That will remove the dependency on the system library for the shell at hand.

Open documentation files for each related system call to get information about return values and parameter types.

```
man 2 read
man 2 write
...
```

Library type definitions can be replaced with the following

```
typedef size_t  unsigned long
typedef ssize_t long
typedef pid_t   int
```

The `struct stat` from `sys/stat.h` for the `stat` system call can be replaced with an array large enough to contain file information from the kernel (as we are only interested in the return code). For an invalid pointer to a `struct stat`, the `stat` system call always returns an error.

```
char stat[1024];
```

Test each custom system call function by replacing a related standard library call in `ish.c` with a call to your new implementation. When everything is ready and works as expected, return the old system call names in `ish.c` and remove the `-D ISH_USE_STDLIB` preprocessor definition from the project's Makefile. Your shell will use the new calls after that. You can always switch back by adding the definition back.

Not all system calls are available on all hardware platforms. In certain cases you will have to find a replacement and adjust the arguments in registers to perform the operation. For example, you may have to replace `stat` with `fstatat` or `open` with `openat`. In order to find which calls to use, you can write a small C program to perform the call from C and then analyze the assembly generate by the compiler. GNU Debugger could be usfull in that particular case.

Managing Sources and Executables

- Edit the code. Add implementation for every system call in files `ish_syscall.<arch>.c`. Change related system call names in `ish.c` to the newly implemented function names (e.g., change `read` to `ish_read`).

```
vim ish_syscalls.<arch>.c
vim ish.c
```

- Build or rebuild the shell.

```
make
```

- Test the shell by typing builtin commands such as `cd` and `exit`. Try to start various system programs such as `ls` or `date` with an absolute path to the executable or without it, with parameters or without them. Try to redirect standard input or output streams with `<` and `>` operators.

```
./ish
/bin/ls
/bin/ls -l /etc
```

press `CTRL+C` to exit

```
./ish
cd
ls
exit
```

```
./ish
date
exit
```

```
./ish
cd /tmp
ls > listing.txt
cat < listing.txt
exit 1
```

Note, that we have a toy shell that does not know how to do a lot of things. You will not be able to edit your input. You will not be able to use piping. Autocomplete of commands is not available here.

In order to test a specific system call, you will have to figure which action in the shell utilizes that particular call under test.

- Remove compiled files.

```
make clean
```

Resources

- GCC Documentation, How to Use Inline Assembly Language in C Code
- GCC-Inline-Assembly-HOWTO
- ARM GCC Inline Assembler Cookbook
- Linux x86-64 System Call Tables
- Linux x86 System Call Tables

- Bionic Standard C Library, Linux ARMv7-A System Call Table
- Bionic Standard C Library, Linux ARMv8-A System Call Table
- Bionic Standard C Library, x86-64 System Calls
- Bionic Standard C Library, x86 System Calls
- Bionic Standard C Library, ARMv7-A System Calls
- Bionic Standard C Library, ARMv8-A System Calls

System call numbers for MIPS and RISC-V you will have to find on your own.

Documentation

```
man 2 syscall
man 2 syscalls

man 2 read
man 2 chdir
man 2 exit
...
```

Reading

- C Books and Guides
- *Understanding the Linux kernel, Third Edition by Daniel P. Bovet and Marco Cesati, Chapters 4, 10*
- *Linux Kernel Development, Third Edition by Robert Love, Chapters 5, 7*