EEE3535-01 **Fall 2020**

Assignment 2: System Call

Due: Sunday, Oct. 11, 2020, 11:59PM

1 Introduction

- The objective of this assignment is to add new system calls to xv6-riscv that you should have installed during the Assignment #0.
- Before starting the assignment, go to the `xv6-riscv/` directory, download the following script file, and execute it to update xv6-riscv. The script makes some minor changes to xv6-riscv for this assignment.

```
$ cd xv6-riscv/
$ wget https://icsl.yonsei.ac.kr/wp-content/uploads/syscall.sh
$ chmod +x syscall.sh
$ ./syscall.sh
```

- Launch xv6-riscv, and try executing the `syscalltest` program. If it prints out a `pid` number, then the update should have been successful. Terminate xv6-riscv by pressing `Ctrl+a` and then `x`.

```
$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 1
-nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,
drive=x0,bus=virtio-mmio-bus.0

EEE3535 Operating Systems: booting xv6-riscv kernel
EEE3535 Operating Systems: starting sh
$ syscalltest
pid = 3
$ QEMU: Terminated
```

- Go to the `user/` directory in xv6-riscv, and open the `syscalltest.c` file. You should find the following piece of simple code.

```
int main(int argc, char **argv) {
    // Get the process ID.
    printf("pid = %d\n", getpid());

    // Get the process ID of parent process.
    printf("ppid = %d\n", getppid());

    // Get the process name.
    char name[16];
    printf("pname = %s\n", getpname(name) < 0 ? 0 : name);

    exit(0);
}
```

- The first `printf()` function prints out the return value of `getpid()` system call that retrieves the process ID of current process (i.e., `syscalltest` program). `getpid()` is one of the default system calls already implemented in xv6-riscv.
- But, you may notice that the second and third `printf()` functions are commented out. `getppid()` and `getpname()` are the new system calls that you will have to implement in this assignment. Since they are not implemented yet, the system calls are commented out to avoid compile errors.
- The `getppid()` (note double p's) is supposed to return the process ID of parent process that forked and executed this `syscalltest` process. The `getpname()` should retrieve the process name of current process (i.e., `syscalltest` program) in the `char name[16]` character array. This system returns `-1` if it fails to read the process name or otherwise returns `0`.

2 Implementation

- Where are the system calls in xv6-riscv? To get an answer to this question, go to the `kernel/` directory in xv6-riscv. If you are still in the `user/` directory, type `cd ../kernel/` to relocate to the kernel.

- Since the `getpid()` (note a single 'p') is already implemented, search where `getpid`-related things are written. The following `grep` result tells us that there are three files including the keyword of `getpid`, i.e., `syscall.c`, `syscall.h`, and `sysproc.c` files.

```
$ grep -n getpid *
syscall.c:93:extern uint64 sys_getpid(void);
syscall.c:119:[SYS_getpid]  sys_getpid,
syscall.h:12:#define SYS_getpid 11
sysproc.c:21:sys_getpid(void)
```

- Visit the related files to learn how the `getpid()` system call is implemented. They will give you some ideas or hints about how to write the new `getppid()` (note double p's) system call.

- A process ID (for the `getpid()` system call) is included in a process control block (PCB). The PCB is defined as `struct proc` at the bottom of `proc.h` file, and the C struct contains an integer variable `int pid` that stores the process ID. The `getpid()` system call simply reads and returns the integer value.

- With this much information, getting the process ID of parent process must be a piece of cake, since the PCB already contains all the necessary things.

- However, enabling the `getppid()` system call still needs to modify a couple of files in the `user/` directory. Do the same `grep` search in the user directory to find which files contain `getpid`-related things, and get a hint from them to figure out how to implement the new `getppid()` system call.

- You may wonder at this moment how system calls actually work in xv6-riscv, although you learned it conceptually in the class. When a user process invokes a system call, it jumps to the `user/usys.S` file where the syscall function is defined in RISC-V assembly code. It puts a syscall number in the `a7` register and then executes an `ecall` trap instruction. Upon calling the `ecall` instruction, the CPU jumps to `kernel/trampoline.S`. It was agreed between the OS and hardware to do so when initializing xv6-riscv in `kernel/kernel.ld`. The `uservec` function in `trampoline.S` is executed, and it saves the context of current user process that initiated this syscall process. Then, it calls the `usertrap()` function defined in the `trap.c` file. The `usertrap()` function recognizes that the CPU was interrupted because of the system call, and it invokes the `syscall()` function in the `syscall.c` file. The `syscall()` function identifies the syscall type by reading the `a7` register that the user process has set earlier. Then, the OS performs the requested system call. When done, the `usertrapret()` function at the end of `usertrap()` is called to restore the context of user process and return to it.

- Implementing the `getpname()` system call is trickier than `getppid()`. The `getppid()` system call has no function input arguments, and it only needs to return a copy of integer value.

- On the other hand, the `getpname()` system call takes a pointer to a character array as an input. A PCB contains the `char name[16]` field that stores the name of corresponding process, and the `getpname()` system call is supposed to return this string. However, a user process is not allowed to directly access the string name in the kernel space, and doing so will make xv6-riscv crash. Instead, the system call must copy the string to user-provided character array that comes as the input of `getpname()` system call.

- Copying a string from kernel to user space is not trivial as simply calling a `memcpy()` function, since it needs to perform virtual-to-physical memory address translation - memory is virtualized! Luckily, this can be easily done by calling a `copyout()` function defined in the `vm.c` file. The `copyout()` function is analogous to `memcpy()`, but it takes the page table of a process as its first function argument. You do not have to worry about the meaning of page table at this moment but simply put `pagetable` of the process' PCB as the argument.

- The `getppid()` system call returns `0` if everything went fine, or otherwise it should return `-1`.

- To test your implementation of `getppid()` and `getpname()` system calls, open the `user/syscalltest.c` file and uncomment the second and third `printf()` functions. Then, go to the `xv6-riscv/` directory, and rebuild xv6-riscv. A complete implementation should work as follows.

```
$ make clean
...
$ make qemu
...
EEE3535 Operating Systems: booting xv6-riscv kernel
EEE3535 Operating Systems: starting sh
$ syscalltest
pid = 3
ppid = 2
name = syscalltest
$ syscalltest | grep ppid
ppid = 4
```

3 Submission

- In the `xv6-riscv/` directory, execute the `tar.sh` script. This script will compress the current `xv6-riscv/` directory into a tar file named after your student ID (e.g., `2020142020.tar`).
- Upload the created tar file on YSCEC. Do not rename the tar file by adding your name, project2, etc.

4 Grading Rules

- The following is a general guideline for grading the assignment. 30-point scale will be used for this assignment. The minimum score is zero, and negative scores will not be given. Grading rules are subject to change, and a grader may add a few extra rules for fair evaluation of students' efforts.

  -3 points: The tar file is renamed and includes some other tags such as a student name.

  -5 points: Program codes do not have sufficient comments. Comments in the baseline xv6-riscv do not count. You must make an effort to clearly explain what each part of your implementation intends to do.

  -15 points per incomplete system call: This assignment asks you to write two system calls, `getppid()` and `getpname()`. Each incomplete system call will lose 15 points.

  -30 points: No or late submission.

  F grade: A submitted code is copied from someone else. All students involved in the incident will be penalized and given F for final grades irrespective of assignments, attendance, etc.

- Your teaching assistant (TA) will grade your assignments. If you think your assignment score is incorrect for any reasons, discuss your concerns with the TA. Always be courteous when contacting the TA. In case no agreement is made between you and the TA, elevate the case to the instructor to review your assignment. Refer to the course website for the contact information of TA and instructor: `https://icsl.yonsei.ac.kr/eee3535`
- Begging for partial credits for no viable reasons will be treated as a cheating attempt, and thus such a student will lose all scores for the assignment.