

Assignment 2: System Call and Process

Due: Sunday, Oct. 15, 2023, 11:59PM

1 Introduction

- The objective of this assignment is to add a new system call to xv6-riscv that probes processes and prints out their information, including process ID, execution state, runtime, and program name.
- A `ps` command in the Linux terminal prints the information of active processes, including process ID, controlling terminal, CPU time, and command name. You may append options to print more detailed process information, such as `ps -af`.

```
$ ps
  PID TTY          TIME CMD
 52724 pts/1    00:00:00 bash
 60967 pts/1    00:00:00 ps
```

- Although the `ps` program in Linux uses a file to record and read process states, we will use a system call in xv6-riscv to implement similar functionality.
- This assignment mainly consists of two parts; implementing i) the `ps` user program and iii) the new `pstate` syscall in the kernel.
- The following shows an expected output of the `ps` program in xv6-riscv. It prints the process ID (PID), parent process ID (PPID), execution state, elapsed runtime, and program name of active processes. More explanations will follow in the next section.

```
$ ps
PID      PPID      State   Runtime   Name
1         0         S       0:0.7     init
2         1         S       0:0.5     sh
3         2         X       0:0.0     ps
```

2 Implementation

- To start the assignment, go to `xv6-riscv/`, download `syscall.sh`, and run the script to update xv6-riscv.

```
$ cd xv6-riscv/
$ wget https://ics1.yonsei.ac.kr/wp-content/uploads/syscall.sh
$ chmod +x syscall.sh
$ ./syscall.sh
```

- If the update is successful, you can run the new user program `ps`. Since the program is nearly empty in the skeleton code, nothing is printed.

```
$ make qemu
```

...

```
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 1
-nographic -global virtio-mmio.force-legacy=false -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
```

```
$ ps
$
```

- The following shows the skeleton code of `user/ps.c`.

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

// A xv6-riscv syscall can take up to six arguments.
#define max_args 6

// Print a help message.
void print_help(int argc, char **argv) {
    fprintf(2, "%s <options: pid or S/R/X/Z>%s\n",
            argv[0], argc > 7 ? ": too many args" : "");
}

int main(int argc, char **argv) {
    // Print a help message.
    if(argc > 7) { print_help(argc, argv); exit(1); }

    // Argument vector
    int args[max_args];
    memset(args, 0, max_args * sizeof(int));

    /* Assignment 2: System Call and Process
       Convert the char inputs of argv[] to integers in args[].
       In this skeleton code, args[] is initialized to zeros,
       so technically no arguments are passed to the pstate() syscall. */

    // Call the pstate() syscall.
    int ret = pstate(args[0], args[1], args[2], args[3], args[4], args[5]);
    if(ret) { fprintf(2, "pstate failed\n"); exit(1); }

    exit(0);
}
```

- Near the end of `main()`, `pstate()` is the new system call you will have to implement in this assignment.
- The syscall is defined as `int pstate(int, ...)` in `user/user.h`, which takes one or more `int` arguments. `xv6-riscv` allows passing up to six arguments to a system call.
- The basics of `pstate()` are already added to various files in `kernel/` and `user/` directories, so you do not have to worry about how to enable the new syscall. It is already enabled but does nothing at this moment.
- If you are not familiar with `main()` taking inputs, have a look at the example below to understand what `int argc` and `char **argv` are about. This code has nothing to do with `xv6-riscv`.

```
/* arg.c */

#include <stdio.h>

int main(int argc, char **argv) {
    printf("argc = %d\n", argc);
    for(unsigned i = 0; i < argc; i++) {
        printf("argv[%u] = %s\n", i, argv[i]);
    }
    return 0;
}
```

- Compiling and executing the `arg.c` code gives the following output. The result shows that `int argc` is the number of arguments in the run command, and `char **argv` is the array of `char` pointers, each pointing to a character string.

```
$ gcc -o arg arg.c
$ ./arg input1 input2 input3
argc = 4
argv[0] = ./arg
argv[1] = input1
argv[2] = input2
argv[3] = input3
```

- We will revisit `ps.c` later to discuss what has to be passed to the input arguments of `pstate()`.
- Where are the system calls, and how are they implemented in `xv6-riscv`?
- Since several basic system calls such as `getpid()` are already implemented in `xv6-riscv`, find where `getpid`-related codes are in the `kernel/` directory.

```
$ grep -n getpid *
syscall.c:93:extern uint64 sys_getpid(void);
syscall.c:118:[SYS_getpid] sys_getpid,
syscall.h:12:#define SYS_getpid 11
sysproc.c:19:sys_getpid(void)
```

- Open the `syscall.h`, `syscall.c`, and `sysproc.c` files to learn how `getpid()` is implemented. You should also find `pstate`-related lines near `getpid` in these files.
- Since process ID is a part of the information that the new `pstate()` syscall has to collect, take a look at how `getpid()` retrieves it.
- A process ID is stored in a process control block (PCB), which is defined as `struct proc` at the end of `proc.h`. The struct has `int pid` to save the process ID. The `getpid()` syscall simply reads its value.
- Who sets the process ID? Presumably, it is set when a process is created.
- Process creation occurs in the `allocproc()` function of `proc.c`. In the function, `p->pid = allocpid()` assigns an ID to a new process.
- You just learned that a process is created in `allocproc()`. This must be a good place to record the start time of the new process because `pstate()` needs to print the elapsed runtime of the process.
- In this assignment, the elapsed runtime will be measured simply as a time difference between the process creation and the `pstate()` call.
- How can we get the time information in `xv6-riscv`?
- It is impossible to get the exact time, but an approximate value can be obtained by reading `ticks`, which is defined in `defs.h` and globally available in the kernel.
- This variable is incremented every 100ms after `xv6-riscv` is booted up. For instance, `ticks == 7` means that the current time is about 0.7 seconds since the kernel launch. Using `ticks`, you can trace the elapsed runtime of a process at 100ms granularity.
- When printing the runtime, display it in the form of `1:23.4`, meaning 1 minute and 23.4 seconds have passed since the process was created.
- The PCB also has `enum procstate state`, `struct proc *parent`, `char name[16]`, indicating a process state (e.g., `RUNNABLE`, `SLEEPING`), pointer to the parent process, and program name (e.g., `sh`, `grep`), respectively.

- Process states are defined as `enum procstate {UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE}` right above `struct proc` in `proc.h`.
 - **UNUSED**: If a process state is **UNUSED**, this does not represent a valid process. `xv6-riscv` creates a process list simply as a static array defined as `struct proc proc[NPROC]` in `proc.c`, where `NPROC` is 64. It means that `xv6-riscv` can schedule only up to 64 processes at a time. **UNUSED** is to indicate an entry in the `proc[]` array that is currently not in use.
 - **USED**: When a process is newly created, its state is initially set to **USED**. Then, the state is soon changed to **RUNNABLE**. It is technically impossible to observe a process in a **USED** state, so the `pstate()` syscall will not trace which processes are in **USED** states.
 - **SLEEPING**: If a process in **SLEEPING**, it is blocked until some conditions are met to resume its execution.
 - **RUNNABLE**: This state indicates that a process is runnable and waiting for a job scheduler to give it a chance to use the CPU.
 - **RUNNING**: A process is running. Since `xv6-riscv` in this assignment is configured to use a single CPU, the only running process when `pstate()` is called must be the `ps` program itself.
 - **ZOMBIE**: Lastly, this state indicates that a process is done and being terminated. A process finishes its execution by calling the `exit()` syscall, and it finally gets deallocated via `freeproc()` in `proc.c`. The **ZOMBIE** state is very short but observable.
- It is tedious for `pstate()` to simply print the information of all active processes. So, let us switch back to `user/ps.c` and add a few options to the program.
- A `ps` command may be associated with options to print the information of processes only in certain states.
- For instance, `ps S` prints the information of processes in **SLEEPING** states only. Similarly, `R`, `X`, and `Z` options are for **RUNNABLE**, **RUNNING** (**EXECUTING**), and **ZOMBIE** states, respectively.
- You can also set multiple states such as `ps S R` to display processes in either **SLEEPING** or **RUNNABLE** states. The following shows a few examples with state options.

```
$ ps X
PID      PPID      State   Runtime   Name
4         2         X       0:0.1     ps
```

```
$ ps S
PID      PPID      State   Runtime   Name
1         0         S       0:6.7     init
2         1         S       0:6.5     sh
```

```
$ ps S X
PID      PPID      State   Runtime   Name
1         0         S       0:9.5     init
2         1         S       0:9.3     sh
6         2         X       0:0.0     ps
```

- Another option is to specify process IDs to print. For instance, `ps 1` only prints the process whose `pid` is 1. You may also specify multiple process IDs such as `ps 1 2` or `ps 5 1 2`. If a specified process ID is invalid, `ps` simply disregards it.
- The following shows a few examples with specified process IDs.

```
$ ps 1
PID      PPID      State   Runtime   Name
1         0         S       0:12.1    init
```

```
$ ps 8 2
PID      PPID      State   Runtime   Name
```

```

2      1      S      0:14.8      sh
8      2      X      0:0.1      ps

```

```

$ ps 9 1 3
PID      PPID      State   Runtime   Name
1        0        S       0:19.4    init
9        2        X       0:0.0     ps

```

- How about we mix both state and process ID options? Let the `ps` program print processes that satisfy any conditions. For instance, `ps X 2` prints the information of processes that are either in `RUNNING` states or have the process ID of 2. It is also possible that no processes satisfy the conditions.

```

$ ps X 2
PID      PPID      State   Runtime   Name
2        1        S       0:25.8    sh
10       2        X       0:0.0     ps

```

```

$ ps S Z 11
PID      PPID      State   Runtime   Name
1        0        S       0:29.6    init
2        1        S       0:29.5    sh
11       2        X       0:0.0     ps

```

```

$ ps Z R 3
PID      PPID      State   Runtime   Name

```

- By allowing the `ps` program to take multiple state and/or process ID options in a run command, you should pass them to the `pstate()` syscall.
- The `pstate()` syscall is defined as `int pstate(int, ...)` in `user/user.h`, which takes one or more integers as input arguments. `xv6-riscv` supports passing up to six arguments to a syscall, and the case of having more than six arguments is already screened in `user/ps.c`.
- Suppose a run command is `ps R X 15 2`, which should print the information of processes in either `RUNNABLE` or `RUNNING` states or having `pid` of 15 or 2.
- `argc` of `main()` in this case should be 5, where `argv[0] = ps`, `argv[1] = R`, `argv[2] = X`, `argv[3] = 15`, and `argv[4] = 2`. Since 15 and 2 are characters, you should convert them to integers using `atoi()` defined in `user/ulib.c`.
- How should you handle the state options of `R` and `X`, which are not integers? Since the number of process states is finite (i.e., only four states of `S`, `R`, `X`, `Z`), a possible solution is to encode them into negative integers. Process IDs are always positive integers, so encoding the process states into negative numbers will not conflict with process IDs. Use your own encoding method to translate the process states into integers.
- Then, the input arguments of `pstate()` can have three cases. If an argument is a positive number, it is a process ID. A negative number means a process state. If an argument is zero, it indicates the end of the options, similar to the null-terminated `argv[]` array of an `execvp()` call.
- Make the `ps` program in `user/ps.c` also do basic error checking for input arguments to screen invalid options such as `ps A -5 # 1+2`; `A` is not a valid state option, `-5` is not a valid process ID, `#` and `+` are invalid characters.
- For an invalid run command, call `print_help()` to dump a help message and exit as follows.

```

$ ps A -5 # 1+2
ps <options: pid or S/R/X/Z>

```

- The invocation of `pstate()` in the skeleton code is over-specified with six arguments regardless of whether there are actually six options.

- Calling the `pstate()` syscall traps into `usertrap()` in `kernel/trap.c` via a trampoline routine in `kernel/trampoline.S`. After looking up a trap table via `syscall()` in `kernel/syscall.c`, it will finally reach the `uint64 sys_pstate(void)` function at the bottom of `kernel/sysproc.c`.
- The input arguments of the `pstate()` syscall can be retrieved by using `argint()`, which reads RISC-V function argument registers from `a0` (or `x10`) to `a5` (or `x15`). Take a look at other syscalls that take some input arguments as a reference. `argint()` is defined in `kernel/syscall.c`.
- While sequentially reading CPU registers from `a0` to `a5`, encountering a zero indicates that there are no more arguments passed to the `pstate()` syscall. Otherwise, the sixth argument should be the last one.
- Once all the input arguments are retrieved in `sys_pstate()`, the syscall should search the process list (i.e., `struct proc proc[NPROC]`) to find if there are matching processes.
- Print out a header line and the information of matching processes.
- If the syscall is done without errors, return 0. Otherwise, a non-zero return value will be regarded as an error.

3 Validation

- When printing the result of `ps`, follow the formatting rules.
 1. **Header:** The first line must be a header in the form of `PID PPID State Runtime Name`. Separate each column by a single tab in the header and process information rows.
 2. **State:** A process state must be printed in a single uppercase character; `S` for `SLEEPING`, `R` for `RUNNABLE`, `X` for `RUNNING`, and `Z` for `ZOMBIE` states. Do not use other expressions.
 3. **Runtime:** The time must be printed in the `1:23.4` format as explained earlier.
- After launching `xv6-riscv`, execute the `usertests` program in the background with an `&` symbol.
- `usertests` will run for several minutes and create thousands of processes. But, you will see only a handful of them at each `ps` invocation. Concurrent execution of multiple processes (e.g., `sh`, `usertests`, `ps`) will mess up the ordering of `printf` results, but they are normal and expected.
- Your assignment will be graded based on the following five cases, but they are not the exact and only ones that will be on the test. Your code should be able to handle other similar common-sense situations.

1. Simple `ps` command:

```
$ usertests&
usertests starting

...

$ ps
PID      PPID     State   Runtime   Name
1         0        S       0:6.6     init
2         1        S       0:6.5     sh
5         4        R       0:3.5     usertests
4         1        S       0:3.5     usertests
6         2        X       0:0.8     ps
```

2. `ps` with state options:

```
$ ps X R
PID      PPID     State   Runtime   Name
17        4        R       0:2.2     usertests
18       17        R       0:2.1     usertests
19        2        X       0:0.2     ps
```

```
$ ps R S X Z
PID      PPID      State   Runtime   Name
1         0         S       0:19.1    init
2         1         S       0:19.0    sh
17        4         R       0:7.8     usertests
4         1         S       0:16.0    usertests
18        17        Z       0:7.7     usertests
20        2         X       0:0.0     ps
```

3. **ps** with process ID options:

```
$ ps 1
PID      PPID      State   Runtime   Name
1         0         S       0:27.4    initS
```

```
$ ps 4 1 2 3
PID      PPID      State   Runtime   Name
1         0         S       0:29.9    init
2         1         S       0:29.8    sh
4         1         S       0:26.8    usertests
```

4. **ps** with mixed state and process ID options:

```
$ ps S 1 2 R
PID      PPID      State   Runtime   Name
1         0         S       0:36.5    init
2         1         S       0:36.4    sh
36        4         R       0:8.9     usertests
4         1         S       0:33.4    usertests
```

5. **ps** with invalid options:

```
$ ps -5 A
ps <options: pid or S/R/X/Z>

$ ps 1+5 R S Z
ps <options: pid or S/R/X/Z>
```

4 Submission

- In the `xv6-riscv/` directory, execute the `tar.sh` script to create a tar file named after your student ID (e.g., `2023143535`).

```
$ ./tar.sh
$ ls
2023143535.tar  kernel  LICENSE  Makefile  mkfs  README  tar.sh  user
```

- Upload the tar file (e.g., `2023143535.tar`) on LearnUs. Do not rename the file.

5 Grading Rules

- The following is the general guideline for grading. A 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; a grader may add a few extra rules without notice for a fair evaluation of students' efforts.

-5 points: The submitted tar file includes redundant tags such as a student name, `hw2`, etc.

-5 points: The code has insufficient comments. Comments in the skeleton code do not count. You must clearly explain what each part of your code does.

-6 points each: The validation section has five test cases. Each failed test will lose 6 points.

-30 points: No or late submission.

Final grade = F: The submitted tar file is copied from someone else. All students involved in the incidents will get Fs for the final grade.

- Your teaching assistant (TA) will grade your assignments. If you think your assignment score is incorrect, discuss your concerns with the TA. Always be courteous when contacting the TA. If no agreements are 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 the TA and instructor: <https://icsl.yonsei.ac.kr/eee3535>
- Arguing for partial credits for no valid reasons will be regarded as a cheating attempt; such a student will lose the assignment scores.