Project 1A
COM S 362
Fall 2021

Note: Iteration 1A must be completed by yourself. Only future project iterations will allow working in pairs (teams of 2).

1. Introduction

For this project, you will be creating your own system call in xv6. The new system call, `pcount`, will return the total number processes currently on the system.

2. Setup

Follow the steps in the Xv6 Development Setup guide linked to by Module 2 in Canvas. You can use an alternative setup, for example, installing xv6 on your local machine.

3. Organization of Xv6

The xv6 source is divided into two folders: `kernel` and `user`. As their names imply kernel is the part of the operating system that executes in kernel mode and user contains several utility programs including a simple shell.

To implement a system call we need to bridge the gap between user and kernel. Recall, a simple function call will not work because a user mode application does not have the privileges required to jump to instructions in the kernel's code. A system call is invoked when the user mode application executes a privileged instruction, which cause a trap, resulting in one of the kernel's interrupt handlers being executed.

On the user side, we can see the calls to `ecall` (a privileged instructions) in the file `user/usys.S` for each of the system calls (e.g., `fork()`, `exit()`, `wait()`…). We will not need to directly modify this assembly language file, it is automatically generated by a Perl script.

On the kernel side, the interrupt handler that is invoked on a user caused trap is `uservec` in `kernel/trampoline.S`. We can see that it saves the user's CPU registers (context), calls `usertrap()` located in the file `kernel/trap.c`, and then restores the user's CPU registers.

The function `usertrap()` calls `syscall()` in `knernel/syscall.c` which uses the system call number to call the correct function from the array of function pointers `syscalls[]`.

4. Adding a New System Call (Kernel Side)

Now you will add and test your own system call in xv6! Start by modifying the following files.

kernel/syscall.h

Add a new system call number using the name `SYS_pcount`. Follow the example of the other system calls in `syscall.h`. The system call number is used to look up which function to call from the `syscalls[]` array of function pointers, we will modify that next.

kernel/syscall.c

Declare the kernel side function that will be called: `sys_pcount(void)`. Search for another system call function such as `sys_uptime` and follow the same pattern for `sys_pcount`. You will need to add code to two places in this file.

kernel/proc.h

The Process Control Blocks (PCBs) in xv6 are stored in an array called `proc` (we will refer to this as the process table). To get access to the process table from any file in the kernel, add the following line to the bottom of `kernel/proc.h`:

```
extern struct proc proc[NPROC];
```

kernel/sysproc.c

Now create the full definition for the `sys_count` function in `sysproc.c`. Again, search for `sys_uptime` and use it as an example.

The purpose of `pcount` is the return the total number of processes on the system. All processes are maintained in the array of proc structs called `proc[]`. The size of the array is `NPROC`. Iterate through the array to find all array entries that have a state that is not `UNUSED`. The state of `UNUSED` indicates the array entry is empty, it does not store a process.

5. Testing the System Call (User Side)

user/usys.pl

Add the new system call to the Perl script that automatically generates the required assembly code. Follow the example of `uptime` to add an entry for `pcount`.

`user/user.h`

Search for `uptime` and add a similar function declaration for `pcount`.

Makefile

Now we will make a new utility program to test the system call. In the `Makefile` search for `zombie` and add a similar line for `ptest`.

`user/ptest.c`

Copy the code from `user/zombie.c` to a new file called `user/ptest.c`.

Add statements to print the process count before calling fork and after the parent calls sleep have the parent print the process count. Are both counts the same or different, what should be the expected result and why? Add a comment to the code answering this question.

6. Submission

Document `user/ptest.c` with your name and answer to the question above, no other documentation is required. Submit a zip file of the xv6-riscv directory. On pyrite the zip file can be created using:

```
$ zip -r xv6-riscv.zip xv6-riscv
```

Submit xv6-riscv.zip.