CS 550 Operating Systems, Spring 2024
Programming Project 1 (PROJ1)

Out: 2/4/2024, SUN
Due date: 2/24/2024, SAT 23:59:59

In this project, you will implement a functionality that prints the ancestor processes of a given process in xv6. This project allows you to learn how system calls are implemented in modern OSes and how system calls are invoked from a user program in xv6.

1 GitHub account username

All the projects in this semester will be administered collectively using GitHub Classroom and Brightspace. By default, our grading script assumes that your GitHub username is the same as your BU username (if your BU email address is "`abc@binghamton.edu`", "`abc`" is your BU username). If your GitHub username differs from your BU username, inform us by completing the following form so our grading script can be updated accordingly. If your GitHub username is not submitted before grading starts, 5 points will be deducted from your final score for this project.

Form link: `https://forms.gle/bLV2TRNFmDiW9Jg8A`

(Continue to the next page.)

2 Baseline source code

For this and all the later projects, you will work on the base code that needs to be cloned/downloaded from your own private GitHub repository. Make sure you read this whole section, as well as the grading guidelines (Section 7), before going to the following link at the end of this section.

- Go to the link at the end of this section to accept the assignment.
- Work on and commit your code to the default branch of your repository. Do not create a new branch. Failure to do so will lead to problems with the grading script and 5 points off of your project grade.

Assignment link: `https://classroom.github.com/a/lCWeF5-L`

(Continue to the next page.)

3 Build and run xv6

The following provides instructions on building and running xv6 using either a CS machine or your computer.

(1) Log into a CS machine (i.e., a local machine or a remote cluster machine).

(2) Clone or download the baseline xv6 code.

(3) Compile and run xv6:

```
$ make qemu-nox
```

After compiling the kernel source and generating the root file system, the Makefile will start a QEMU VM to run the xv6 kernel image just compiled (you can read the Makefile for more details). Then, you will see an output similar to the following, indicating you have successfully compiled and run the xv6 system.

```
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
```

Troubleshooting:

- If you see the following error message:

```
make: execvp: ./sign.pl: Permission denied
```

  Solve it by running the following command:

```
chmod ugo+x ./sign.pl
```

  Then

```
make clean
make qemu-nox
```

- If you get a "No bootable device" error message, make sure you "make clean" before re-building the OS using "make qemu-nox".

(Continue to the next page.)

4 Coding: printing ancestors of a given process (60 points)

4.1 Implementing the system service

The coding task in this project is to implement a system service that allows user programs to obtain the names of ancestor processes of a given process. Your job is to

- Implement the interface for user programs to use the system service (Section 4.1.1).
- Implement the system calls for the service (Section 4.1.2).
- Implement the logic of the system service (Section 4.1.3).

The details are provided as follows.

4.1.1 The user program interface of the system service

You learned in class that OS system services are provided to user programs through system calls. The user program interface of system services is a set of library functions that hide the details of invoking system calls, making using system services easy for user programs. Each system call has a corresponding library function that user programs call to invoke the system call.

For the system service in this project, implement the following two library functions for a user program to use the service.

- `int set_proc_name(int pid, char * name)`
  - Description: This function sets the "name" field in the PCB (process control block) of the process specified.

    In xv6, when a process executes a program, the "name" field of the process' PCB is set to the name of the program. When a new process is forked, the new process' name is inherited from the parent. Therefore, without this new functionality to set the name of a process, parent/child processes that run the same program always have the same name.
  - Arguments: This function takes two arguments.
    - `pid`: the PID of the process of which the "name" is being set.
    - `name`: the name to set.
  - Return value: On the successful setting of the process name, this function returns 0. The function returns -1 on failures.
- `int print_proc_ancestors(int pid);`
  - Description: This function requests the OS to print the ancestor processes of the process specified.
  - Arguments: This function takes a single argument, `pid`, which is the PID of the process whose ancestor info is wanted.
  - Return value: This function returns 0 on success and -1 on failures.

Important note: The above two functions will be used by our test program to test your implementation. Make sure that your implementation follows exactly the above prototype specification.

4.1.2 The system calls

Implement the system calls corresponding to the two interface functions detailed above. You can name the system calls the way you want because they are not directly interfaced with the user program (i.e., the test program).

4.1.3 The system service logic

The service logic is fairly simple: print the names of all the ancestor processes of the specified process (in OS kernel).

4.1.4 Testing your implementation

Write a program (which is a user program) to test your implementation. For example, for the logic shown in the following code,

```
int pid = 0;

set_proc_name(getpid(), "testprog:parent");

pid = fork();
if (pid == 0)
{
    set_proc_name(getpid(), "testprog:child");
    exit();
}
else if (pid > 0)
{
    print_proc_ancestors(pid);
    wait();
}
else
{
    printf(1, "ERROR: fork() failed!\n");
}
```

The expected output should be:

```
testprog:child <== testprog:parent <== sh <== init
```

The above output shows that

- The "`init`" process is the oldest ancestor process ("`init`" process is the very first process crated in xv6).
- The "`init`" process forked the "`sh`" process (i.e, shell).
- The "`sh`" process forked the "`testprog:parent`" process.
- The "`testprog:parent`" process forked the "`testprog:child`" process.

NOTE:

1. The output generated by your implementation should follow the above format strictly. Specifically, in the case where the process of which the ancestor processes are wanted has $N-1$ ancestor processes, the output format should exactly be:

```
name-of-proc-N <== name-of-proc-(N-1) <== ... <== name-of-proc-1
```

where proc-N is the process whose PID was specified as the function argument when calling the `print_proc_ancestors()` function.

2. The test code logic shown above does not work as shown. You will need to make certain adjustments to allow it to be fully working.

4.1.5 Hints

- Reading and understanding how the existing user commands and the associated system calls are implemented will be helpful. For example, you can look at how the `cat` user command is implemented. The `cat` user command is implemented in `cat.c`, in which the system call `open()` is called. The actual work of the `open()` system call is done in the `sys_open()` function defined in sysfile.c.
- Understanding the mechanism is important. Pay attention to all the related files, which include assembly files (relax, the related assembly file is very easy to read).

(Continue to the next page.)

5 xv6 system call implementation Q&A (40 points)

Answer the following questions about system call implementation.

(1) (10 points) Where is the wrapper function of this system call that prints the ancestor processes of the specified process (i.e., the `int print_proc_ancestors(int pid)` function) defined?

(2) (10 points) Explain (with the actual code) how the above wrapper function triggers the system call.

(3) (10 points) Explain (with the actual code) how the OS kernel locates a system call and calls it (i.e., the kernel-level operations of calling a system call).

(4) (10 points) How are arguments of a system call passed from user space to OS kernel?

Key in your answers to the above questions with the editor you prefer, export them in a PDF file named "xv6-syscall-mechanisms.pdf", and submit the file to the assignment link in Brightspace.

6 Submit your work

Once your code in your GitHub private repository is ready for grading, submit a text file named "DONE" (and the previous "xv6-syscall-mechanisms.pdf") to the assignment link in Brightspace. We will not be able to know your code in your GitHub repository is ready for grading until we see the "DONE" file in Brightspace. Forgetting to submit the "DONE" file will lead to a late penalty applied, as specified later in the "Grading" section.

Important notes:

- If you have referred to any form of online materials or resources when completing this project (code and Q&A), please state all the references in this "DONE" file. Failure to do so, once detected, will lead to zero points for the entire project and further penalties depending on the severity of the violation.
- To encourage (discourage) early (late) starts on this project, the instructor and the TAs will not respond to questions related to the project on the due date.

Suggestion: Test your code thoroughly on a CS machine before submitting.

7 Grading

The following are the general grading guidelines for this and all future projects.

(1) **The code in your repository will not be graded until a "DONE" file is submitted to Brightspace**.

(2) The submission time of the "DONE" file shown on the Brightspace system will be used to determine if your submission is on time or to calculate the number of late days. Late penalty is 10% of the points scored for each of the first two days late, and 20% for each of the days thereafter.

(3) If you are to compile and run the xv6 system on the department's remote cluster, remember to use the baseline xv6 source code provided by our GitHub classroom. Compiling and running xv6 source code downloaded elsewhere can cause 100% CPU utilization on QEMU.

Removing the patch code from the baseline code will also cause the same problem. So make sure you understand the code before deleting them.

If you are reported by the system administrator to be running QEMU with 100% CPU utilization on QEMU, 10 points off.

(4) If the submitted patch cannot successfully patched to the baseline source code, or the patched code does not compile:

```
1   TA will try to fix the problem (for no more than 3 minutes);
2   if (problem solved)
3     1%-10% points off (based on how complex the fix is, TA's discretion);
4   else
5     TA may contact the student by email or schedule a demo to fix the problem;
6     if (problem solved)
7       11%-20% points off (based on how complex the fix is, TA's discretion);
8     else
9       All points off;
```

So in the case that TA contacts you to fix a problem, please respond to TA's email promptly or show up at the demo appointment on time; otherwise the line 9 above will be effective.

(5) If the code is not working as required in the project spec, the TA should take points based on the assigned full points of the task and the actual problem.

(6) **Lastly but not the least, stick to the collaboration policy stated in the syllabus: you may discuss with you fellow students, but code should absolutely be kept private. Any kind of cheating will result in zero point on the project, and further reporting.**