CS 350 Operating Systems, Spring 2024
Programming Project 1 (PROJ1)

Out: 09/09/2024, MON
Due date: 09/29/2024, SUN 23:59:59

In this project, you will add new functionality to xv6, including the ability to shut the system down, print exit statuses, and output uptime. You will also answer questions related to system call implementation in xv6. The goal of this project is to learn how system calls are implemented in modern OSes and how system calls are invoked from a user program in xv6.

This is a group project, and is expected to be completed with your group members. Each group member is responsible for completing the shutdown system call. Furthermore, each group member should choose one of four unique system calls to implement themselves. Each group member is responsible for submitting their own code on Brightspace.

This project contributes 6.25% toward the final grade.

1 Baseline source code

For this and all the later projects, you will be working on the baseline code that needs to be downloaded from brightspace. Please make sure you read this whole section, as well as the grading guidelines (Section 6), before downloading your code from Brightspace.

2 Build and run xv6

The following provides instructions on how to build and run xv6 using either a CS machine or your own computer.

2.1 Using a CS machine

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

(2) Download and SCP the baseline xv6 code.

(3) Compile and run xv6:

```
$ make qemu-nox
```

After the 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 the following output (disregard the warning message(s)), 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
```

2.2 Using your own computer

You will need a Linux distribution that supports GNU C, which is targeted at the x86 architecture and using ELF binaries.

(1) Install QEMU in your computer. For example, on Ubuntu, just run:

```
$ sudo apt-get install qemu
```

(2) Perform Section 2.1 step (2).

(3) Open the Makefile in the xv6 source directory, and comment out the following line:

```
QEMU = ~boubinjg/fs/bin/qemu-system-i386
```

by putting a "#" at the beginning of the line.

(4) Perform Section 2.1 step (3).

3 Shutdown commands and system calls (60 points)

The original xv6 system doesn't have a "`shutdown`" command to properly turn off the machine. In this part, you are going to add a "shutdown" command to shut down the machine and four other system calls.

3.1 `shutdown` command and the system call

The first command you need complete is named "shutdown", which simply shuts down the machine. In the baseline code, the file "shutdown.c" provides the implementation of this command. What you need to do is to complete the corresponding system call and its user space wrapper routine.

3.2 Additional system calls

Operating systems developers regularly update their operating systems with new features, which may include new system calls. Your group will add **N** additional system calls to xv6 where $\mathbf{N} == GroupSize$. Each group member is responsible for one system call, and can be helped by other gropu members. 5 example system calls are documented below. Select N of these 5, without repetition.

3.2.1 `shutdown2`

The first command you may complete is named "shutdown2", which prints a shutdown message provided by the user **within the system call implementation** before shutting down the machine. In the baseline code, the file "shutdown2.c" provides the implementation of this command. What you need to do is to complete the corresponding system call and its user space wrapper routine.

3.2.2 `exit2`

The second command you may complete is named "exit2", which behaves like the exit() system call and voluntarily exits a running process. exit2 takes an integer exit status argument and prints that status **within the system call implementation** before exiting.

In the baseline code, the file "exit2.c" provides the implementation of this command. What you need to do is to complete the corresponding system call and its user space wrapper routine.

3.2.3 `mkdir2`

The third command you may complete is named "mkdir2", which behaves like the mkdir system call which creates a new directory in your file system. mkdir2 takes two directory names as arguments and creates both directories in your file system.

In the baseline code, the file "mkdir2.c" provides the implementation of this command. What you need to do is to complete the corresponding system call and its user space wrapper routine.

3.2.4 `uptime2`

The fourth command you may complete is named "uptime2", which behaves like the uptime system call which returns the amount of ticks (100ths of a second) your system has been turned on. uptime2 provides takes an integer argument which changes the format of the return value **in kernel space**. uptime2(1) returns uptime in ticks, uptime2(2) returns uptime in seconds, and uptime2(3) returns uptime in minutes.

In the baseline code, the file "uptime2.c" provides the implementation of this command. What you need to do is to complete the corresponding system call and its user space wrapper routine.

3.2.5 `Custom System Call (5 extra credit points)`

If you would like, one group member can complete a custom system call. This system call can be based on an existing system call, or entirely of your own invention. Your custom system call **MUST** take at least one argument of any type and use that argument in its functionality.

To gain credit for this system call, you must implement it and add its user space wrapper for its command similar to the code provided for each of the previous five system calls in this assignment. Please document the purpose, functionality, and usage of this system call in a PROJ1.txt file included with your code.

3.3 Requirements and hints

- You implementation should not change the code in shutdown.c or other user space wrappers other than commenting out the "`STUB_FUNCS`" macro definition to remove the stub function. **Failure to follow this requirement will lead to zero points for each pertinent part of the assignment**.
- To shut down the xv6 virtual machine in your system call, you only need two lines of code:

```
outw(0xB004, 0x0|0x2000);
outw(0x604, 0x0|0x2000);
```

- 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). You may be requested to explain how things work in xv6 in exams.

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

Answer the following questions about system call implementation.

(1) (10 points) What is the name of the (user space) wrapper function of the system call for the `shutdown2` command (2 points)? Where is this wrapper function invoked (3 points)? Where is this function defined (5 points)?

(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 any 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.

5 Submit your work

Once your code is ready for grading, submit it in its entirety as a zip file to the assignment link in Brightspace.

Please also include a PROJ1.txt file explaining how you and your group members worked on this project. Please comment on the work that each of you completed, including which group members completed which syscalls.

You can choose to include the following info in the "PROJ1.txt" document

- The status of your implementation (especially, if not fully complete).
- A description of how your code works, if that is not completely clear by reading the code (note that this should not be necessary, ideally your code should be self-documenting).
- Possibly a log of test cases which work and which don't work.
- Any other material you believe is relevant to the grading of your project.

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

6 Grading

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

(1) Late penalty is 30% of the points scored for each of the first two days late, and will not be graded thereafter.

(2) If you are to compile and run the xv6 system on the department's remote cluster, remember to use baseline xv6 source code provided on brightspace. 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.

(3) If the submitted code 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.

(4) 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.

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