

CS 347m Operating Systems

Spring 2020-21

Lab 2: System calls (with xv6)

1a. Setting up xv6

- Follow the instructions given at www.cse.iitb.ac.in/~puru/courses/spring20/vm-xv6-howto.html to set up the xv6 operating system on your machine.
- Download, read and use as reference the xv6 source code companion book and the xv6 source code at:
<http://www.cse.iitb.ac.in/~puru/courses/spring20/xv6/xv6-rev10.pdf>
<http://www.cse.iitb.ac.in/~puru/courses/spring20/xv6/xv6-book-rev10.pdf>

1b. Getting to know xv6 better

Before you get started with making changes to the xv6 source code, it is advisable to gain a basic understanding of the xv6 OS code. Here we will list out some files from where you can start looking at the xv6 source code.

The basic way a user program communicates with an operating system is via system calls, and an interface to the system call must be provided to the user program defining the **contract** between the user program and OS on the arguments that the OS expects from the user program and the meaning of the return value when the system call completes.

The files **syscall.h**, **syscall.c**, **usys.S** and **user.h** help provide such an interface to the user programs. Take a look at these files to see how it is done.

Also take a look at **sysproc.c**, **proc.h**, and **proc.c**. These are files where the xv6 finally implements functionality relating to the system call.

After executing **make qemu-nox** as given in the instructions to setup xv6, you will see a prompt. The prompt is the xv6 command line interface to execute user level programs. Begin with **ls** and find which will list all the user programs that exist and try executing them.

The source code for all programs is included as part of the xv6 distribution. Look up the implementation of these programs. For example, **cat.c** has the source code for the **cat** program. Execute and lookup the following: **ls**, **cat**, **wc**, **echo**, **grep** etc. Understand how the syntax in some places is different than normal C syntax.

Check the makefile to see how the program `wc` is set up for compilation.

Also, try modifying some of these user programs, for example as a starter exercise, modify the existing shell program `sh.c` in `xv6` to change the shell prompt.

e.g., `turtle$`

This modification will need an update to `sh.c`, compilation and restart of `xv6`.

Note: Part 1 does not carry any grade and is for you to get accustomed to the steps in compiling and running and going through `xv6` source code.

2. Hello `xv6`!

Ready to dive in?

There is no turning back from here. The exciting and adventurous journey inside `xv6` and many more operating systems to follow starts from here. Buckle-up, see you on the other side!

(a) the hello system call

Implement a `xv6` system call, called `hello()`, which outputs the following string

Hi! Welcome to the world of `xv6`!

to the console. You can use the `cprintf` function inside the system call for this. Look at the `xv6` source code to look for example usages of `cprintf`.

Also, a simple C program named `helloxv6.c` has been provided along with the source, and you are supposed to edit this C program to make use of your new `hello()` system call. This user program should effectively have only a single line of code where you are calling the `hello()` system call.

Note:

- You will need to modify a number of different files for this exercise, though the total number of lines of code you will be adding is quite small. At a minimum, you will need to alter `syscall.h`, `syscall.c`, `user.h`, and `usys.S` to implement your new system call.
- The usual C function calls and systems call wrapper used in the user-mode, do not work when implementing a system call or any other functionality as part of the operating system. Analogous functions (e.g., `cprintf`, `kalloc`, `kfree` etc.)

(b) System call tracing

Start with a fresh installation of XV6!!!

In this part you will add a system call tracing feature that may help you when debugging later labs. You'll create a new trace system call that will control tracing. It should take one argument, an integer "mask", whose bits specify which system calls to trace. For example, to trace the fork system call, a program calls `trace(1 << SYS_fork)`, where `SYS_fork` is a syscall number from `kernel/syscall.h`. You have to modify the xv6 kernel to print out a line when each system call is about to return, if the system call's number is set in the mask. The line should contain the process id, the name of the system call and the return value; you don't need to print the system call arguments. The trace system call should enable tracing for the process that calls it and any children that it subsequently forks, but should not affect other processes.

Note: The trace system call must return -1 if anything goes wrong. For example, if the argument to the trace system call is neither of the 3 possible values mentioned below, then the system call must return -1.

We provide a trace user-level program that runs another program with tracing enabled (see `trace.c`). When you're done, you should see output like this:

```
$ trace 32 grep cat hello.txt
3 : syscall read ---> 34
3 : syscall read ---> 0
```

In the example above, trace invokes grep tracing just the read system call. The number 32 is `1<<SYS_read`. To make things easier, you'll only be expected to trace the `read()` and `fork()` system calls. So, the first argument of trace can be either 32 (`1<<SYS_read`), or 2 (`1<<SYS_fork`), or 34 (`1<<SYS_fork & 1<<SYS_read`).

Note:

- Run `make` and you will see that the compiler cannot compile `trace.c`, because the user-space stubs for the system call don't exist yet: add a prototype for the system call as you had done to add the `hello()` system call in the previous part. To reiterate, compilation will fail until you add both the prototype and implementation of the `trace()` system call.

Hints:

- Add a variable to the process state indicating trace mask, which tells the kernel which system calls to trace for this process. (which is nothing but the integer mask value passed in the trace system call)
- You need to modify the fork system call to copy this trace mask along with the other parent process state while creating the child process.
- Just before the fork and read system call returns, you need to add code to print required tracing output if the system call is being traced by the current process.

3. Submission guidelines

- The Programming assignment resources provided to you will contain two folders named p_2a and p_2b. p_2a contains patch files required for part 2a of this assignment and p_2b contains patch files for part 2b of this assignment. You are supposed to copy-paste these patch files in the xv6 source which you will download following the instruction in part 1a. One of the patch files will be a makefile which you'll have to replace with the original makefile present in the xv6 source. To reiterate, you have to replace the original Makefile in the xv6 source with the Makefile provided to you, otherwise the updated source with the system call won't compile.

So, to do part 2a of this assignment you'll download the xv6 source and then add the patch files given in p_2a folder to the downloaded xv6 source. Now, the xv6 source won't compile until you complete your system call implementation. Then you'll proceed on to implementing the system call and after having done so, you'll verify the implementation.

You'll have to do something similar to do part 2b of this assignment.

The directory structure for Lab 2 resource folder will look like this:

```
[lab2-resources]
|
+-- [p_2a]
|   helloxv6.c
|   hello.txt
|   Makefile
+-- [p_2b]
|   trace.c
|   hello.txt
|   Makefile
```

- We suspect that for both part 2a and 2b of this assignment, you will only have to edit 7 files in the xv6 source, and those 7 files will be **syscall.c**, **syscall.h**, **user.h**, **usys.S**, **sysproc.c**, **proc.c** and **proc.h**.
Submit these 7 (or more) source files as part of your submission **separately** for part 2a and part 2b of this assignment.

Submission folder format

```
[roll_number-lab2]
|
|  README
|  HonorCode.txt
|-- [p_2a]
|   |
|   |  syscall.c
|   |  syscall.h
|   |  user.h
|   |  usys.S
|   |  sysproc.c
|   |  proc.c
|   |  proc.h
|-- [p_2b]
|   |
|   |  syscall.c
|   |  syscall.h
|   |  user.h
|   |  usys.S
|   |  sysproc.c
|   |  proc.c
|   |  proc.h
```

- The README file should contain the specifics of how you implemented both the system calls.
- Add the honor code text (given on the next page) in HonorCode.txt
The 7 files in p_2a should reflect the changes done to the xv6 source to complete part 2a of the assignment.
Similarly, 7 files in p_2b should reflect the changes done to the xv6 source to complete part 2b of the assignment.

- After you are done with the assignment and have finished editing both xv6 sources, compress the resources folder using the following command (Your actual roll number does in place of [roll_number])

`tar -cvf [roll_number-lab2].tar.gz [roll_number-lab2]`

Example command will look like this:

`tar -cvf 194050001-lab2.tar.gz 194050001-lab2`

- Submit the compressed tar file to Moodle.

***** IMPORTANT *****

Please cross-check that your tar file has all the necessary files in it (and that they are the ones you want to submit) before submitting. You can check it by extracting the tar file by doing the following:

`tar -xvf [roll_number-lab2].tar.gz`

Cribs related to code changes, compilation errors, missing files, corrupt files etc. will not be entertained after the deadline has passed.

Honor Code:

The following text needs to be added in HonorCode.txt

I, _____, confirm that my submission has no material, in total or in parts, that is plagiarized or copied. All the work (documents, code) that I am submitting is my own and is prepared and written by me alone. None of the submission is copy in full or in parts of any submission that does not belong to and has been prepared by me.

Note: Discussions with peers and study of material from books and online resources is acceptable and encouraged.