Department of Computer Science and Engineering
Indian Institute of Technology Bombay

# CS 333: Operating Systems Lab
# Autumn 2022
# Lab 4: xv6 inside-out

---

## Goal

In this lab we will learn how to use the xv6 operating system, implement our own system calls and explore examples of OS states.

## Task 0: Setting up xv6

Follow the instructions given **here** for xv6 installation. xv6 runs on an x86 emulator called QEMU that emulates x86 hardware on your local machine. In the xv6 folder, run the following command sequence ..

```
make
make qemu
```
or `make-qemu-nox`

to boot xv6 and open a shell.

I. At the shell try out a few shell commands starting with `ls` to find a list of available programs and then try and execute a few of them.

II. Look up the implementation of these programs. For example, `cat.c` is 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 from normal C syntax.

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

**make qemu**
Build everything and start qemu with the VGA console in a new window and the serial console in your terminal. To exit, either close the VGA window or press Ctrl-c or Ctrl-a x in your terminal.
**make qemu-nox**
Like make qemu, but run with only the serial console. To exit, press Ctrl-a x. This is particularly useful over SSH connections.

## Task 1: Adding new programs to xv6

**(a) Display first n lines of a file**

Copy the existing `cat.c` program of xv6 to a file `head.c` and modify it to print the first **n** lines of a file to the terminal. The command should handle multiple filenames as input, the first argument is always the number of lines from start of file to print. Also, look at the `cat.c` program for the error message if the file passed through argument does not exist.

**Note:**
(i) Will need to make additions to the Makefile to add new programs for compilation and also to include as part of the xv6 viewable disk image (to read/write files e.g., abc.txt, hello.txt).
To do this, look for the following keywords in the makefile.

> **"UPROGS=\":** List names of all user programs which are available after xv6 boot up.
>
> **"EXTRA=\":** List of all files (source programs and other scripts and data files) available after xv6 bootup.
>
> **"fs.img"**: List of files to be added to the xv6 startup disk (imagefile).

(ii) The xv6 OS itself does not have any text editor or compiler support, so you must write and compile the code in your host machine, and then run the executable in the xv6 QEMU emulator.

Sample output:

```
$ head 3 README abc.txt hello.txt
-----README-----
NOTE: we have stopped maintaining the x86 version of xv6, and switched
our efforts to the RISC-V version
(https://github.com/mit-pdos/xv6-riscv.git)
-----abc.txt-----
abc
asdf
lkjl
-----hello.txt-----
hello
bye
$
```

**(b) shell in a shell**

Write a program `cmd.c` that creates a child process, child process executes a program, and `parent` waits till completion of the child process before terminating. This program should use the `fork` and exec system calls of xv6. The program to be executed by the child process can be any of the simple xv6 programs and should be specified at the command line.

Sample output:

```
$ cmd ls
.              1 1 512
..             1 1 512
README         2 2 2286
hello.txt      2 3 10
abc.txt        2 4 43
cat            2 5 16288
echo           2 6 15140
forktest       2 7 9452
```

```
init: starting sh
$ cmd echo Welcome to xv6
Welcome to xv6
$
```

## Task 2: Adding new system calls to xv6

To understand and work with system calls and process related information and action, the following files of the xv6 OS are important:

`usys.S, user.h, defs.h, sysproc.c, syscall.h, syscall.c, proc.h, proc.c.`

- `user.h` contains the system call definitions in xv6.
- `usys.S` contains a list of system calls exported by the kernel, and the corresponding invocation of the trap instruction.
- `syscall.h` contains a mapping from system call name to system call number. Every system call must have a number assigned here.
- `syscall.c` contains helper functions to parse system call arguments, and pointers to the actual system call implementations.
- `sysproc.c` contains the implementations of process related system calls.
- `defs.h` is a header file with function definitions in the kernel.

- `proc.h` contains the `struct proc` structure.
- `proc.c` contains implementations of various process related system calls, and the scheduler function. This file also contains the definition of `ptable`, and several examples of functions traversing/using the process list.

All or most of these files will have to be used/updated to implement new system calls.
New files, new programs, new data files need to be added to xv6 via the xv6 Makefile.
All changes (updates/new files) are to be followed by a clean compile, followed by executing xv6.

Note that the xv6 OS itself does not have any text editor or compiler support, so you must write and compile the code in your host machine, and then run the executable in the xv6 QEMU emulator.

**(a) system call first look!**

Modify the `kill()` system call to print the PID of the process it kills.
You should use `cprintf()` inside the `kill()` system call to print the process PID. in proc.c
A simple test program `test-mod-kill.c` is also provided you can use it to test your implementation.

Sample Output:

```
$ test-mod-kill
Child Pid - 4
Parent Pid - 3
Process with pid-'4' killed
$z ombie!

$ _
```

*** 7 files modified ***

proc.c -- helloYou defn // can be kalloc.c, etc...
defs.h -- void helloYou(char*)
syscall.c --
  extern int sys_helloYou(void)
  [SYS_helloYou] sys_helloYou
syscall.h -- #define SYS_helloYou 22
sysproc.c -- int sys_helloYou(void)
user.h -- void helloYou(char*)
usys.S -- SYSCALL(helloYou)

**(b)** the **helloYou** system call

Implement a system call, with the following declaration `helloYou(char* name)`, which prints the string name to the console. The function `cprintf` if used for printing in the kernel mode.
A simple test program `test-helloworld.c` is also provided to test your implementation.

**Note**: Look up the helper functions **argint**, **argstr**, **argptr** for arguments of system calls. also argfd used here: https://stackoverflow.com/questions/53383938/pass-struct-to-xv6-system-call

**(c) More process information**

Implement the following system calls:

`getNumProc()` which returns the total number of processes in the system (either in embryo, running, runnable, sleeping, or zombie states).

`getMaxPid()` that returns the maximum PID amongst the PIDs of all currently active processes in the system. <=> all that're not UNUSED

Use the test program `test-getprocesses.c` to test your implementation.

Sample Output:

```
$ test-getprocesses
Total Number of Processes: 3
Maximum PID: 3
---After single fork---
Total Number of Processes: 4
Maximum PID: 5
---After multiple fork---
Total Number of Processes: 7
Maximum PID: 7
```

### (d) More system calls (Optional)

struct proc [proc.h] has: struct file *ofile[NOFILE]; // Open files

- **int numOpenFiles()** – Implement a system call to report the number of open file descriptors used by the current process. You can write a test program to test your implementation.

  Hint : Look at the struct proc in the proc.h file and observe how other system calls retrieve information about the current process. proc.c

I have counted both switch-in and switch-out [+1 for each]

- **int csinfo()** – Implement a system call to return the number of context switches that took place in the process from the time it started.

  A simple test program `test_csinfo.c` is provided, you can use it to test your implementation.

  **Hint:** Implementing this will require keeping an additional counter in the per process PCB objects—`struct proc` of xv6.

I've initialized num_...=0 in fork(), but seems better to do that in allocproc() -- bcoz userinit() and fork() are the only functions that call allocproc() -- and if num_...=0 in fork(), then num_... ~ uninitialized for init-process

- **get_sibling()** – Implement a system call to print the details of siblings of the calling process to the console. The output should be in the format of:
  <pid> <process status>
  <pid> <process status>
  ….

  **Sample output:** [when run just after bootup]
  **$** my_siblings 6 1 2 1 0 2 0
  4 RUNNABLE
  5 ZOMBIE
  6 RUNNABLE
  7 SLEEPING
  8 ZOMBIE
  9 SLEEPING

  Sample user program `my_siblings.c` and a sample output file `output_my_sibling.txt` . i is provided.
  The program takes an integer n, followed by a combination of 0, 1 and 2 of length n, as command line arguments— 0/1/2 specify the process state of the **n** child processes. The (n+1)th child process executes the **get_sibling()** system call and displays the output.

  **Hints**: You need to find the process ID of the calling process, and process ID of its parent and traverse all the PCBs and compare their parent PID with the parent of the calling process.

ptable [lock...]

- Extend the **get_sibling** call to accept a PID as an argument, to perform the same task (as of get_sibling) on the specified PID.

## Submission instructions:

- All submissions via moodle. Name your submissions as: `<rollnumber>_lab4.tar.gz`
  `(e.g. 21q050003_lab4.tar.gz)`
- The tar should contain the following files in the specified directory structure:

```
<rollnumber>_lab4/
        |____ head.c
        |____ cmd.c
        |____ <all modified files in xv6 such as proc.c, syscall.c, syscall.h,
        sysproc.c, defs.h, user.h, and usys.S>
        |____ <any new files added to the xv6 img>
        |____ Makefile
```

  **Please adhere to this format strictly.**

- Command to tar your submission directory …
  `tar -zcvf <rollnumber>_lab4.tar.gz <rollnumber>_lab4`

- **Due date: 29th August 2022, 5.15 pm**