CS 333: Operating Systems Lab
Autumn 2023
Lab4: accio, xv6!

In this lab we will learn how to use the xv6 operating system, learn how system calls are implemented and explore examples of OS metadata and actions.

Task 0: Setting up xv6

Follow the instructions given below for xv6 installation.
Run the following commands to get the xv6 source (if you are using lab machine)

- `wget https://www.cse.iitb.ac.in/~puru/courses/xv6-public.tar.gz`
- `tar -xf xv6-public.tar.gz`
- `cd xv6-public`
- `make`

If you are using a Linux environment on a personal machine, you will need a set of other tools as well for xv6 … use the following commands to install required packages.

- `sudo apt-get update`
- `sudo apt -y install build-essential gdb coreutils util-linux sysstat procps wget tar qemu`

The booklet describing/listing all source files is available **here**. (also after make in the xv6 dir)

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 to boot xv6 on an emulated machine. QEMU boots the machine and if all goes well drops to a user space shell program.

- **`make clean`**
- **`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.

  Ctrl+A X ⇒ First press Ctrl + A (A is just key a, not the alt key), 2. then release the keys and press X.

- At the shell start with **ls** to list available programs and then execute a few of them.
- 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.
- Check the makefile to see how the program wc is set up for compilation.

Hola, fellow OS enthusiasts! We are absolutely stoked that we have the xv6 setup ready. It's time to roll up our sleeves, dive into the xv6 code, and get our hands dirty. Our journey begins with the task of adding a simple program to xv6, and then we'll take it up a notch by introducing some brand new system calls. This is where the real fun begins! 🚀 🤓

Task 1: Adding new programs to the xv6 environment

(a) pingpong with xv6

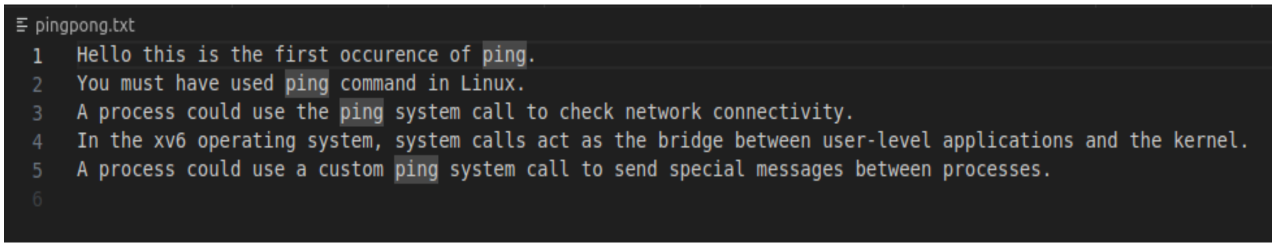
Write a program named **pingpong** which reads a text file as an input argument and outputs "**pong**" to the standard output every time it finds the word **"ping"** in the input text file.

I. Additions to the Makefile will be needed to add new programs for compilation and also to be included as part of the xv6 viewable disk image (to read/write files e.g., abc.txt, hello.txt) via fs.img.
Look for the following keywords in the makefile.
`UPROGS=\`
Lists 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. xv6 OS itself does not have a text editor or compiler support, all source code of programs has to be written and compiled on the host machine, all its references added to the makefile and then via fs.img and xv6.img be used via QEMU emulator.

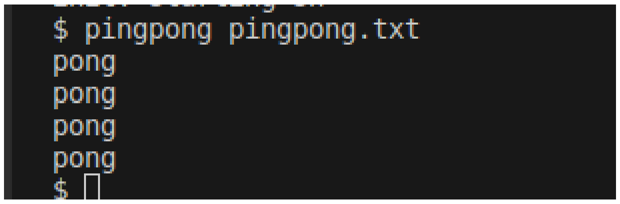
III. You will need to include input text files for e.g., "pingpong.txt" or any other files in the xv6 OS image that you will use for running the program.
Refer to the README included in xv6 image.

IV. Sample input file **pingpong.txt** is provided as part of this lab archive file.
Consider using **wc.c**, source code of the **wc** program as a starting point for this task.



```
pingpong.txt
Hello this is the first occurence of ping.
You must have used ping command in Linux.
A process could use the ping system call to check network connectivity.
In the xv6 operating system, system calls act as the bridge between user-level applications and the kernel.
A process could use a custom ping system call to send special messages between processes.
```

Sample usage

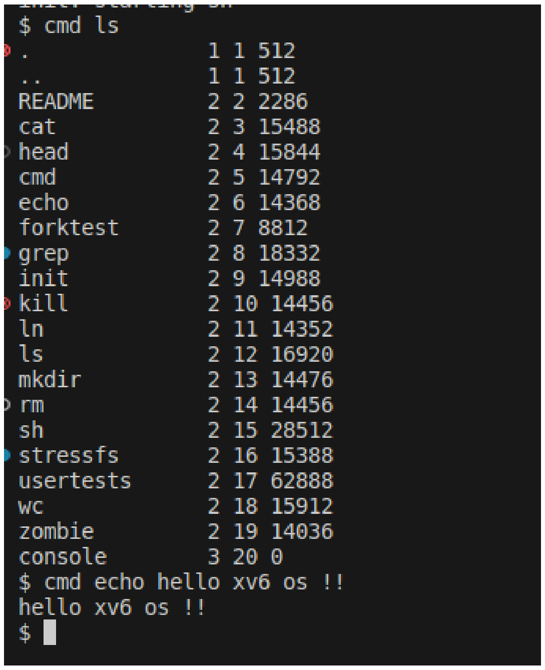


```
$ pingpong pingpong.txt
pong
pong
pong
pong
$ 
```

(b) inception (shell in a shell)

Write a program **`cmd.c`** that creates a child process — the child process executes a program, and the parent process 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 sample xv6 programs and should be specified at the command line.
Refer to Sheet 66,85 of xv6 source code booklet for `fork()`, `exec()` system calls in xv6.

Sample usage



```
$ cmd ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15488
head           2 4 15844
cmd            2 5 14792
echo           2 6 14368
forktest       2 7 8812
grep           2 8 18332
init           2 9 14988
kill           2 10 14456
ln             2 11 14352
ls             2 12 16920
mkdir          2 13 14476
rm             2 14 14456
sh             2 15 28512
stressfs       2 16 15388
usertests      2 17 62888
wc             2 18 15912
zombie         2 19 14036
console        3 20 0
$ cmd echo hello xv6 os !!
hello xv6 os !!
$ 
```

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 xv6 system call declarations
- **usys.S** contains a list of system calls exported by the kernel, and the corresponding invocation of the trap instruction
- **syscall.h** contains the mapping of system call name to system call number
- **syscall.c** contains helper functions to handle the system call entry, parse 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 declarations in the xv6 kernel
- **proc.h** contains the process abstraction related variable definitions
- **proc.c** contains implementations of various process related system calls, functions and the scheduler, also contains the declaration of ptable, and several examples of functions traversing/using the process list
- System call related functions are also listed in **sysfile.c**

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 are to be followed by a clean compile and build, followed by executing xv6.

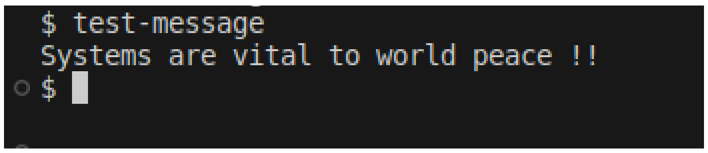
Note that xv6 itself does not have a text editor or compiler support, so all xv6 source code changes are on the host machine then **xv6.img** and **fs.img** are used as inputs to QEMU.

(a) hello, system calls!

Implement a system call, with the following declaration **worldpeace(),** which prints the message "*Systems are vital to world peace !!*" in the kernel mode.

The function **cprintf** is used for printing in the kernel mode (refer to sheet 30 line 3026 of **xv6 source code booklet** for usage).
A simple test program **worldpeace.c** is also provided to test your implementation.
ChatGPT's take on systems + world peace is here.

Sample usage



```
$ test-message
Systems are vital to world peace !!
$ 
```

(b) who all are ready?

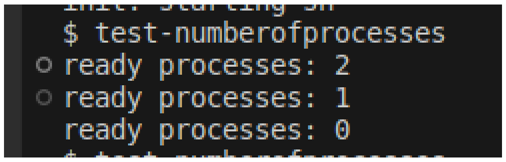
Implement a system call, with the following declaration **numberofprocesses()** which returns the total number of processes in **READY**(xv6 naming is **RUNNABLE**) state to the user program.

Refer to the **PCB** structure defined on line 2336 sheet 23 of [xv6 source code booklet]() and lines 10-14 in **proc.c** in the given source code to refer to **struct ptable**.

Refer to sheet 24 Line 2480 to understand how to iterate through the process table and sheet 23 Line 2334 to for the process state enum of [xv6 source code booklet]() .

A simple test program **nump.c** is also provided to test your implementation.

Sample usage



```
$ test-numberofprocesses
ready processes: 2
ready processes: 1
ready processes: 0
```

(c) status check

Implement a system call, with the following declaration **whatsthestatus(int pid),** which returns the **parent pid**, prints the **name of the parent process** and the **current state of the process** given the pid of the process.

The function **cprintf** is used for printing in the kernel mode (refer to sheet 30 line 3026 of [xv6 source code booklet]() for usage).

Please refer to the PCB structure defined on line 2336 sheet 23 of [xv6 source code booklet]() and lines 10-14 in proc.c in the given source code to refer to **ptable** struct.

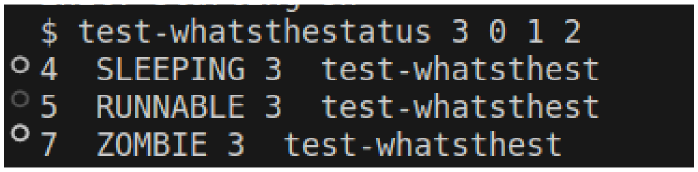
Refer to sheet 24 Line 2480 to understand how to iterate through the process table and sheet 23 Line 2334 to understand the process enum structure of [xv6 source code booklet]() .

Refer to sheet 36 line 3631-3632 to refer **argint** usage of [xv6 source code booklet]()
Note: **argint**, **argstr**, **argptr** are helper functions for handling system calls arguments.

A simple test program **status.c** is also provided to test your implementation.
Input should be in the format **status 3 0 1 2** when "3" denotes the number of children to fork and 0 signifies **Sleeping**, 1 signifies **Runnable** and 2 signifies **Zombie** states**.**

Output should be in the format **<pid> <status> <ppid> <parent_name>** where **pid** represents the pid of the child process for which status needs to be checked, **ppid** represents the parent pid and **parent_name** represents the name of the parent.

Sample usage



```
 $ test-whatsthestatus 3 0 1 2
○4  SLEEPING 3  test-whatsthest
○5  RUNNABLE 3  test-whatsthest
○7  ZOMBIE 3  test-whatsthest
```

(d) spawn — one call, many processes!

Implement a system call, with the following declaration `int spawn(int n, int* pids)` which creates **n** child processes with a single system call.

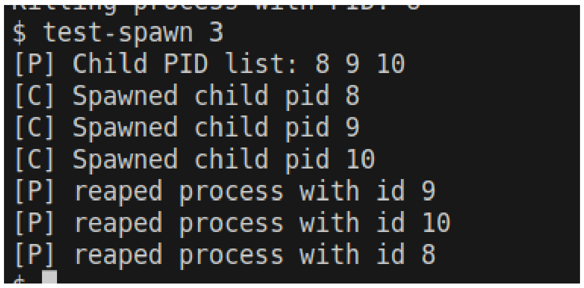
- The system call must return 0 to the child processes and number of children created to the parent process.
- Additionally, the **pids** array should contain the pids of the spawned children after the **spawn** system call. The parent should gracefully reap all the child processes which are present in the **pids** array.

A simple test program **spawn.c** is also provided to test your implementation.

Note: **argint**, **argstr**, **argptr** are helper functions for handling system calls arguments. (Refer to sheet 65 line 6557 of [xv6 source code booklet]() for **argptr** usage)

Refer to **fork()** system call implementation in proc.c to understand how a child process is created and how the call handles return values for parent and child processes.

Sample usage



```
$ test-spawn 3
[P] Child PID list: 8 9 10
[C] Spawned child pid 8
[C] Spawned child pid 9
[C] Spawned child pid 10
[P] reaped process with id 9
[P] reaped process with id 10
[P] reaped process with id 8
```

Submission Instructions

- All submissions have to be done via moodle. Name your submission as <rollnumber_lab4>.tar.gz

  (e.g 190050096_lab4.tar.gz)

- The tar should contain the xv6 source code files in the following directory structure: **<rollnumber_lab4>/xv6**
- The directory must contain the new programs **pingpong.c** and **cmd.c** which you will implement as a part of 1a and 1b respectively and the entire xv6 source code with necessary changes to it.
- Your code should be well commented and readable.
- tar -czvf <rollnumber_lab4>.tar.gz <rollnumber_lab4>

Deadline: Thursday 31st August 2023 5:00 PM via moodle.

More Exercises (Optional)

1. you got siblings?

Implement a system call **`int get_sibling()`**
to print the details of siblings of the calling process to the console and to return the number of siblings of the calling process. A sibling is all processes with the same parent process.
The output should be in the format of:
<pid> <process status>
<pid> <process status>
….

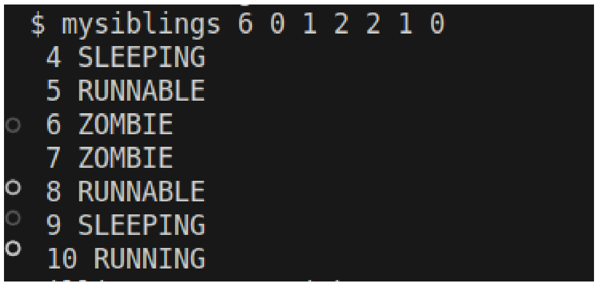
Sample usage

```
$ 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** 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.

Hint: 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.

Sample usage



```
$ mysiblings 6 0 1 2 2 1 0
 4 SLEEPING
 5 RUNNABLE
 6 ZOMBIE
 7 ZOMBIE
 8 RUNNABLE
 9 SLEEPING
 10 RUNNING
```

2. entrypoint 2.0

Implement a new type of **system_call_handler** which instead of handling TRAP NUMBER 64 handles trap number say 65. You should implement a new type of system call **fork2()** that uses a different trap number (**65**) instead of the commonly used trap number 64 (which corresponds to the traditional `int $0x64` instruction for making system calls in x86 assembly). Using a different trap number, such as 65 in this exercise, allows you to define and handle your custom system calls independently from the standard ones.

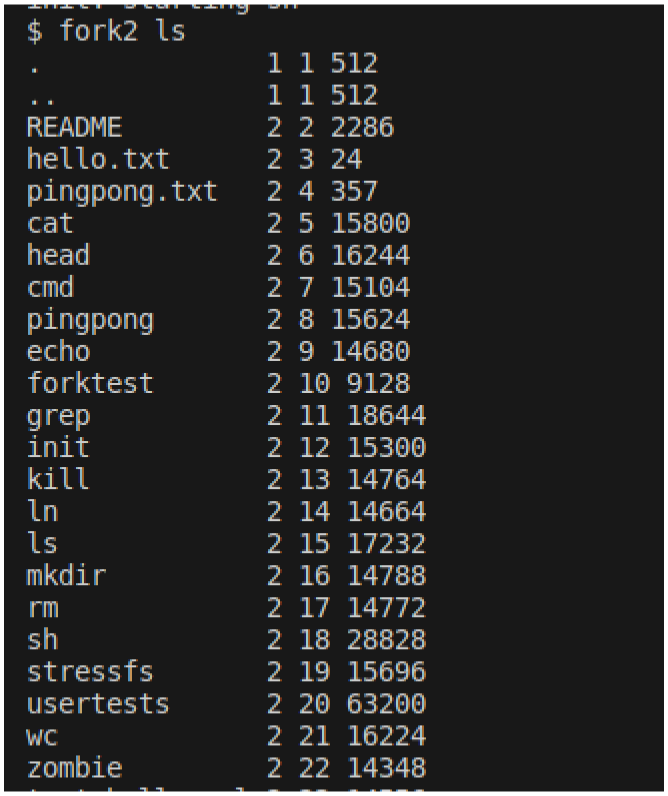
In order to achieve this you should first need to look at how system calls are handled in xv6. List of files you will need to refer to:
sysc.all.c syscall.h defs.h user.h proc.c sysproc.c proc.h trap.c trap.h usys.S

Note: First you need to write a trap handler and after that you can implement a new type of system call. **(TRAP NUMBER for SYSCALL is 64 but the actual system call number of system call like fork in xv6 is 1).**

Refer to sheet 32 33 and 34 of xv6 source code booklet

Hint: usys.s has the entry point from where int n is called. :)
One way to implement this is to change only the entry point to the trap handler and use the same underlying system call implementation for all system calls.
A simple test program **userfork2.c** is provided to test your implementation.

Sample usage:



```
init: starting sh
$ fork2 ls
.              1 1 512
..             1 1 512
README         2 2 2286
hello.txt      2 3 24
pingpong.txt   2 4 357
cat            2 5 15800
head           2 6 16244
cmd            2 7 15104
pingpong       2 8 15624
echo           2 9 14680
forktest       2 10 9128
grep           2 11 18644
init           2 12 15300
kill           2 13 14764
ln             2 14 14664
ls             2 15 17232
mkdir          2 16 14788
rm             2 17 14772
sh             2 18 28828
stressfs       2 19 15696
usertests      2 20 63200
wc             2 21 16224
zombie         2 22 14348
```