OPERATING SYSTEM

PROJECT 1 – Xv6 and Unix utilities

1. General rule:

- The project is done in groups: each group has a maximum of **2 students**

- **The same exercises will all be scored 0 for the entire practice (even though there are scores for other exercises and practice projects).**

- Môi trường lập trình: **Linux**

2. Submission:

Submit assignments directly on the course website (MOODLE), not accepting submissions via email or other forms.

Filename: **StudentID1_StudentID2.zip** (with StudentID1 < StudentID2)

Ex: Your group has 2 students: 2312001 and 2312002, the filename is: **2312001_2312002.zip**

Include:

- **StudentID1_StudentID2_Report.pdf:** Writeups should be short and sweet. Do not spend too much effort or include your source code on your writeups. The purpose of the report is to give you an opportunity to clarify your solution, any problems with your work, and to add information that may be useful in grading. If you had specific problems or issues, approaches you tried that didn't work, or concepts that were not fully implemented, then an explanation in your report may help us to assign partial credit
- **Release**: File diff (diff patch, Ex: $ git diff > <MSSV1>_<MSSV2>.patch)
- **Source**: Zip file of xv6 (the version is made clean)

Lưu ý: Cần thực hiện đúng các yêu cầu trên, nếu không, bài làm sẽ không được chấm.

3. Demo Interviews

Your implementation is graded on completeness, correctness, programming style, thoroughness of testing, your solution, and code understanding.

When administering this course, we do our best to give a fair assessment to each individual based on each person's contribution to the project

4. Requirements

This lab will familiarize you with xv6 and its system calls.

1. sleep

Implement a user-level sleep program for xv6, along the lines of the UNIX sleep command. Your sleep should pause for a user-specified number of ticks. A tick is a notion of time defined by the xv6 kernel, namely the time between two interrupts from the timer chip. Your solution should be in the file user/sleep.c.

Some hints:

- Before you start coding, read Chapter 1 of the xv6 book.
- Put your code in ***user/sleep.c***. Look at some of the other programs in user/ (e.g., user/echo.c, user/grep.c, and user/rm.c) to see how command-line arguments are passed to a program.
- Add your sleep program to **UPROGS** in ***Makefile***; once you've done that, make qemu will compile your program and you'll be able to run it from the xv6 shell.
- If the user forgets to pass an argument, sleep should print an error message.
- The command-line argument is passed as a string; you can convert it to an integer using atoi (see user/ulib.c).
- Use the system call sleep.
- See kernel/sysproc.c for the xv6 kernel code that implements the sleep system call (look for sys_sleep), user/user.h for the C definition of sleep callable from a user program, and user/usys.S for the assembler code that jumps from user code into the kernel for sleep.
- sleep's main should call exit(0) when it is done.
- Look at Kernighan and Ritchie's book *The C programming language (second edition)* (K&R) to learn about C.

Run the program from the xv6 shell:

```
$ make qemu
...
init: starting sh
$ sleep 10
(nothing happens for a little while)
$
```

Your program should pause when run as shown above. Run make grade in your command line (outside of qemu) to see if you pass the sleep tests.

Note that make grade runs all tests, including the ones for the tasks below. If you want to run the grade tests for one task, type:

```
$ ./grade-lab-util sleep
```

This will run the grade tests that match "sleep". Or, you can type:

```
$ make GRADEFLAGS=sleep grade
```

which does the same.

2. pingpong

Write a user-level program that uses xv6 system calls to "ping-pong" a byte between two processes over a pair of pipes, one for each direction. The parent should send a byte to the child; the child should print "**<pid>: received ping**", where <**pid**> is its **process ID**, write the byte on the pipe to the parent, and exit; the parent should read the byte from the child, print "**<pid>: received pong**", and exit. Your solution should be in the file ***user/pingpong.c***.

Some hints:

- Add the program to **UPROGS** in ***Makefile***.
- You'll need to use the **pipe**, **fork**, **write**, **read**, and **getpid** system calls.
- User programs on xv6 have a limited set of library functions available to them. You can see the list in ***user/user.h***; the source (other than for system calls) is in ***user/ulib.c***, ***user/printf.c***, and ***user/umalloc.c***.

Run the program from the xv6 shell and it should produce the following output:

```
$ make qemu
...
init: starting sh
$ pingpong
4: received ping
3: received pong
$
```

Your program should exchange a byte between two processes and produces output as shown above. Run make grade to check.

3. primes

Write a concurrent prime sieve program for xv6 using pipes and the design illustrated in the picture halfway down this page and the surrounding text. This idea is due to Doug McIlroy, inventor of Unix pipes. Your solution should be in the file user/primes.c.

Your goal is to use pipe and fork to set up the pipeline. The first process feeds the numbers 2 through 280 into the pipeline. For each prime number, you will arrange to create one process that reads from its left neighbor over a pipe and writes to its right neighbor over another pipe. Since xv6 has a limited number of file descriptors and processes, the first process can stop at 280.

Some hints:

- Be careful to close file descriptors that a process doesn't need, because otherwise your program will run xv6 out of resources before the first process reaches 280.
- Once the first process reaches 280, it should wait until the entire pipeline terminates, including all children, grandchildren, &c. Thus the main primes process should only exit after all the output has been printed, and after all the other primes processes have exited.
- Hint: read returns zero when the write-side of a pipe is closed.
- It's simplest to directly write 32-bit (4-byte) ints to the pipes, rather than using formatted ASCII I/O.
- You should create the processes in the pipeline only as they are needed.
- Add the program to UPROGS in Makefile.
- If you get an infinite recursion error from the compiler for the function primes, you may have to declare void primes(int) __attribute__((noreturn)); to indicate that primes doesn't return.

Your solution should implement a pipe-based sieve and produce the following output:

```
$ make qemu
...
init: starting sh
$ primes
prime 2
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
prime 23
```

```
prime 29
prime 31
...
$
```

4. find

Write a simple version of the UNIX find program for xv6: find all the files in a directory tree with a specific name. Your solution should be in the file user/find.c.

Some hints:

- Look at user/ls.c to see how to read directories.
- Use recursion to allow find to descend into sub-directories.
- Don't recurse into "." and "..".
- Changes to the file system persist across runs of qemu; to get a clean file system run make clean and then make qemu.
- You'll need to use C strings. Have a look at K&R (the C book), for example Section 5.5.
- Note that == does not compare strings like in Python. Use strcmp() instead.
- Add the program to UPROGS in Makefile.

Your solution should produce the following output (when the file system contains the files b, a/b and a/aa/b):

```
$ make qemu
...
init: starting sh
$ echo > b
$ mkdir a
$ echo > a/b
$ mkdir a/aa
$ echo > a/aa/b
$ find . b
./b
./a/b
./a/aa/b
$
```

Run make grade to see what our tests think.

5. xargs

Write a simple version of the UNIX xargs program for xv6: its arguments describe a command to run, it reads lines from the standard input, and it runs the command for each line, appending the line to the command's arguments. Your solution should be in the file user/xargs.c.

The following example illustrates xarg's behavior:

```
$ echo hello too | xargs echo bye
 bye hello too
$
```

Note that the command here is "echo bye" and the additional arguments are "hello too", making the command "echo bye hello too", which outputs "bye hello too".

Please note that xargs on UNIX makes an optimization where it will feed more than one argument to the command at a time. We don't expect you to make this optimization. To make xargs on UNIX behave the way we want it to for this lab, please run it with the -n option set to 1. For instance

```
$ (echo 1 ; echo 2) | xargs -n 1 echo
1
2
$
```

Some hints:

- Use fork and exec to invoke the command on each line of input. Use wait in the parent to wait for the child to complete the command.
- To read individual lines of input, read a character at a time until a newline ('\n') appears.
- kernel/param.h declares MAXARG, which may be useful if you need to declare an argv array.
- Add the program to UPROGS in Makefile.
- Changes to the file system persist across runs of qemu; to get a clean file system run make clean and then make qemu.

xargs, find, and grep combine well:

```
$ find . b | xargs grep hello
```

will run "grep hello" on each file named b in the directories below ".".

To test your solution for xargs, run the shell script xargstest.sh. Your solution should produce the following output:

```
$ make qemu
...
init: starting sh
$ sh < xargstest.sh
$ $ $ $ $ $ hello
hello
hello
$ $
```

You may have to go back and fix bugs in your find program. The output has many $ because the xv6 shell doesn't realize it is processing commands from a file instead of from the console, and prints a $ for each command in the file.

6. Grade:

No.	Exercise	Grade
1	sleep	2
2	pingpong	2
3	primes	2
4	find	2
5	xargs	2

5. Reference:

https://pdos.csail.mit.edu/6.1810/2024/labs/util.html