

# Lab: Xv6 and Unix utilities

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

## Boot xv6 ([easy](#))

You can do these labs on your own computer. To setup your computer look at the [lab tools page](#) for tips.

Fetch the git repository for the xv6 source for the lab:

```
$ git clone https://github.com/freeuni-05-fall23/labs
Cloning into 'labs'...
...
$ cd labs
```

The files you will need for this and subsequent lab assignments are distributed using the [Git](#) version control system. For each of the labs you will check out a version of xv6 tailored for that lab. To learn more about Git, take a look at the [Git user's manual](#), or, you may find this [CS-oriented overview of Git](#) useful. Git allows you to keep track of the changes you make to the code. For example, if you are finished with one of the exercises, and want to checkpoint your progress, you can *commit* your changes by running:

```
$ git commit -am 'my solution for util lab exercise 1'
Created commit 60d2135: my solution for util lab exercise 1
1 files changed, 1 insertions(+), 0 deletions(-)
$
```

You can keep track of your changes by using the `git diff` command. Running `git diff` will display the changes to your code since your last commit, and `git diff origin/util` will display the changes relative to the initial util code. Here, `origin/util` is the name of the git branch for this lab.

Build and run xv6:

```
$ make qemu
riscv64-unknown-elf-gcc -c -o kernel/entry.o kernel/entry.S
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_UTIL -MD -mcmodel=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -
...
riscv64-unknown-elf-ld -z max-page-size=4096 -N -e main -Ttext 0 -o user/_zombie user/zombie.o user/ulib.o user/usys.o user/printf.o user/umalloc.o
riscv64-unknown-elf-objdump -S user/_zombie > user/zombie.asm
riscv64-unknown-elf-objdump -t user/_zombie | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > user/zombie.sym
mkfs/mkfs fs.img README user/xargstest.sh user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_ln user/_ls user/_mkdir user/_rm user
nmeta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) blocks 954 total 1000
ballocc: first 591 blocks have been allocated
ballocc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk
```

xv6 kernel is booting

```
hart 2 starting
hart 1 starting
init: starting sh
$
```

If you type `ls` at the prompt, you should see output similar to the following:

```
$ ls
.          1 1 1024
..         1 1 1024
README    2 2 2227
xargstest.sh 2 3 93
cat        2 4 32864
echo       2 5 31720
forktest   2 6 15856
grep       2 7 36240
init       2 8 32216
kill       2 9 31680
ln         2 10 31504
ls         2 11 34808
mkdir      2 12 31736
rm         2 13 31720
sh         2 14 54168
stressfs   2 15 32608
usertests  2 16 178800
grind      2 17 47528
wc         2 18 33816
zombie     2 19 31080
console    3 20 0
```

These are the files that `mkfs` includes in the initial file system; most are programs you can run. You just ran one of them: `ls`.

xv6 has no `ps` command, but, if you type `Ctrl-p`, the kernel will print information about each process. If you try it now, you'll see two lines: one for `init`, and one for `sh`.

To quit `qemu` type: `Ctrl-a x` (press `Ctrl` and `a` at the same time, followed by `x`).

## Grading and hand-in procedure

You can run `make grade` to test your solutions with the grading program. The TAs will use the same grading program to assign your lab submission a grade. Separately, we will also have check-off meetings for labs (see [Grading policy](#)).

The lab code comes with GNU Make rules to make submission easier. After committing your final changes to the lab, type `make tarball` to create the archive for submission. For detailed instructions on how to submit see [below](#).

## sleep (**easy**)

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 solution is correct if your program pauses when run as shown above. Run `make grade` to see if you indeed pass the sleep tests.

Note that `make grade` runs all tests, including the ones for the assignments below. If you want to run the grade tests for one assignment, 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.

## pingpong (**easy**)

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`.
- Use `pipe` to create a pipe.
- Use `fork` to create a child.
- Use `read` to read from a pipe, and `write` to write to a pipe.
- Use `getpid` to find the process ID of the calling process.
- 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 solution is correct if your program exchanges a byte between two processes and produces output as shown above.

## primes (**hard**)

Write a concurrent prime sieve program for xv6 using pipes. This idea is due to Doug McIlroy, inventor of Unix pipes. The picture halfway down [this page](#) and the surrounding text explain how to do it. 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 35 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 limited number of file descriptors and processes, the first process can stop at 35.

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 35.
- Once the first process reaches 35, 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.

Your solution is correct if it implements a pipe-based sieve and produces 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
$
```

## find ([moderate](#))

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 is correct if produces 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
$
```

## xargs ([moderate](#))

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 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 is correct if it produces 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.

## Submit the lab

**This completes the lab.** Make sure you pass all of the make grade tests. If this lab had questions, don't forget to write up your answers to the questions in `answers-lab-name.txt`. Commit your changes (including adding `answers-lab-name.txt`) and type `make handin` in the lab directory to hand in your lab.

### Time spent

Create a new file, `time.txt`, and put in it a single integer, the number of hours you spent on the lab. Don't forget to `git add` and `git commit` the file.

### Submit

You will turn in your assignments using the google classroom.

After committing your final changes to the lab, type `make tarball` to create archive which you'll submit to classroom.

```
$ git commit -am "ready to submit my lab"
[util c2e3c8b] ready to submit my lab
2 files changed, 18 insertions(+), 2 deletions(-)

$ make tarball
$
```

If you run `make tarball` and you have either uncommitted changes or untracked files, you will see output similar to the following:

```
M hello.c
M bar.c
```

```
You have uncommitted changes. Please commit or stash them.
make: *** [Makefile:348: handin-check] Error 1
```

Inspect the above lines and make sure all files that your lab solution needs are tracked i.e. not listed in a line that begins with `M`. You can cause `git` to track a new file that you create using `git add filename`.

If `make tarball` does not work properly, try fixing the problem with the Git commands.

- Please run `'make grade'` to ensure that your code passes all of the tests
- Commit any modified source code before running `'make tarball'`

## Optional challenge exercises

- Write an uptime program that prints the uptime in terms of ticks using the `uptime` system call. ([easy](#))
- Support regular expressions in name matching for `find`. `grep.c` has some primitive support for regular expressions. ([easy](#))
- The `xv6` shell (`user/sh.c`) is just another user program and you can improve it. It is a minimal shell and lacks many features found in real shell. For example, modify the shell to not print a `$` when processing shell commands from a file ([moderate](#)), modify the shell to support `wait` ([easy](#)), modify the shell to support lists of commands, separated by `;"` ([moderate](#)), modify the shell to support sub-shells by implementing `"(` and `)"` ([moderate](#)), modify the shell to support tab completion

([easy](#)), modify the shell to keep a history of passed shell commands ([moderate](#)), or anything else you would like your shell to do. (If you are very ambitious, you may have to modify the kernel to support the kernel features you need; xv6 doesn't support much.)