

# Homework 1

## Academic Honesty

Aside from the narrow exception for collaboration on homework, all work submitted in this course must be your own. Cheating and plagiarism will not be tolerated. If you have any questions about a specific case, please ask me. We will be checking for this!

NYU Poly's Policy on Academic Misconduct: <http://engineering.nyu.edu/academics/code-of-conduct/academic-misconduct>

## Homework Notes :

### General Notes:

- Read the assignment carefully, including what files to include.
- Don't assume limitations unless they are explicitly stated.
- Treat provided examples as just that, not exhaustive list of cases that should work.
- When in doubt regarding what needs to be done, ask. Another option is test it in the real UNIX operating system. Does it behave the same way?
- **TEST** your solutions, make sure they work. It's obvious when you didn't test the code.

### Rubric :

Since we had some issues before on homework 1. Here are **some** of the things we know we will test, but these are not the **only** things we will test. Therefore make sure to test your program thoroughly and thoughtfully.

Total: 100 points

-10: Can't specify number of lines when input is from a pipe

-10: No exit() at the end of hello.c

-10: Does not handle long lines (more than 512 characters)

-20: tail does not allow specifying number of lines -50: tail does not work -10: "cat README | tail" does not work -40: "tail README" does not work -10: Debug printf left in code

In this assignment, you'll start getting familiar with xv6 by writing a couple simple programs that run in the xv6 OS.

As a prerequisite, make sure that you have followed the **install instructions from NYU classes** to get your build environment set up.

A common theme of the homework assignments is that we'll start off with xv6, and then add something or modify it in some way. This assignment is no exception. Start by getting a copy of xv6 using `git` (commands typed at the terminal, and their output, will be shown using a monospace font; the commands type will be indicated by a `$`):

```
$ git clone https://github.com/moyix/xv6-public.git
Cloning into 'xv6-public'...
remote: Counting objects: 4475, done.
remote: Compressing objects: 100% (2679/2679), done.
remote: Total 4475 (delta 1792), reused 4475 (delta 1792), pack-reused 0
Receiving objects: 100% (4475/4475), 11.66 MiB | 954.00 KiB/s, done.
Resolving deltas: 100% (1792/1792), done.
Checking connectivity... done.
```

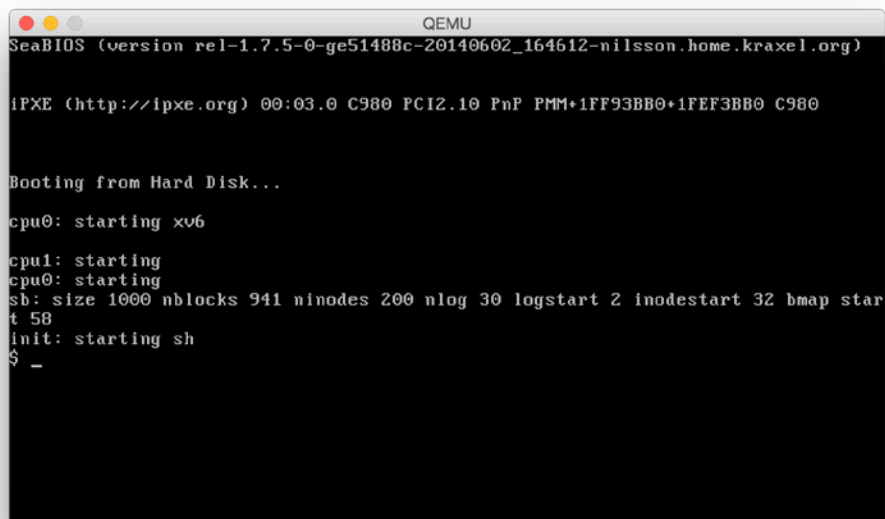
Make sure you can build and run xv6. To build the OS, use `cd` to change to the xv6 directory, and then run `make` to compile xv6:

```
$ cd xv6-public
$ make
```

Then, to run it inside of QEMU, you can do:

```
$ make qemu
```

QEMU should appear and show the xv6 command prompt, where you can run programs inside xv6. It will look something like:



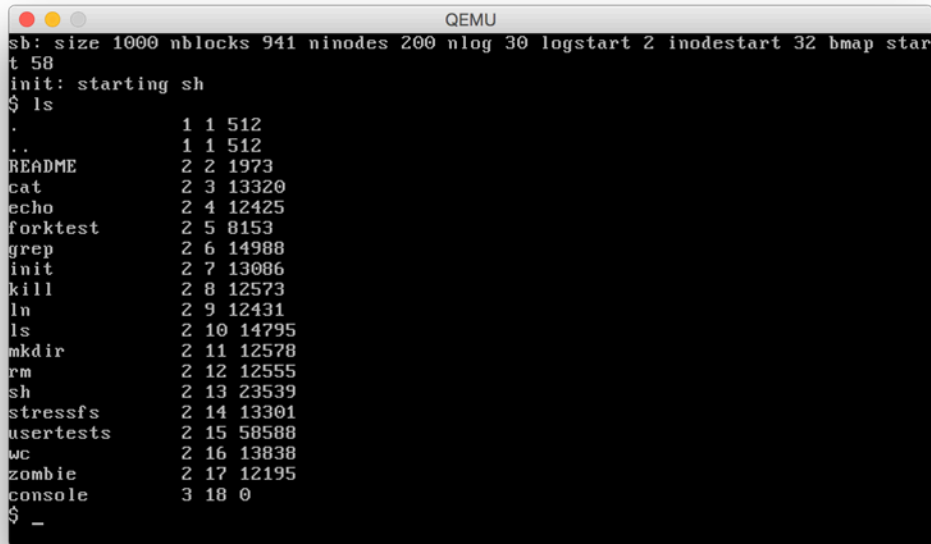
```
QEMU
SeaBIOS (version rel-1.7.5-0-ge51488c-20140602_164612-nilsson.home.kraxel.org)

IPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF93BB0+1FEF3BB0 C980

Booting from Hard Disk...

cpu0: starting xv6
cpu1: starting
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ _
```

You can play around with running commands such as `ls`, `cat`, etc. by typing them into the QEMU window; for example, this is what it looks like when you run `ls` in `xv6`:



```
QEMU
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ ls
.          1 1 512
..         1 1 512
README    2 2 1973
cat        2 3 13320
echo       2 4 12425
forktest   2 5 8153
grep       2 6 14988
init       2 7 13086
kill       2 8 12573
ln         2 9 12431
ls         2 10 14795
mkdir      2 11 12578
rm         2 12 12555
sh         2 13 23539
stressfs   2 14 13301
usertests  2 15 58588
wc         2 16 13838
zombie     2 17 12195
console    3 18 0
$ _
```

## Part 1: Hello World (20 points)

Write a program for `xv6` that, when run, prints "Hello world" to the `xv6` console. This can be broken up into a few steps:

1. Create a file in the `xv6` directory named `hello.c`
2. Put code you need to implement printing "Hello world" into `hello.c`
3. Edit the file `Makefile`, find the section `UPROGS` (which contains a list of programs to be built), and add a line to tell it to build your Hello World program. When you're done that portion of the `Makefile` should look like:

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
```

```
_zombie\  
_hello\  

```

4. Run `make` to build `xv6`, including your new program (repeating steps 2 and 4 until you have compiling code)
5. Run `make qemu` to launch `xv6`, and then type `hello` in the QEMU window. You should see "Hello world" be printed out.

Of course step 2 is where the bulk of the work lies. You will find that many things are subtly different from the programming environments you've used before; for example, the `printf` function takes an extra argument that specifies where it should print to. This is because you're writing programs for a new operating system, and it doesn't have to follow the conventions of anything you've used before. To get a feel for how programs look in `xv6`, and how various APIs should be called, you can look at the source code for other utilities: `echo.c`, `cat.c`, `wc.c`, `ls.c`.

### Hints:

1. In places where something asks for a file descriptor, you can use either an actual file descriptor (i.e., the return value of the `open` function), or one of the *standard I/O descriptors*: 0 is "standard input", 1 is "standard output", and 2 is "standard error". Writing to either 1 or 2 will result in something being printed to the screen.
2. The standard header files used by `xv6` programs are "`types.h`" (to define some standard data types) and "`user.h`" (to declare some common functions). You can look at these files to see what code they contain and what functions they define.

### A brief digression on IDEs and text editors

I do not have strong preferences as to *how* you create source code. I personally prefer to use a traditional text editor that can be run at the command line such as `pico`. Although `vim` and `emacs` are great as well and there are plenty of alternatives out there. On OS X, some may prefer to use XCode, others may prefer to use something like [TextMate](#) or [Sublime Text](#). In the Linux VM I have provided, `pico` works fine. As long as you get a plain text file out of it with valid C syntax, you can choose whatever you like.

How you *compile* the code is another matter. The `xv6` OS is set up to be built using `make`, which uses the rules defined in `Makefile` to compile the various pieces of `xv6`, and to allow you to run the code. The simplest way to build and run it is to use this system. Trying to coerce an IDE such as XCode into building `xv6` is far more trouble than it's worth.

## Part 2: Implementing the `'tail'` command (50 points)

Write a program that prints the last 10 lines of its input. If a filename is provided on the command line (i.e., `tail FILE`) then `tail` should open it, read and print the last 10 lines, and then close it. If no filename is provided, `tail` should read from standard input.

```
$ tail README
```

```
To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run "make".
On non-x86 or non-ELF machines (like OS X, even on x86), you will
need to install a cross-compiler gcc suite capable of producing x86 ELF
binaries. See http://pdos.csail.mit.edu/6.828/2014/tools.html.
Then run "make TOOLPREFIX=i386-jos-elf-".
To run xv6, install the QEMU PC simulators. To run in QEMU, run "make qe
mu".
To create a typeset version of the code, run "make xv6.pdf". This
requires the "mpage" utility. See http://www.mesa.nl/pub/mpage/.
```

You should also be able to invoke it without a file, and have it read from standard input. For example, you can use a *pipe* to direct the output of another xv6 command into tail:

```
$ grep the README | tail

Version 6 (v6). xv6 loosely follows the structure and style of v6,
xv6 borrows code from the following sources:
JOS (asm.h, elf.h, mmu.h, bootasm.S, ide.c, console.c, and others)
Plan 9 (entryother.S, mp.h, mp.c, lapic.c)
In addition, we are grateful for the bug reports and patches contributed
by
The code in the files that constitute xv6 is
To run xv6, install the QEMU PC simulators. To run in QEMU, run "make qe
mu".
To create a typeset version of the code, run "make xv6.pdf". This
requires the "mpage" utility. See http://www.mesa.nl/pub/mpage/.
```

The above command searches for all instances of the word `the` in the file `README`, and then prints the last 10 matching lines.

### Hints:

1. Many aspects of this are similar to the `wc` program: both can read from standard input if no arguments are passed or read from a file if one is given on the command line. Reading its code will help you if you get stuck.

## Part 3: Extending `tail` (30 points)

The traditional UNIX `tail` utility can print out a configurable number of lines from the end of a file. Implement this behavior in your version of `tail`. The number of lines to be printed should be specified via a command line argument as `tail -NUM FILE`, for example `tail -2 README` to print the last 2 lines of the file `README`. The expected output of that command is:

```
$ tail -2 README

To create a typeset version of the code, run "make xv6.pdf". This
requires the "mpage" utility. See http://www.mesa.nl/pub/mpage/.
```

If the number of lines is not given (i.e., if the first argument does not start with -), the number of lines to be printed should default to 10 as in the previous part.

**Hints:**

1. You can convert a string to an integer with the `atoi` function.
2. You may want to use *pointer arithmetic* (discussed in class in Lecture 2) to get a string suitable for passing to `atoi`.

## Submitting the Assignment

Submit `hello.c` and the completed `tail.c` on NYU Classes.