Project 1

Robert Utterback

January 19, 2019

1 Overview

There are two parts to this project. The goal of the first part of this project is to get up to speed on C programming. You will get experience making system calls from your program, compiling with gcc in the linux environment, and reading man pages to understand how to use library functions or system calls.

In the second part of the project, you will get an introduction to `xv6`. `Xv6` is a fully functional operating system developed at MIT. `Xv6` is much simpler than modern operating systems, making it a great tool for teaching OS concepts. We will use `xv6` to better understand how the concepts we cover in class are implemented in a real operating system. The first lab will be reasonably light on coding, but will give you an opportunity to explore and understand the `xv6` code base as well as understand how system calls are handled by `xv6`.

You will work alone on both parts of this project; however you may feel free to discuss the project and give/receive help debugging your or someone else's project. **Sharing code is cheating.** A good rule of thumb is: if you look at someone's code, do something else for an hour before returning to your own code.

1.1 Programming environment

You should code in C in a linux environment. The easiest way to do this is to use the department server as in many of the department's classes. Other options may be:

- Installing Linux on one of your personal machines, either as the main OS or dual-booting
- Installing Linux on a virtual machine
- installing the correct toolchains and tools on a system of your choice[1]

[1]This can be tricky and time consuming: if you choose to do this it will be done on your own time. Instructions on this can be found here: https://pdos.csail.mit.edu/6.828/2016/tools.html.

2 Part A: Systems Programming Intro

2.1 Overview

You will write a program in C to read records in from a file, sort and filter the records based on their key values, and output the sorted and filtered records to a new file.

The program is run from a terminal using:

```
./sort -i inputfile -o outputfile
```

This line means the `sort` program is being run with 2 required command-line arguments:

1. an input file called `inputfile`
2. an output file called `outputfile`

Input files can be generated using `generate.c` which will be given to you. After compiling and running generate, you will have a file of unsorted records (40 bytes each) in the form:

kddddddddd

where each letter represents a 32-bit unsigned int. The k (the first 32 bit unsigned int) is the key of the record and should be used for sorting and filtering. d's are data corresponding to the key and are to be kept alongside the key. This means the entire record must be moved when sorted.

Some helper files are included in `/home/comp345/proj1a.zip`. Copy this file to whatever directory you'd like to work in and unzip the files with `unzip proj1a.zip`.

You can use `dump.c` to output the contents of a generated file in a readable format. This will be useful for debugging and testing your program.

Your goal is to implement the `sort` program. It should read input records from a generated input file, sort them based on the key, and write the correct records out to the specified output file in sorted order.

Name your source file for sorting `sort.c`. A script will be used to compile and test your code, please ensure you follow this instruction!! If you have multiple source files or additional header files, be sure to submit those as well.

2.2 Helper Program Details

Compile `generate.c` using:

```
gcc -o generate generate.c -Wall -Werror
```

Note: You will also need the `record.h` header file to compile

Run generate using:

```
./generate -s 0 -n 100 -o ./data
```

`generate` takes 3 flags:

- `-s` specifies the random number seed; this allows you to generate different files to use for testing
- `-n` specifies the number of records to generate
- `-o` specifies the file name to write output to, which will be the input to your `sort` program

To compile `dump.c` use:

```
gcc -o dump dump.c -Wall -Werror
```

Run using:

```
./dump -i inputfile
```

This will read in 40 bytes at a time from the input file specified by the `-i` flag, and print the bytes formatted as a record to stdout.

2.3 Hints

- Use `open()`, `read()`, `write()`, and `close()` to access files. See code in `generate.c` and `dump.c` for examples. Feel free to copy code out of `generate.c` and `dump.c` as needed. Any files given to you are fair game to model your code after.
- To get the size of an input file before reading from it, use `stat()` or `fstat()` calls.
- To sort the records, feel free to use the `qsort()` library routine.
- To exit, call `exit()` with one parameter. The parameter is visible to the user after exiting, allowing the user to see if the program ran successfully or if the program exited with an error. Generally, `exit(0)` is used for a successful exit while a non-zero parameter indicates the program exited with an error. Use `fprintf()` to output a useful error message to stderr anytime your program exits on an error.
- Use the `malloc()` routine to dynamically allocate memory. Make sure to check that the call to `malloc()` succeeded, exit gracefully if not.
- Read the man pages for any routine or system call that you don't understand. For example, `man malloc` will give you information on how to use `malloc()`.

2.4 Bonus

For some bonus points, extend your sort program with an optional option `-t`. This option should take an integer argument. This argument serves as a threshold for the records — you should filter out any records with keys strictly greater than the threshold value.

3 Part B: Xv6 Intro

3.1 Overview

In this part we'll download the source code, build, and install `xv6`. We'll run `xv6` on `qemu`, a system emulator, as opposed to running it on real hardware, since running it on real hardware would mean manually restarting the machine every time something fails. I've already installed `qemu`[2] on the server for you to use.

You will then implement a system call called `getreadcount`. This system call simply returns the number of times the `read` system call has been called since startup. This will not require much coding, however it will require you to dig into the source code a bit and begin to understand how it is working.

3.2 Getting and building xv6

- Create a directory for the xv6 source code.
- Clone the git repository for xv6 using:

```
git clone git://github.com/mit-pdos/xv6-public.git
```

- `cd` into the directory created by the clone. Build the source code using `make`
- Run the `make qemu` command.
- `Qemu` should start and emulate a system running your built `xv6` code. Pretty awesome! Play around with `xv6` a bit and notice how little functionality it has. When you're ready to exit, type "Ctrl-a x" (hit the 'a' key while holding down the control key, then release both and hit the 'x' key).

3.3 Project details

Now that you have a running emulator and the source code for `xv6`, your task is to implement a system call called `getreadcount`. This system call should simply return the number of times the `read` system call has been called since startup. You will need a counter to track this.

Look through the `xv6` source code at some other system calls (`sys_getpid`, `sys_kill`, etc.) to get an idea of which files you will need to modify and how you should go about creating a new system call. I suggest reading about linux utilities such as grep to make searching through the source code easier.

To test your system call, you may decide to implement a user space program in the `xv6` code[3]. `ls` is an example of a user space program built into `xv6`, I would suggest modeling your program off of it and looking how it is included in the `Makefile` to make sure your program is compiled along with `xv6`.

Most of your time on this part of the project should be spent looking through the source code and making enough sense of it to be able to modify it to do what you need to do. There are likely other pieces of code that do something similar to what you want to do. You are free to copy and modify any code already in the `xv6` source.

[2] Actually I installed a patched version that includes better debugging functionality

[3] I highly recommend this; otherwise it is too easy to miss something

3.4 Bonus

For some bonus points you can implement a slightly harder system call: `sccount`. This system call should simply return the number of times a particular system call (given as a parameter) has been called since startup:

- `sccount` takes 1 parameter — a system call number.
- `sccount` returns an integer — the number of times the above system call has been called. You will need some counters to track this.

4 Submitting

First, in your `xv6` source directory, run `git diff > changes.patch`. This collects all the changes you made to **existing** `xv6` files, but not any new files you might have created. Then you can run `handin` to submit as `proj1` with `changes.patch` and any new files you added. These new files would include `sort.c` and any additional files for part A, as well as **new** files you added for `xv6`. Do **not** submit compiled code, `generate.c`, `dump.c`, or any `xv6` files that you didn't change. Further, you should include a "README" file that describes each submitted file and an overall summary of your changes, both for part A and part B.