Lecture 01: Welcome to CS110: Principles of Computer Systems

Principles of Computer Systems

Spring 2019

Stanford University

Computer Science Department

Lecturer: Chris Gregg

PDF of this presentation

Lecture 01: Welcome to CS110: Principles of Computer Systems

- **What is this class all about?**
  - Five main topics (more detail in a few slides):
    1. Unix Filesystems
    2. Multiprocessing (multiple *processes* running simultaneously)
    3. Signal Handling (sending a *signal* to a process)
    4. Multithreading (multiple *threads* in a single process running simultaneously)
    5. Networking Servers and Clients
  - There will be *eight* assignments, with each assignment at least one week in duration
    1. C and C++ refresher
    2. Unix Filesystems
    3. Multiprocessing Warmup
    4. Multiprocessing: Stanford Shell
    5. Multithreading I
    6. Multithreading II: ThreadPool
    7. Networking
    8. Networking, Threading, Multiprocessing: *MapReduce*

Lecture 01: Welcome to CS110: Principles of Computer Systems

- **I'm Chris Gregg** (cgregg@stanford.edu)
  - Electrical Engineering undergrad Johns Hopkins, Master's of Education, Harvard, Ph.D. in Computer Engineering, University of Virginia
  - Lecturer in CS, teaching CS 106B/X, CS 107/107E, CS208E, CS 110
  - At Stanford since 2016, at Tufts prior, and high school teaching prior to that.
  - First time teaching CS 110 was last quarter
    - I love this material! It is challenging, yet interesting, and it is a new window into systems that you haven't yet seen in the CS curriculum.
    - This class was carefully crafted by Jerry Cain, and I will keep it mostly the same this quarter.
  - I love to tinker
    - Stop by my office (Gates 201) some time to see my musical typewriter project.
    - I'm always happy to chat about Arduino / Raspberry Pi / iOS apps you are working on

Lecture 01: Welcome to CS110: Principles of Computer Systems

- Staff and Students
  - 209 students as of noon on April 1, 2019
  - Each of you should know C and C++ reasonably well so that you can...
    - write moderately complex programs
    - read and understand portions of large code bases
    - trace memory diagrams
  - Each of you should be fluent with Unix, **`gcc`**, **`valgrind`**, and **`make`** to the extent they're covered in CS107 or its equivalent.
  - 10 graduate student CAs
    - Feross, Ikechi, Caroline, Ryan, Sarah, Garrick, Wantong, Peter, Jake, Armin
    - The CAs will hold office hours, lead lab sections, and grade your work

Lecture 01: Welcome to CS110: Principles of Computer Systems

- Course Web Site: https://cs110.stanford.edu
  - Check the website for information about upcoming lectures, assignment handouts, discussion sections, and links to lecture slides like the one you're working through right now
- Online Student Support
  - Peer-collaborative forum I: Piazza (for the questions that require staff response)
  - Peer-collaborative forum II: Slack (for the questions that needn't involve course staff)
- Office Hours
  - Chris's office hours are Tuesdays from 9:00 until 11:00am and then again on Thursdays from 10:00am until 12:00pm, or by appointment
  - CA's will provide a full matrix of office hours, soon to be determined
  - Office hours are not for debugging your assignments, and the CA's have been instructed to not look at code. Ever.

Lecture 01: Welcome to CS110: Principles of Computer Systems

- Two Textbooks
  - First textbook is other half of CS107 textbook
    - "Computer Systems: A Programmer's Perspective", by Bryant and O'Hallaron
    - Stanford Bookstore stocks custom version of just the four chapters needed for CS110
  - Second textbook is more about systems-in-the-large, less about implementation details
    - "Principles of Computer System Design: An Introduction", by Jerome H. Saltzer and M. Frans Kaashoek
    - Provided free-of-charge online, chapter by chapter. Not stocked at Stanford Bookstore by design. You can buy a copy of it from Amazon if you want.

Lecture 01: Welcome to CS110: Principles of Computer Systems

- Lecture Examples
  - Lectures will be driven by slides and coding examples, and all coding examples can be copied/cloned into local space so you can play and confirm they work properly
  - Code examples will be developed and tested on the **`myth`** machines, which is where you'll complete all of your CS110 assignments
  - The accumulation of all lecture examples will be housed in a git repository at **`/usr/class/cs110/lecture-examples`**, which you can initially **`git clone`**, and then subsequently **`git pull`** to get the newer examples as we check them in

Lecture 01: Welcome to CS110: Principles of Computer Systems

- Lecture Slides
  - I'll try to make the slides as comprehensive as possible, but working with the code yourself is going to teach you more.
  - They are not a substitute for attending lecture
    - We go off script quite a bit and discuss high-level concepts, and you're responsible for anything that comes up in lecture
    - Exams include short answer questions in addition to coding questions, so all aspects of the course are tested

Course Syllabus

- Overview of Linux Filesystems
  - Linux and C libraries for file manipulation: **`stat`**, **`struct stat`**, **`open`**, **`close`**, **`read`**, **`write`**, **`readdir`**, **`struct dirent`**, file descriptors, regular files, directories, soft and hard links, programmatic manipulation of them, implementation of **`ls`**, **`cp`**, **`find`**, and other core Unix utilities you probably never realized were plain old C programs
  - Naming, abstraction and layering concepts in systems as a means for managing complexity, blocks, inodes, inode pointer structure, inode as abstraction over blocks, direct blocks, indirect blocks, doubly indirect blocks, design and implementation of a file system
- Multiprocessing and Exceptional Control Flow
  - Introduction to multiprocessing, **`fork`**, **`waitpid`**, **`execvp`**, process ids, interprocess communication, context switches, user versus kernel mode, system calls and how their calling convention differs from those of normal functions
  - Protected address spaces, virtual memory, virtual to physical address mapping, scheduling
  - Concurrency versus parallelism, multiple cores versus multiple processors, concurrency issues with multiprocessing, signal masks

Course Syllabus

- Threading and Concurrency
  - Sequential programming, desire to emulate the real world within a single process using parallel threads, free-of-charge exploitation of multiple cores (two per **`myth`** machine, 12-16 per **`wheat`** machine, 16 per **`oat`** machine), pros and cons of threading versus forking
  - C++ threads, **`thread`** construction using function pointers, blocks, functors, **`join`**, **`detach`**, race conditions, **`mutex`**, IA32 implementation of **`lock`** and **`unlock`**, spinlock, busy waiting, preemptive versus cooperative multithreading, **`yield`**, **`sleep_for`**
  - Condition variables, **`condition_variable_any`**, rendezvous and thread communication, **`wait`**, **`notify_one`**, **`notify_all`**, deadlock, thread starvation
  - Semaphore concept and **`semaphore`** implementation, generalized counters, pros and cons of **`semaphore`** versus exposed **`condition_variable_any`**, thread pools, cost of threads versus processes
  - Active threads, blocked threads, ready threads, high-level implementation details of a thread manager, **`mutex`**, and **`condition_variable_any`**
  - Pure C alternatives via **`pthreads`**, pros and cons of **`pthreads`** versus C++'s **`thread`** package

Course Syllabus

- Networking and Distributed Systems
  - Client-server model, peer-to-peer model, telnet, protocols, request, response, stateless versus keep-alive connections, latency and throughput issues, **`gethostbyname`**, **`gethostbyaddr`**, IPv4 versus IPv6, **`struct sockaddr`** hierarchy of records, network-byte order
  - Ports, sockets, socket descriptors, **`socket`**, **`connect`**, **`bind`**, **`accept`**, **`read`**, **`read`**, simple echo server, time server, concurrency issues, spawning threads to isolate and manage single conversations
  - C++ layer over raw C I/O file descriptors, introduction to **`sockbuf`** and **`sockstream`** C++ classes (via socket++ open source project)
  - HTTP 1.0 and 1.1, header fields, **`GET`**, **`HEAD`**, **`POST`**, response codes, caching
  - MapReduce programming model, implementation strategies using multiple threads and multiprocessing
  - Nonblocking I/O, where normally slow system calls like accept, **`read`**, and **`write`** return immediately instead of blocking
    - **`select`**, **`epoll`**, and **`libev`** libraries all provide nonblocking I/O alternatives to maximize CPU time using a single thread of execution within a single process

Course Expectations

- Programming Assignments
  - 40% of final grade, with eight assignments
  - Some assignments are single file, others are significant code bases to which you'll contribute. You should *always* become familiar with the header files and the assignment handout before you start writing a single line of code.
  - Late policy is different than it is for many other CS classes
    - Every late day *potentially* costs you (read below why it's *potentially*)
      - If you submit on time, you can get 100% of the points.
      - If you can't meet the deadline, you can still submit up to 24 hours later, but your overall score is capped at 90%
      - If you need more than 24 additional hours to submit, you can submit up to 48 hours later, but overall score is capped at 60%
      - No assignments are ever accepted more than 48 hours after the deadline
    - Exception: first assignment must be submitted on time, no late days allowed
    - Requests for extensions are routinely denied, save for extenuating circumstances (e.g. family emergency, illness requiring medical intervention, and so forth)

Course Expectations

- Discussion Sections
  - In addition to our MW lectures, you'll also sign up for an 80-minute section to meet each week
  - We introduced the CS110 discussion section for the first time almost two years ago, and the general consensus is that they've substantially improved the course
  - If you have a laptop, bring it to discussion section. Section will be a mix of theoretical work, coding exercises, and advanced software engineering etudes using **`gdb`** and **`valgrind`**
  - Discussion section signups will go live later this week
  - 5% of final grade, provided you attend all of them
    - Everyone's discussion section grade is 100%
    - Every time you miss a discussion section, your discussion section grade counts a little less, and your final exam score counts a little more
    - Exact policy details are spelled out in the Course Information handout
- Discussion Section signup will start at noon on **Sunday, April 7th at Noon.** Go to https://web.stanford.edu/class/cs110/templates/labs to sign up.

Course Expectations

- Midterm
  - The midterm is Thursday, May 2nd from 6-8pm.
  - We will use the laptop-based program ***BlueBook*** to take the exams. You may have used *BlueBook* in CS 106A/B or CS 107, but if not, it is an in-house program that allows you to type your answers for questions. It has a similar feel to taking a paper exam (e.g., no code compiling, etc.). If you do not have a laptop to use for the exam, please email Chris and we can set you up with one to use.
  - 20% of final grade, material drawn from first five or so weeks of lecture, mix of implementation and short answer questions
  - Closed-book, closed-notes, closed-electronics, one double-sided cheat sheet that you can prepare ahead of time
  - You must pass the midterm in order to pass the class
    - Passing score will be revealed on midterm solution set, which will be posted well before the withdrawal deadline
  - Multiple practice midterms will be provided
  - If you have a competing class and would prefer to take the midterm another time, we'll allow you to take it earlier that same day, provided you email Chris ahead of time and explain why you need to take the exam earlier
  - If you have testing accommodations, please email Chris as soon as possible.

Course Expectations

- Final Exam
  - Three-hour final is Monday, June 10th at 3:30pm, also using *BlueBook*.
    - 35% of final grade, cumulative, mix of implementation and short answer questions
      - Counts even more with each discussion section absence
    - Closed-book, closed-notes, closed-electronics, two double-sided cheat sheets that you can prepare ahead of time
    - You must pass the final in order to pass the class
    - Multiple practice finals will be provided
  - The final exam will also be offered on Monday, June 10th at 7:00pm
    - Only students with another 3:30pm final on the 10th are permitted to take the final exam at the later time
    - Email Chris directly if you need to take the final exam during the alternate time slot because of a competing final
  - The final exam isn't being offered any other times.

Honor Code

- Please take the honor code seriously, because the CS Department does
  - Everything you submit for a grade is expected to be original work
  - Provide detailed citations of all sources and collaborations
  - The following are clear no-no's
    - Looking at another student's code
    - Showing another student your code
    - Discussing assignments in such detail that you duplicate a portion of someone else's code in your own program
    - Uploading your code to a public repository (e.g. github) so others can find it
      - If you'd like to upload your code to a **private** repository, you can do so on github or some other hosting service that provides free-of-charge private hosting

Introduction to UNIX Filesystems

- You should already be familiar with the Linux filesystem as a user. The filesystem uses a tree-based model to store files and directories of files. You can get details of a file in a particular directory with the **`ls`** command

```
cgregg@myth58:~/cs110/spring-2019/lecture-examples/filesystems$ ls
alphabet.txt  contains.c  copy.c  list.c  Makefile  search.c  t.c  vowels.txt
```

- You can get a more detailed listing with the **`ls -al`** command:

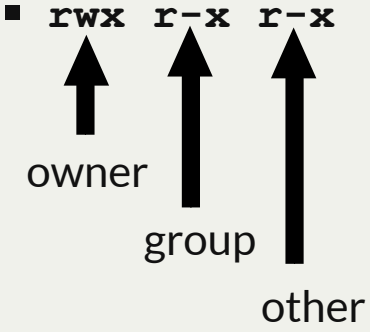
```
ls -al
total 23
drwx------  2 cgregg operator 2048 Mar 29 12:33 .
drwx------ 10 cgregg operator 2048 Mar 29 12:33 ..
-rw-------  1 cgregg operator   27 Mar 29 12:33 alphabet.txt
-rw-------  1 cgregg operator 2633 Mar 29 12:33 contains.c
-rw-------  1 cgregg operator 1882 Mar 29 12:33 copy.c
-rw-------  1 cgregg operator 5795 Mar 29 12:33 list.c
-rw-------  1 cgregg operator  628 Mar 29 12:33 Makefile
-rw-------  1 cgregg operator 2302 Mar 29 12:33 search.c
-rw-------  1 cgregg operator 1321 Mar 29 12:33 t.c
-rw-------  1 cgregg operator    6 Mar 29 12:33 vowels.txt
```

- With this listing, there are two files listed as directories (**`d`**), "**`.`**" and "**`..`**". These stand for:
  - "**`.`**" is the *current* directory
  - "**`..`**" is the *parent* directory
- The "**`rwx------`**" designates the *permissions* for a file or directory, with "**`r`**" for *read* permission, "**`w`**" for *write* permission, and "**`x`**" for *execute* permission (for runnable files).

Introduction to UNIX Filesystems

```
$ ls -l list
-rwxr-xr-x 1 cgregg operator 19824 Mar 29 12:47 list
```

- There are actually three parts to the permissions line, each with the three permission types available:
  - **rwx r-x r-x**



In this case, the owner has read, write, and execute permissions, the group has only read and execute permissions, and the user also has only read and execute permissions.

- Because each individual set of permissions can be either **r**, **w**, or **x**, there are three bits of information per permission field. We can therefore, use *base 8* to designate a particular permission set. Let's see how this would work for the above example:
- permissions: **rwx r-x r-x**
- bits (base 2): **111 101 101**
- base 8: **7 5 5**
- So, the permissions for the file would be, **755**

Introduction to UNIX Filesystems

- In C, a file can be created using the **`open`** *system call*, and you can set the permissions at that time, as well. We will discuss the idea of system calls soon, but for now, simply think of them as a function that can do system-y stuff. The open command has the following signatures (and this works in C, even though C does not support function overloading! How, you ask? See here.):

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

- There are many flags (see **`man 2 open`** for a list of them), and they can be bitwise **`or`**'d together. You must include one of the following flags:
- **`O_RDONLY`** -- read only
- **`O_WRONLY`**-- write only
- **`O_RDWR`**-- read and write

We will generally only care about the following other flags when creating a file:

- **`O_CREAT`** -- If the file does not exist, it will be created.
- **`O_EXCL`** -- Ensure that this call creates the file, and fail if the file exists already

Introduction to UNIX Filesystems

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

- When creating a file, the third argument, **`mode`**, is used, to *attempt* to set the permissions.
- The reason it is "attempt" is because there is a default permissions mask, called **`umask`**, that limits the permissions. **`umask`** has a similar octal value to the permissions, although if a bit is *set* in the **`umask`**, then trying to set that bit with the **`mode`** parameter will not be allowed. The **`umask`** can be set with the following system call:

```
mode_t umask(mode_t mask); // see "man 2 umask" for details
```

- The return value is the *old* mask (the one that was already set).
- If you want to simply *check* the **`umask`** value, you must call the function twice. E.g.:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

int main() {
    mode_t old_mask = umask(0); // set to 0, but get old mask as return value
    umask(old_mask); // restore to original
    printf("umask is set to %03o\n",old_mask);
    return 0;
}
```

```
$ gcc show_umask.c -o show_umask
$ ./show_umask
umask is set to 077
```

- This output means that the only permissions that can be set are for the user (**`rwx`**). The group and other permissions can not be set because all three bits of their respective permissions are set in **`umask`**.

Introduction to UNIX Filesystems

- Today's lecture examples reside within **`/usr/class/cs110/lecture-examples/filesystems`**.
  - The **`/usr/class/cs110/lecture-examples`** directory is a **`git`** repository that will be updated with additional examples as the quarter progresses.
  - To get started, type **`git clone /usr/class/cs110/lecture-examples cs110-lecture-examples`** at the command prompt to create a local copy of the master.
  - Each time I mention there are new examples (or whenever you think to), descend into your local copy and type **`git pull`**. Doing so will update your local copy to match whatever the master has become.

Introduction to UNIX Filesystems

- You can override **`umask`** if you need to set the permissions a particular way.
- The following program creates a file and sets its permissions:

```c
#include <fcntl.h>    // for open
#include <unistd.h>   // for read, write, close
#include <stdio.h>
#include <sys/types.h> // for umask
#include <sys/stat.h>  // for umask
#include <errno.h>

const char *kFilename = "my_file";
const int kFileExistsErr = 17;
int main() {
    umask(0); // set to 0 to enable all permissions to be set
    int file_descriptor = open(kFilename, O_WRONLY | O_CREAT | O_EXCL, 0644);
    if (file_descriptor == -1) {
        printf("There was a problem creating '%s'!\n",kFilename);
        if (errno == kFileExistsErr) {
            printf("The file already exists.\n");
        } else {
            printf("Unknown errorno: %d\n",errno);
        }
        return -1;
    }
    close(file_descriptor);
    return 0;
}
```

```
$ make open_ex
cc     open_ex.c   -o open_ex
$ ./open_ex
$ ls -l my_file
-rw-r--r-- 1 cgregg operator 0 Mar 31 13:29 my_file
```

UNIX Filesystem APIs

- We have already discussed two file system API calls: **`open`** and **`umask`**. We are going to look at other low-level operations that allow programmers to interaction with the file system. We will focus here on the direct system calls, but when writing production code (i.e., for a job), you will often use indirect methods, such as **`FILE *`**, **`ifstream`**s, and **`ofstream`**s.
- Requests to open a file, read from a file, extend the heap, etc., all eventually go through system calls, which are the only functions that can be trusted to interact with the system on your behalf. The operating system *kernel* actually runs the code for a system call, completely isolating the system-level interaction from your (potentially harmful) program.

Implementing `copy` to emulate `cp`

- The implementation of **`copy`** (designed to mimic the behavior of **`cp`**) illustrates how to use **`open`**, **`read`**, **`write`**, and **`close`**. It also introduces the notion of a file descriptor.
  - **`man`** pages exist for all of these functions (e.g. **`man 2 open`**, **`man 2 read`**, etc.)
  - Full implementation of our own **`copy`**, with exhaustive error checking, is right here.
  - Simplified implementation, sans error checking, is on the next slide.

Implementing `copy` to emulate `cp`

```
int main(int argc, char *argv[]) {
  int fdin = open(argv[1], O_RDONLY);
  int fdout = open(argv[2], O_WRONLY | O_CREAT | O_EXCL, 0644);
  char buffer[1024];
  while (true) {
    ssize_t bytesRead = read(fdin, buffer, sizeof(buffer));
    if (bytesRead == 0) break;
    size_t bytesWritten = 0;
    while (bytesWritten < bytesRead) {
      bytesWritten += write(fdout, buffer + bytesWritten, bytesRead - bytesWritten);
    }
  }
  close(fdin);
  close(fdout);
  return 0;
}
```

- **`read`** and **`write`** are defined as follows. **`#include<unistd.h>`** to use them.

```
ssize_t read(int fd, void *buf, size_t count);

ssize_t write(int fd, const void *buf, size_t count);
```

- **`fd`** is a file descriptor (as seen in the return value of **`open`**), and is just an integer.
- **`buf`** is just a **`char *`** array (though technically a **`void *`** array).
- **`count`** is the number of bytes to read or write.
- The return value is a **`ssize_t`**, which is the same magnitude as a **`size_t`**, but with the ability to have negative values. Normal return values are the number of bytes read or written. A return value of **`-1`** indicates an error, and **`errno`** is set appropriately.
- The return value is *not always* the same as count, but only the number of bytes *successfully* read or written.

Implementing `copy` to emulate `cp`

```
int main(int argc, char *argv[]) {
  int fdin = open(argv[1], O_RDONLY);
  int fdout = open(argv[2], O_WRONLY | O_CREAT | O_EXCL, 0644);
  char buffer[1024];
  while (true) {
    ssize_t bytesRead = read(fdin, buffer, sizeof(buffer));
    if (bytesRead == 0) break;
    size_t bytesWritten = 0;
    while (bytesWritten < bytesRead) {
      bytesWritten += write(fdout, buffer + bytesWritten, bytesRead - bytesWritten);
    }
  }
  close(fdin);
  close(fdout)
  return 0;
}
```

- The **`read`** system call will *block* until the requested number of bytes have been read. If the return value is 0, there are no more bytes to read (e.g., the file has reached the end, or been closed).
- If **`write`** returns a value less than **`count`**, it means that the system couldn't write all the bytes at once. This is why the **`while`** loop is necessary, and the reason for keeping track of **`bytesWritten`** and **`bytesRead`**.
- You should close files when you are done using them, although they will get closed by the OS when your program ends. We will use **`valgrind`** to check if your files are being closed.

Pros and cons of file descriptors over `FILE` pointers and C++ `iostreams`

- The file descriptor abstraction provides direct, low level access to a stream of data without the fuss of data structures or objects. It certainly can't be slower, and depending on what you're doing, it may even be faster.
- `FILE` pointers and C++ `iostream`s work well when you know you're interacting with standard output, standard input, and local files.
  - They are less useful when the stream of bytes is associated with a network connection.
  - `FILE` pointers and C++ `iostream`s assume they can rewind and move the file pointer back and forth freely, but that's not the case with file descriptors associated with network connections.
- File descriptors, however, work with `read` and `write` and little else used in this course.
- C `FILE` pointers and C++ streams, on the other hand, provide automatic buffering and more elaborate formatting options.

Implementing `t` to emulate `tee`

- Overview of **`tee`**
  - The **`tee`** program that ships with Linux copies everything from standard input to standard output, making zero or more *extra* copies in the named files supplied as user program arguments. For example, if the file contains 27 bytes—the 26 letters of the English alphabet followed by a newline character—then the following would print the alphabet to standard output and to three files named **`one.txt`**, **`two.txt`**, and **`three.txt`**.

```
$ cat alphabet.txt | ./tee one.txt two.txt three.txt
abcdefghijklmnopqrstuvwxyz
$  cat one.txt
abcdefghijklmnopqrstuvwxyz
$  cat two.txt
abcdefghijklmnopqrstuvwxyz
$  diff one.txt two.txt
$  diff one.txt three.txt
$
```

- If the file **`vowels.txt`** contains the five vowels and the newline character, and **`tee`** is invoked as follows, **`one.txt`** would be rewritten to contain only the English vowels.

```
$ cat vowels.txt | ./tee one.txt
aeiou
$  cat one.txt
aeiou
```

- Full implementation of our own **`t`** executable, with error checking, is right here.
- Implementation replicates much of what **`copy.c`** does, but it illustrates how you can use low-level I/O to manage many sessions with multiple files. The implementation inlined across the next two slides omit error checking.

Implementing `t` to emulate `tee`

- Features:
  - Note that **`argc`** incidentally provides a count on the number of descriptors that write to. That's why we declare an integer array (or rather, a file descriptor array) of length **`argc`**.
  - **`STDIN_FILENO`** is a built-in constant for the number 0, which is the descriptor normally attached to standard input. **`STDOUT_FILENO`** is a constant for the number 1, which is the default descriptor bound to standard output.
  - I assume all system calls succeed. I'm not being lazy, I promise. I'm just trying to keep the examples as clear and compact as possible. The official copies of the working programs up on the **`myth`** machines include real error checking.

```
int main(int argc, char *argv[]) {
  int fds[argc];
  fds[0] = STDOUT_FILENO;
  for (size_t i = 1; i < argc; i++)
    fds[i] = open(argv[i], O_WRONLY | O_CREAT | O_TRUNC, 0644);

  char buffer[2048];
  while (true) {
    ssize_t numRead = read(STDIN_FILENO, buffer, sizeof(buffer));
    if (numRead == 0) break;
    for (size_t i = 0; i < argc; i++) writeall(fds[i], buffer, numRead);
  }

  for (size_t i = 1; i < argc; i++) close(fds[i]);
  return 0;
}

static void writeall(int fd, const char buffer[], size_t len) {
  size_t numWritten = 0;
  while (numWritten < len) {
    numWritten += write(fd, buffer + numWritten, len - numWritten);
  }
}
```