Low-Level I/O – the POSIX Layer CSE 333 Winter 2025

Instructor: Hal Perkins

Teaching Assistants:

Lainey Jeon	Hannah Jiang	Irene Lau
Nathan Li	Janani Raghavan	Sean Siddens
Deeksha Vatwani	Yiqing Wang	Wei Wu
Jennifer Xu		

Administrivia

- HW1 due tomorrow night
 - Any last-minute surprises? Questions?
- No exercise due Friday morning!
- Sections tomorrow: POSIX I/O and reading directories
- Next exercise: find text files in directory and print contents
 - Based on section stuff and is a warmup for hw2
 - Out tomorrow after sections; due Monday morning
- Friday: HW2 out; demo in class, starter code pushed to repos by beginning of weekend

Exercises, hw, and code quality

- In the initial set of exercises, we've been focusing on trying to provide helpful feedback as we get better at writing high-quality code, without necessarily affecting final scores. But there are a few things that should be fixed by now but are persisting, and will be more of a problem if they continue. A couple of particular ones:
 - Use cpplint.py to check code and fix things it catches (except for known exceptions to the rules, like matching library functions that use long ints)
 - Match the Google Style guide for conventions like variable naming, using 2 spaces (only) for indenting, etc.
 - All functions must be declared in a file before any function definitions either in a #included header for "public" things, or at the top of the file for local "private" things, and there need to be appropriate comments specifying how the function works
 - Run valgrind to check for memory bugs
- Overall, keep up the good work we're seeing lots of improvement and great code that continues to get better

Lecture Outline

* POSIX Lower-Level I/O

Remember This Picture?

- Your program can access many layers of APIs:
 - C standard library
 - Some are just ordinary functions (<string.h>, for example)
 - Some also call OS-level (POSIX) functions (<stdio.h>, for example)
 - POSIX compatibility API
 - C-language interface to OS system calls (fork(), read(), etc.)
 - Underlying OS system calls
 - Assembly language 🙂



C Standard Library File I/O

- So far you've used the C standard library to access files
 - Use a provided FILE* stream abstraction
 - fopen(),fread(),fwrite(),fclose(),fseek()
- These are convenient and portable
 - They are buffered
 - They are implemented using lower-level OS calls

Lower-Level File Access

- Most UNIX-en support a common set of lower-level file access APIs: POSIX – Portable Operating System Interface
 - open(),read(),write(),close(),lseek()
 - Similar in spirit to their $\pm\,\star$ () counterparts from C std lib
 - Lower-level and unbuffered compared to their counterparts
 - Also less convenient
 - We will have to use these to read file system directories and for network I/O, so we might as well learn them now

open()/close()

- ✤ To open a file:
 - Pass in the filename and access mode
 - Similar to **fopen**()
 - Get back a "file descriptor"
 - Similar to FILE* from **fopen**(), but is just an int
 - Defaults: 0 is stdin, 1 is stdout, 2 is stderr

```
#include <fcntl.h> // for open()
#include <unistd.h> // for close()
...
int fd = open("foo.txt", O_RDONLY);
if (fd == -1) {
    perror("open failed");
    exit(EXIT_FAILURE);
}
...
close(fd);
```

Reading from a File

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

- Returns the number of bytes read
 - Might be fewer bytes than you requested (!!!)
 - Returns **0** if you're already at the end-of-file
 - Returns **-1** on error
- read has some surprising error modes...

Read error modes

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

- On error, read returns -1 and sets the global errno variable
- You need to check **errno** to see what kind of error happened
 - EBADF: bad file descriptor
 - EFAULT: output buffer is not a valid address
 - EINTR: read was interrupted, please try again (ARGH!!!! 😤 🧐)
 - And many others...

One way to read () n bytes

```
int fd = open(filename, O RDONLY);
char* buf = ...; // buffer of appropriate size
int bytes left = n;
int result;
while (bytes left > 0) {
 result = read(fd, buf + (n - bytes left), bytes left);
 if (result == -1) {
   if (errno != EINTR) {
     // a real error happened, so return an error result
    }
   // EINTR happened, so do nothing and try again
   continue;
 } else if (result == 0) {
   // EOF reached, so stop reading
   break;
 bytes left -= result;
close(fd);
```

Other Low-Level Functions

- Read man pages to learn about:
 - write () write data
 - fsync() flush data to the underlying device
 - opendir(), readdir(), closedir() deal with directory listings
 - Make sure you read the section 3 version (e.g. man 3 opendir)
- A useful shortcut sheet (from CMU): <u>http://www.cs.cmu.edu/~guna/15-123S11/Lectures/Lecture24.pdf</u>
- More in sections this week.... (as in, tomorrow!)