CSE 333 Section 3

POSIX I/O

Checking In & Logistics

Quick check-in:

Do you have any questions, comments, or concerns?

Exercises going ok?

Lectures making sense?

REMINDERS:

Due **TOMORROW** (10/13): Homework 1 @ 10:00 pm PDT

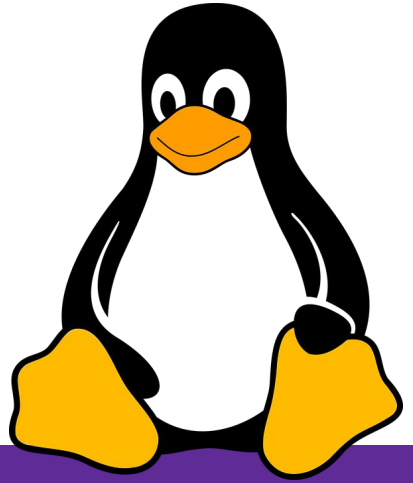
- You have until 10:00 pm on Monday without a late penalty [no early bonus]!

Due **Wednesday** (10/18): Exercise 4 @ 10:00 pm PDT

- Longest exercise of the quarter

- Please start early!!!

POSIX



POSIX (Portable Operating System Interface)

A family of IEEE standards that maintains compatibility across variants of Unix-like operating systems for basic I/O (*file*, terminal, and *network*) and for *threading*.

1. Why might a POSIX standard be beneficial (*e.g.*, from an application perspective or vs. the C stdio library)?

   - More explicit control since read and write functions are system calls and you can directly access system resources.
   - POSIX calls are unbuffered so you can implement your own buffer strategy on top of read()/write().
   - There is no standard higher level API for network and other I/O devices

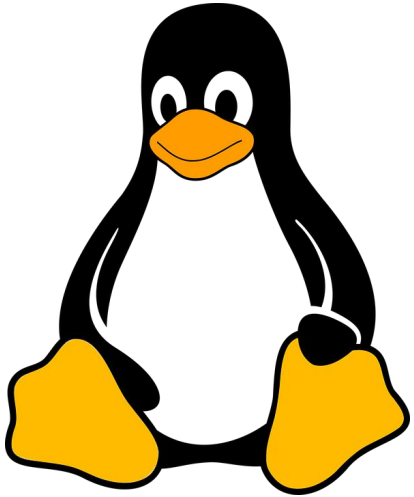
What's Tricky about (POSIX) File I/O?

- Communication with input and output devices doesn't always work as expected

  - Some details might be unknown (*e.g.*, size of a file)
  - May not process all data or fail, necessitating read/write *loops*
- Different system calls have a variety of different failure modes and error codes
  - Look up in the documentation and use pre-defined constants!
  - Lots of error-checking code needed
    - Need to handle resource cleanup on *every* termination pathway

Messy Roommate

I/O Analogy – Messy Roommate



- The Linux kernel (Tux) now lives with you in room #333
- There are `N` pieces of trash in the room
- There is a single trash can, `char bin[N]`
  - (For some reason, the trash goes in a particular order)
- You can tell your roommate to pick it up, but they are unreliable

I/O Analogy – Messy Roommate

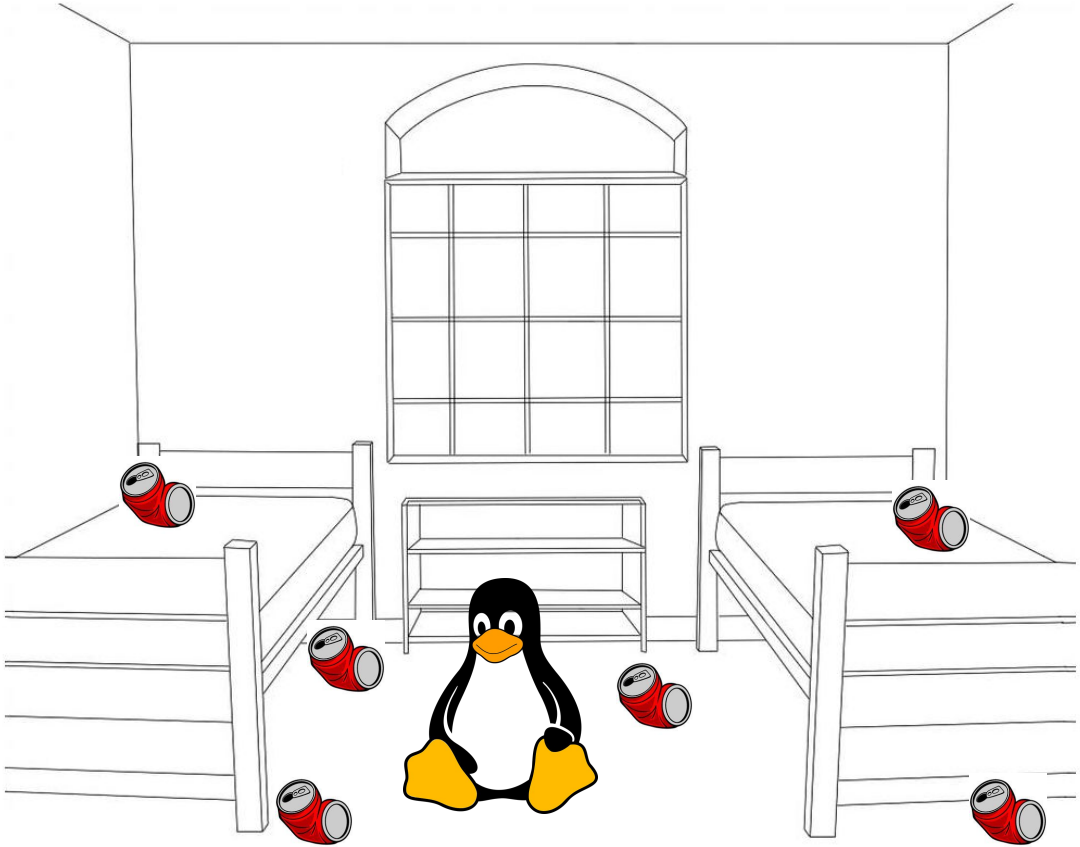
```
num_trash = Pickup(room_num, trash_bin, amount)
```

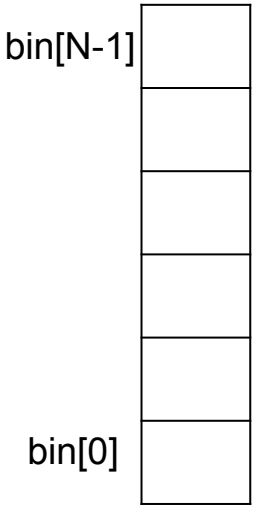
"*I tried to start cleaning, but something came up*" (got hungry, had a midterm, room was locked, etc.)	`num_trash == -1` `errno == excuse`
"*You told me to pick up trash, but the room was already clean*"	`num_trash == 0`
"*I picked up some of it, but then I got distracted by my favorite show on Netflix*"	`num_trash < amount`
"*I did it! I picked up all the trash!*"	`num_trash == amount`

```
num_trash = Pickup(room_num, trash_bin, amount)
```

`num_trash == -1, errno == excuse`
`num_trash == 0`
`num_trash < Amount`
`num_trash == Amount`

How do we get room 333 clean?



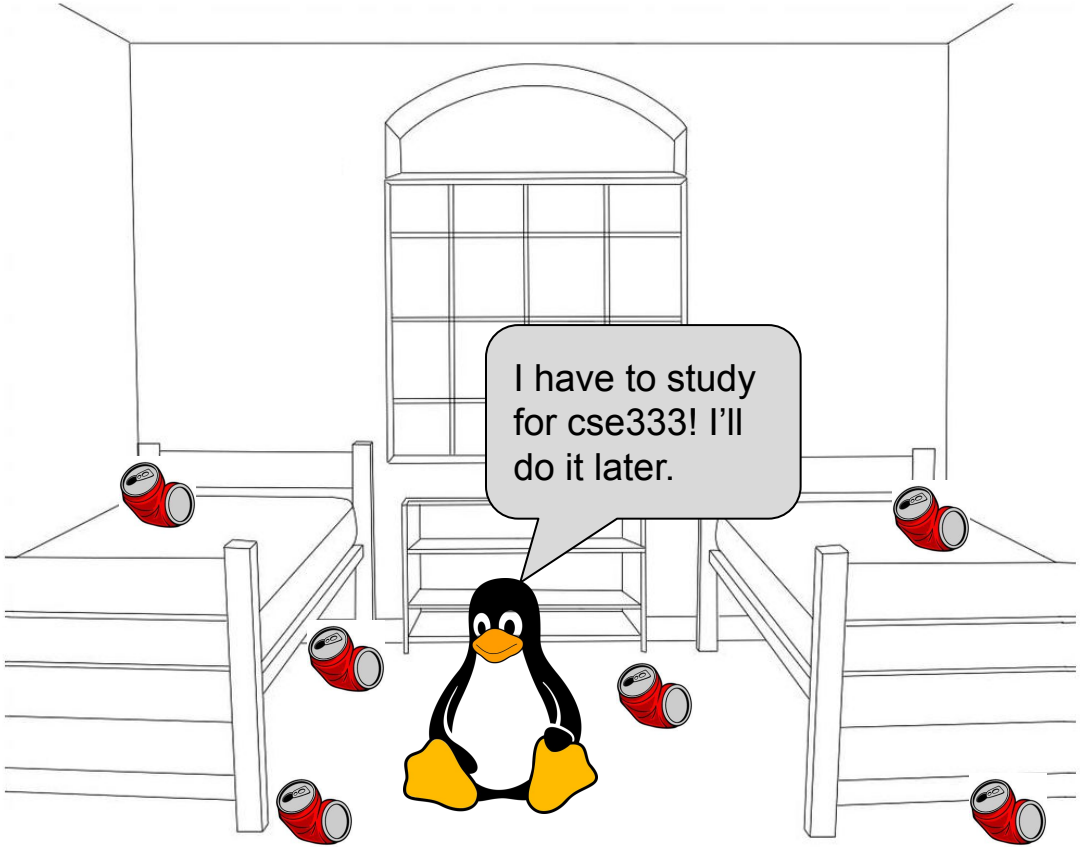


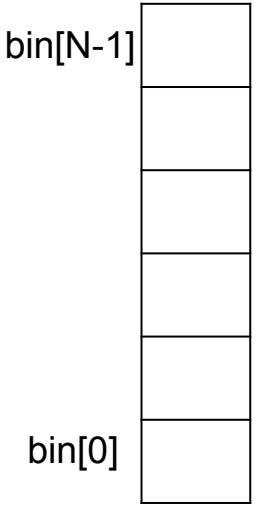
What do we do in the following scenarios?

```
num_trash pickup(roomNum, trashBin, Amount)
```

How do we get room 333 clean?

`num_trash == -1, errno == excuse`
`num_trash == 0`
`num_trash < Amount`
`num_trash == Amount`



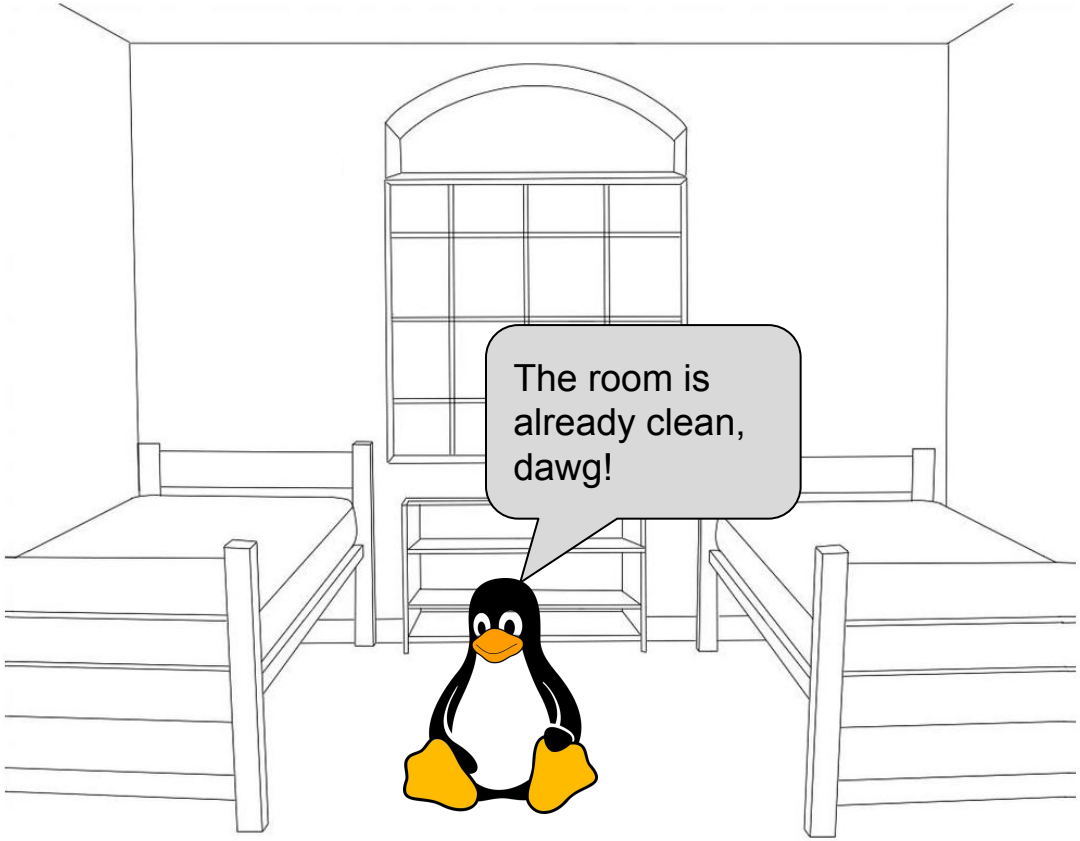


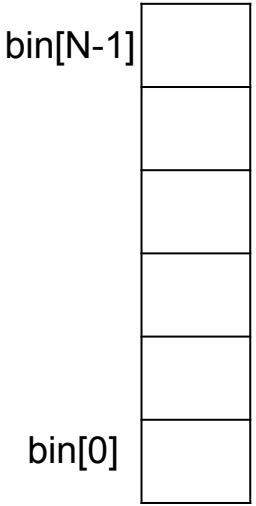
Decide if the excuse is reasonable, and either let it be or ask again.

```
num_trash pickup(roomNum, trashBin, Amount)
```

How do we get room 333 clean?

`num_trash == -1, errno == excuse`
`num_trash == 0`
`num_trash < Amount`
`num_trash == Amount`



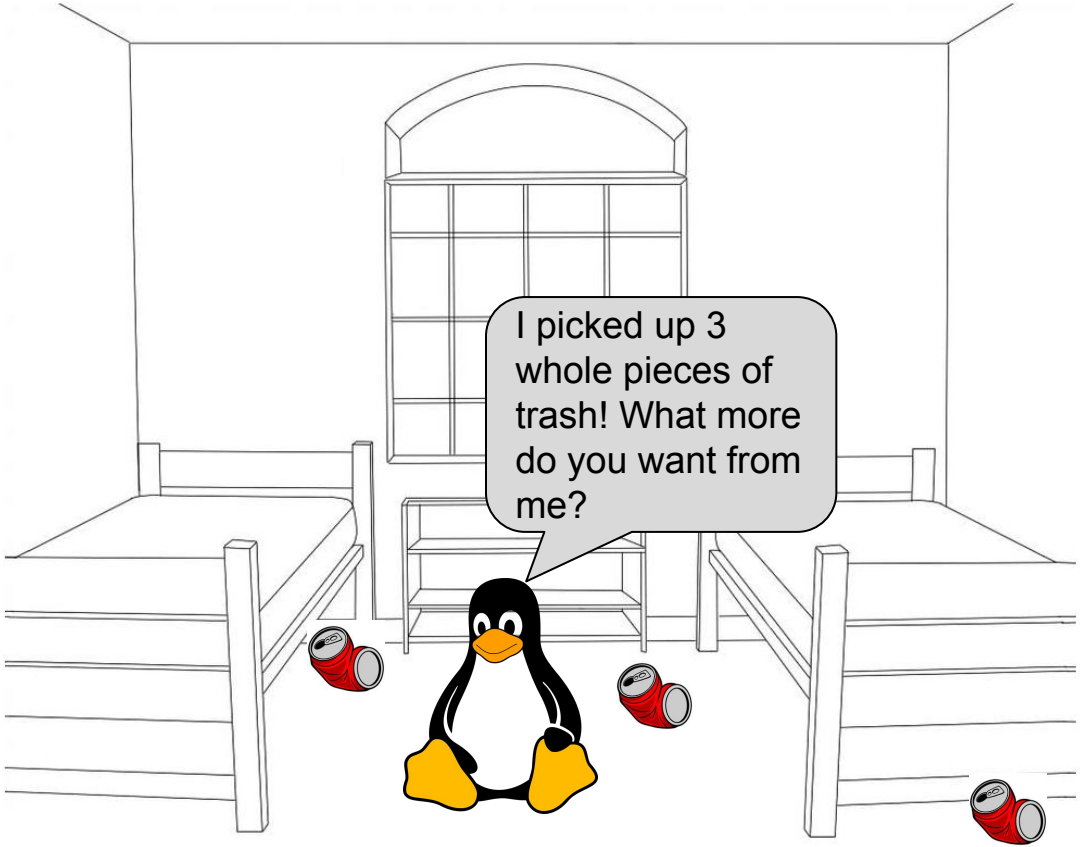


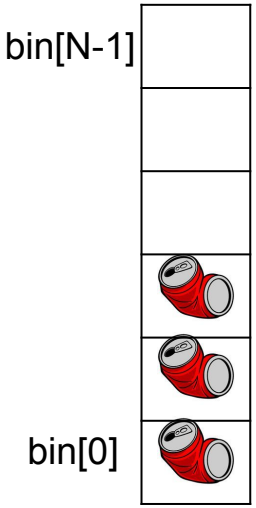
Stop asking them to clean the room! There's nothing to do.

```
num_trash pickup(roomNum, trashBin, Amount)
```

`num_trash == -1, errno == excuse`
`num_trash == 0`
`num_trash < Amount`
`num_trash == Amount`

How do we get room 333 clean?



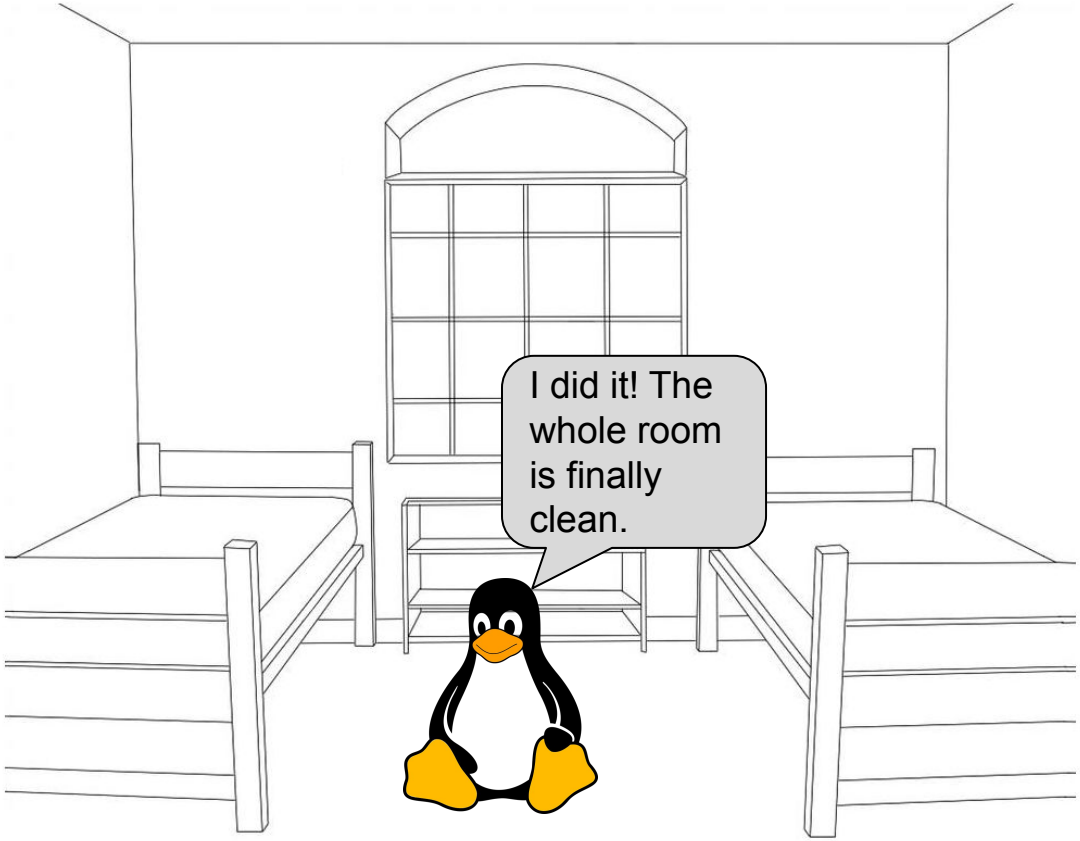


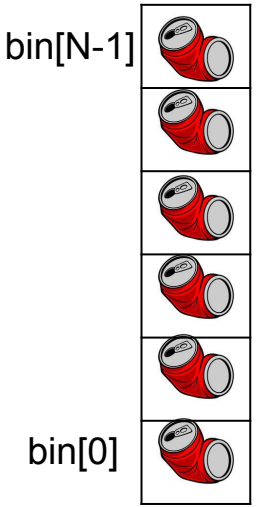
Ask them again to pick up the rest of it.

```
num_trash pickup(roomNum, trashBin, Amount)
```

How do we get room 333 clean?

`num_trash == -1, errno == excuse`
`num_trash == 0`
`num_trash < Amount`
`num_trash == Amount`





They did what you asked, so stop asking them to pick up trash.

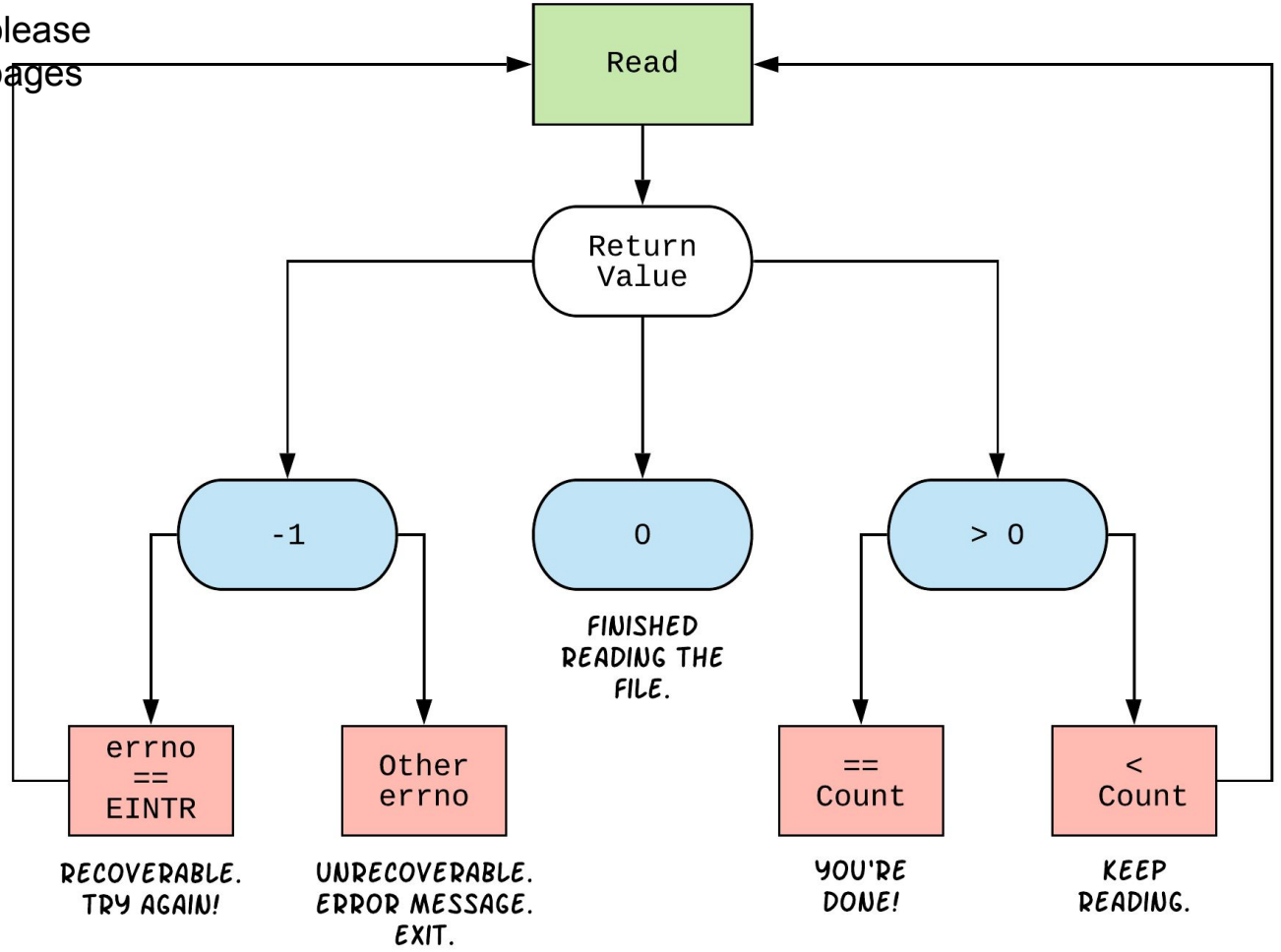
Review from Lecture – POSIX Read

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

An error occurred	`result == -1` `errno = error`
Nothing left to read (already at EOF)	`result == 0`
Partial Read	`result < count`
Success!	`result == count`

https://man7.org/linux/man-pages/man2/read.2.html

Not fully comprehensive, please refer to the man pages



Exercises 2-4

```
int open(char *name, int flags);
```

- ➔ `name` *is a string representing the name of the file. Can be relative or absolute.*
- ➔ `flags` *is an integer code describing the access. Some common flags are listed below:*
  - ◆ `O_RDONLY` – *Open the file in read-only mode.*
  - ◆ `O_WRONLY` – *Open the file in write-only mode.*
  - ◆ `O_RDWR` – *Open the file in read-write mode.*
  - ◆ `O_APPEND` – *Append new information to the end of the file.*
- ★ Returns an integer which is the file descriptor. Returns `-1` if there is a failure.

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

- ➔ `fd` *is the file descriptor (as returned by open()).*
- ➔ `buf` *is the address of a memory area into which the data is read or written.*
- ➔ `count` *is the* maximum *amount of data to read from or write to the stream.*
- ★ Returns the *actual* amount of data read from or written to the file.

```
int close(int fd);
```

```c
int fd = open("333.txt", O_WRONLY);  // open 333.txt
int n = ...;
char *buf = ...;  // Assume buf initialized with size n
int result;
char *ptr = buf;   // initialize variable for loop

...  // code that populates buf happens here

while (ptr < buf + n) {
    result = write(fd, ptr, buf + n - ptr);
    if (result == -1) {
        if (errno != EINTR && errno != EAGAIN) {
            // a real error happened, return an error result
            close(fd);  // cleanup
            perror("Write failed");
            return -1;
        }
        continue;  // EINTR or EAGAIN happened, so loop around and try again
    }
    ptr += result;  // update loop variable
}
close(fd);  // cleanup
```

(ℹ️) This is just ONE possible way to solve this exercise!

POSIX Analysis

3. Why is it important to store the return value from `write`? Why don't we check for a return value of `0` like `read`?

   `write` may not actually write all the bytes specified in count.

   The `0` case for reading was EOF, but writing adds length to your file and we know exactly how much we are trying to write.

4. Why is it important to remember to call `close` once you have finished working on a file?

   In order to free resources (*i.e.*, locks on those files, file descriptor table entries).

There is No One True Loop!!!

You will need to tailor your POSIX loops to the specifics of what you need.

Some design considerations:

- Read data in fixed-sized chunks or all at once?
  - Trade-off in disk accesses versus memory usage.
- What if we don't know `N` (how many bytes to read) ahead of time?
  - Keep calling **`read`** until we get `0` back (`EOF`).
  - Can determine N dynamically by tracking the number of bytes read and using **`malloc`**/**`realloc`** to allocate more space as we go.
  - This case comes up when reading/writing to the network (later in 333)!

Directories

Directories

- A directory is a special file that stores the names and locations of the related files/directories
  - This includes itself (`.`), its parent directory (`..`), and all of its children (*i.e.*, the directory's contents)
  - Take CSE 451 to learn more about the directory structure
- Accessible via POSIX (`dirent.h` in C/C++)
- Why might we want to work with directories in a program?

  List files, find files, search files, recursively traverse directories, etc.

POSIX Directory Basics

- POSIX defines operations for directory *traversal*
  - `DIR *` is not a file descriptor, but used similarly
  - `struct dirent` describes a directory entry
  - **`readdir`**`()` returns the ‘next’ directory entry, or `NULL` at end
- Error values (they also set `errno`):
  - `DIR *`**`opendir`**`(const char *name);` `// NULL`
  - `struct dirent *`**`readdir`**`(DIR *dirp);` `// NULL`
  - `int` **`closedir`**`(DIR *dirp);` `// -1`

`struct dirent`

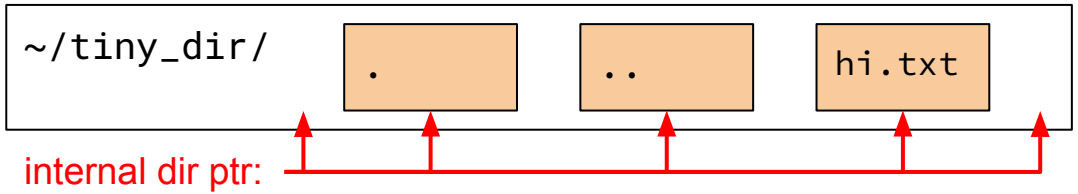
- Returned value from **`readdir`**
  - Does *not* need to be “freed” or “closed” 🎉
- Fields are “unspecified” (depends on your file system)
  - glibc specifies:

```
struct dirent {
  ino_t          d_ino;
  off_t          d_off;
  unsigned short d_reclen;
  unsigned char  d_type;
  char           d_name[256];
};
```

directory entry metadata stored in integer types

Null-terminated directory entry name (what we care about in 333)

readdir Example




```
DIR *dirp = opendir("~/tiny_dir");    // opens directory
struct dirent *file = readdir(dirp);  // gets ptr to "."
file = readdir(dirp);    // gets ptr to ".."
file = readdir(dirp);    // gets ptr to "hi.txt"
file = readdir(dirp);    // gets NULL
closedir(dirp);          // clean up
```

Exercise 5

Given the name of a directory, write a C program that is analogous to **ls**, *i.e.* prints the names of the entries of the directory to stdout. Be sure to handle any errors!

```c
int main(int argc, char** argv) {
  /* 1. Check to make sure we have a valid command line arguments */
   if (argc != 2) {
     fprintf(stderr, "Usage: ./dirdump <path>\n");
     return EXIT_FAILURE;
   }
  /* 2. Open the directory, look at opendir() */
   DIR *dirp = opendir(argv[1]);
   if (dirp == NULL) {
     fprintf(stderr, "Could not open directory\n");
     return EXIT_FAILURE;
   }
```

...

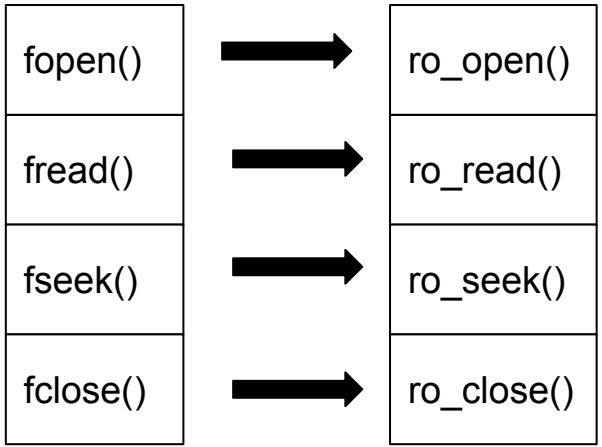
Given the name of a directory, write a C program that is analogous to **ls,** *i.e.* prints the names of the entries of the directory to stdout. Be sure to handle any errors!

```
...
  /* 3. Read through/parse the directory and print out file names
        Look at readdir() and struct dirent */
  struct dirent *entry;
  entry = readdir(dirp);
  while (entry != NULL) {
    printf("%s\n", entry->d_name);
    entry = readdir(dirp);
  }
  /* 4. Clean up */
  closedir(dirp);
  return EXIT_SUCCESS;
}
```

Ex4 Demo

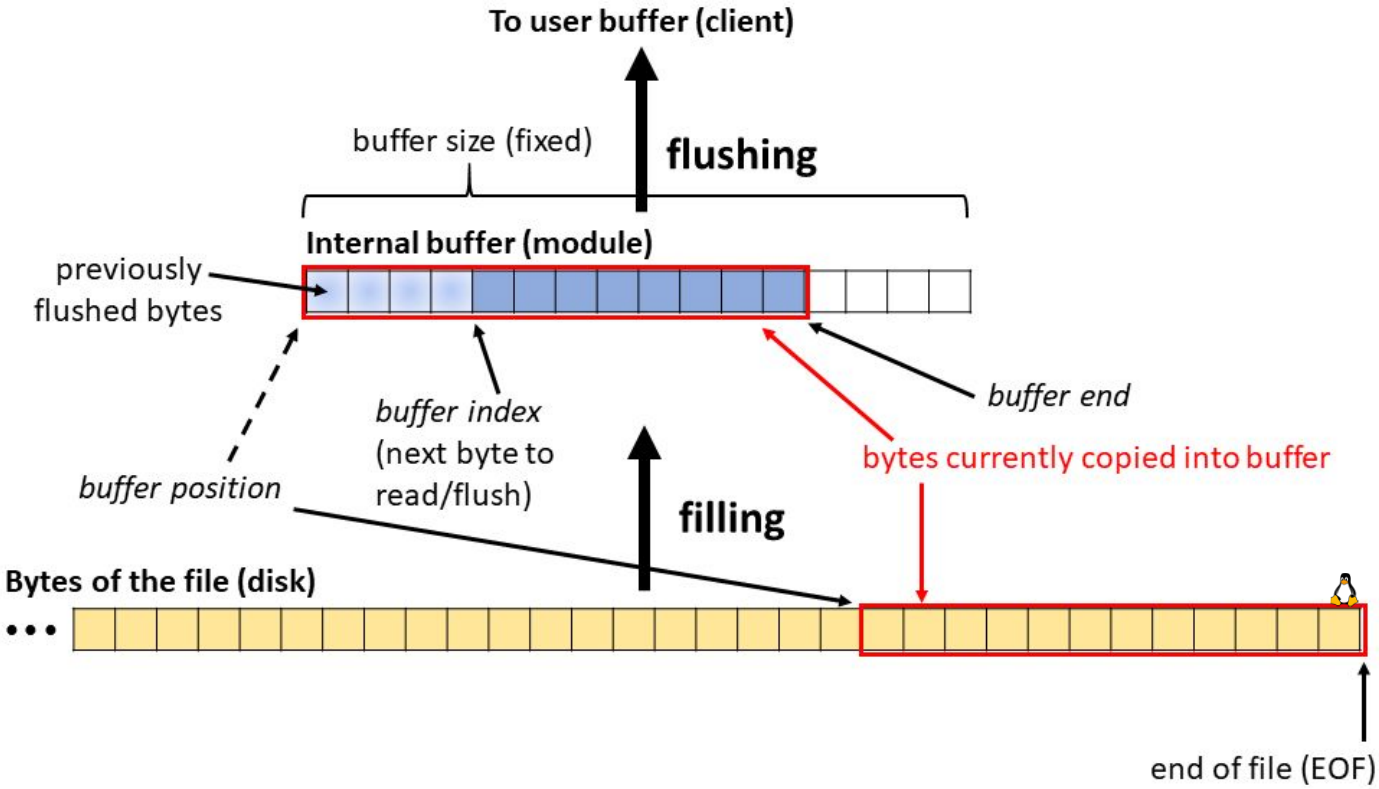
What you will be doing…

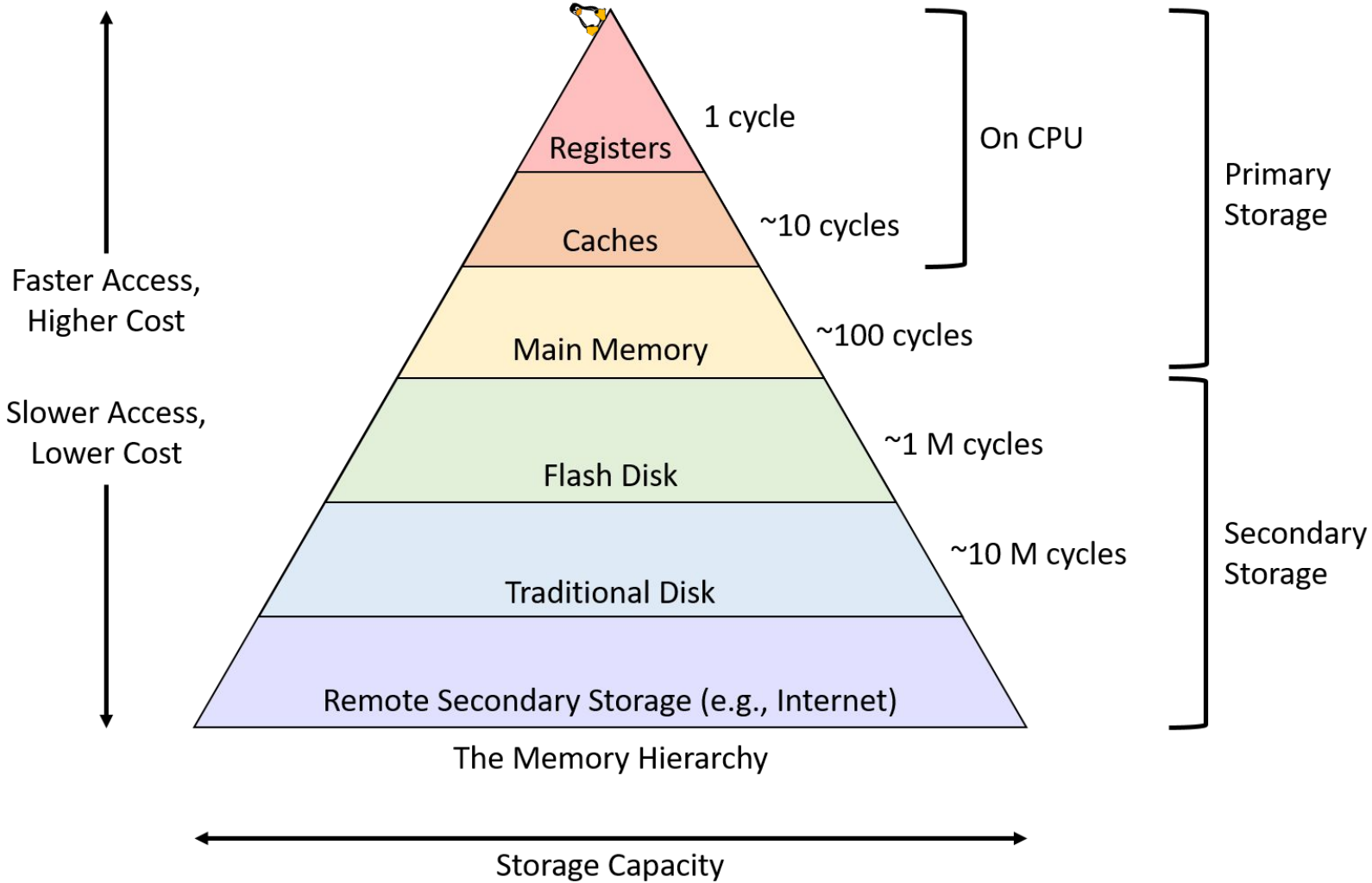
- Implementing your own file I/O library using only POSIX calls!
- https://courses.cs.washington.edu/courses/cse333/23au/exercises/ex04.html



- You will then use your library to read files from directories
- You will also need to implement an internal buffer, invisible to the client, within your implementation

Exercise 4 Internal Buffering




Faster Access,
Higher Cost
Slower Access,
Lower Cost
Registers
1 cycle
Caches
~10 cycles
Main Memory
~100 cycles
Flash Disk
~1 M cycles
Traditional Disk
~10 M cycles
Remote Secondary Storage (e.g., Internet)
On CPU
Primary
Storage
Secondary
Storage
The Memory Hierarchy
Storage Capacity

More Buffering

```
static const int RO_FILE_BUF_LEN = 512;  // do not modify

struct ro_file_st {
  int fd;         // The file descriptor we are currently managing.

  char* buf;      // Pointer to our internal buffer for this file.

  off_t buf_pos;  // The position in the file that the beginning of our
                  // internal buffer currently corresponds to.

  int buf_index;  // The index in our internal buffer that corresponds to the
                  // user's current position in the file.

  int buf_end;    // How many bytes currently in our internal buffer are
                  // actually from the file.
                  // To understand why this is important to track, consider
                  // the case when the file length is < RO_FILE_BUF_LEN.
};
```