Low-Level I/O & System Calls Intro

CSE 333

Instructor: Alex Sanchez-Stern

Teaching Assistants:

Audrey Seo

Deeksha Vatwani

Derek de Leuw

Katie Gilchrist

Administrivia

- Exercise 6 was due this morning
- Exercise 7 is out tomorrow, due on Friday
  - You'll cover some of it in sections tomorrow

- Today, we cover the materials for Exercise 7:
  - POSIX I/O for directories and reading data from files
  - Read a directory and open/copy text files found there
    - Copy *exactly* and ***only*** the bytes in the file(s). No extra output, no "formatting", no "titles", no other transformations.

Administrivia

- Homework 1 due on **Tomorrow @ at 11pm**

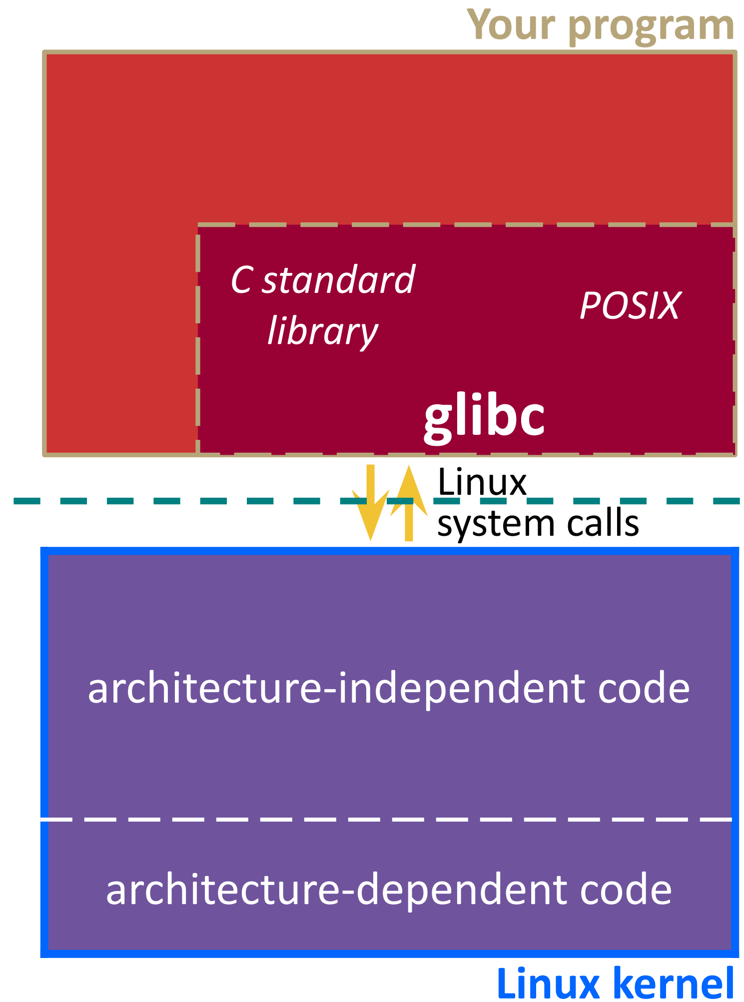
- What are two pieces of functionality that the OS provides to processes that run on it?
  - File System
  - Network Abstraction
  - Virtual Memory
  - Process Management
  - Portability

POSIX (Portable Operating System Interface)

- Standards for Unix-like operating system interfaces
- Maintained by the IEEE
- Allows more code to be portable across OS's
- Mostly handling:
  - I/O (including from files, terminals, and the network)
  - Threading

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 ☺



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

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

Why use POSIX File I/O?

- Same tasks on files can be accomplished with the C standard library API

- But they're often implemented in terms of POSIX operations
  - Helpful to understand how things work at a lower level
  - Can be more efficient to use the POSIX APIs
  - Generalizes beyond files (network, directories, etc).

Lecture Outline

- **Reading and Writing Files**
- Reading and Writing Directories
- System Calls Introduction

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 `f*()` 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

Many kinds of flags!
See the man page
for reference

```
#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…

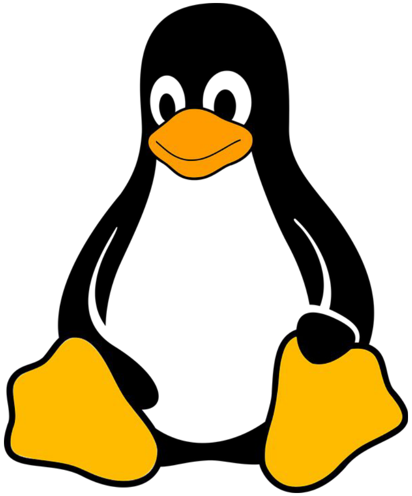
`ssize_t` is a signed version of `size_t`

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…
  ```

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

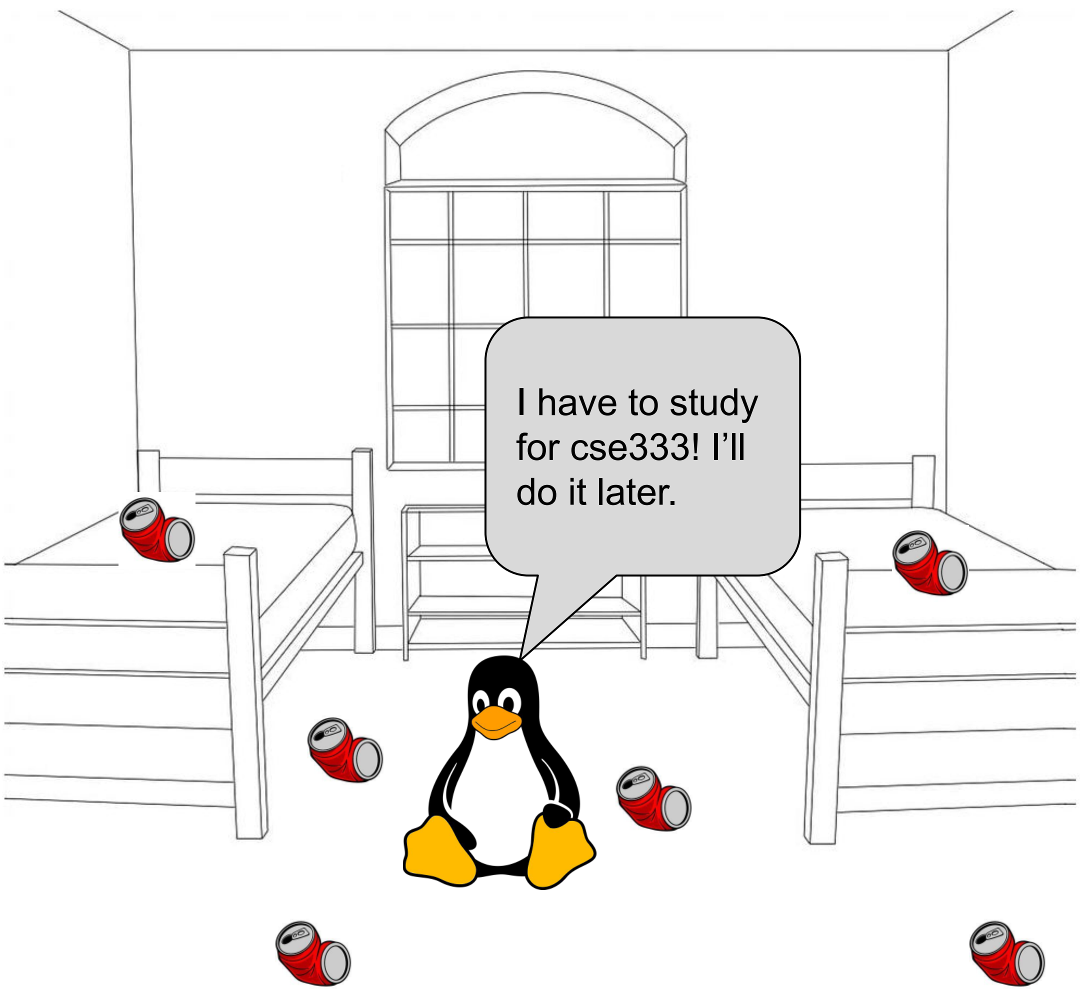
I/O Analogy – Messy Roommate

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

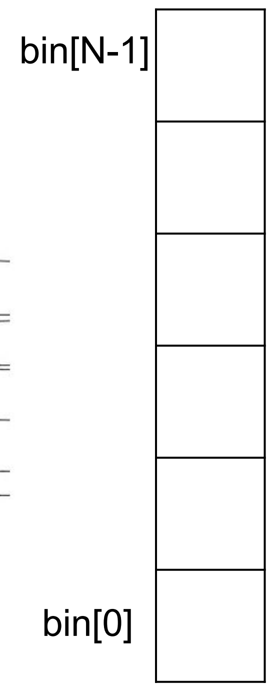
"*I tried to start cleaning, but something came up*" (got hungry, had a midterm, room was locked, etc.)	`num_trash == -1` `errno == excuse`
"*You told me to pick up trash, but the room was already clean*"	`num_trash == 0`
"*I picked up some of it, but then I got distracted by my favorite show on Netflix*"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

How do we get the room clean?

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



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

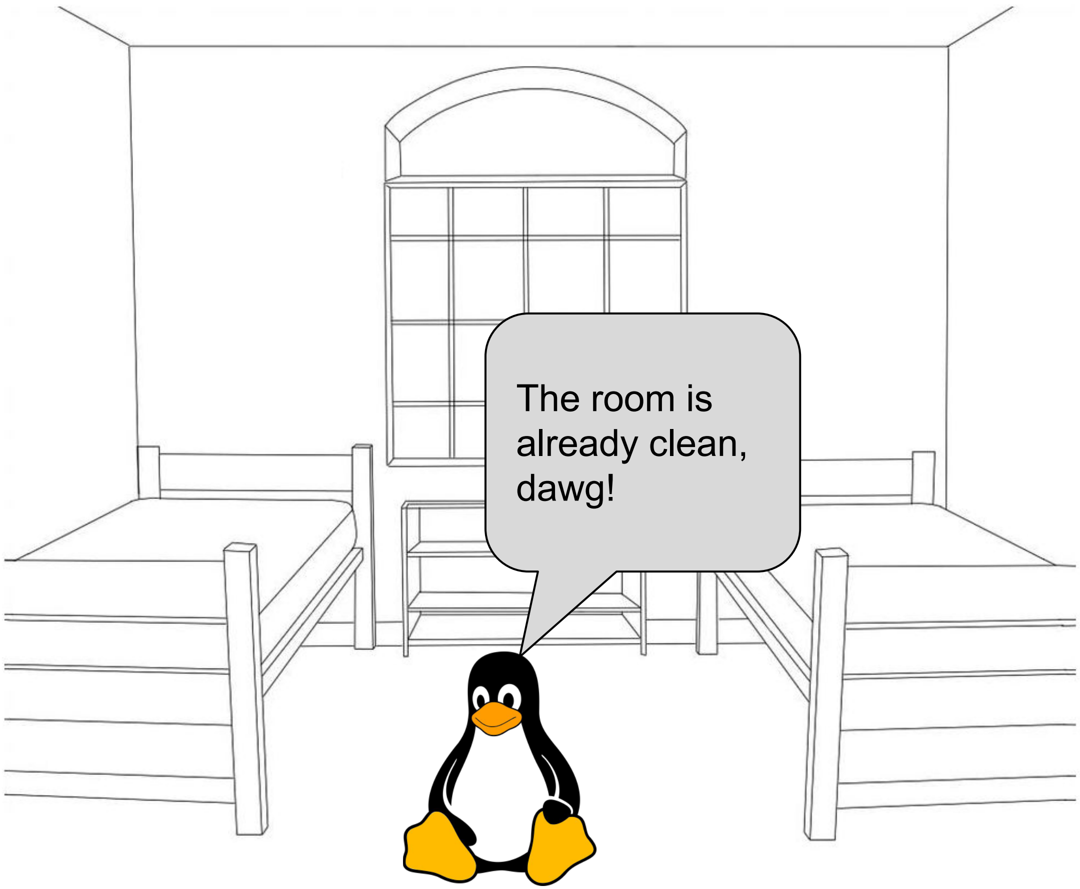


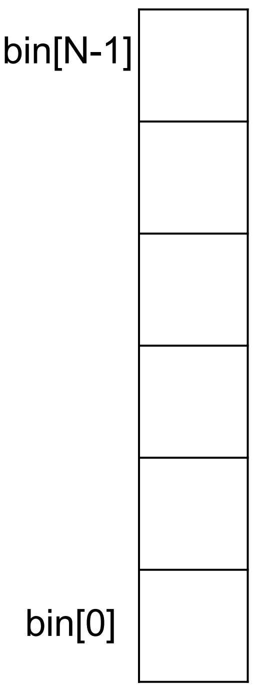
Decide if the excuse is reasonable, and either let it be or ask again.

How do we get the room clean?

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

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



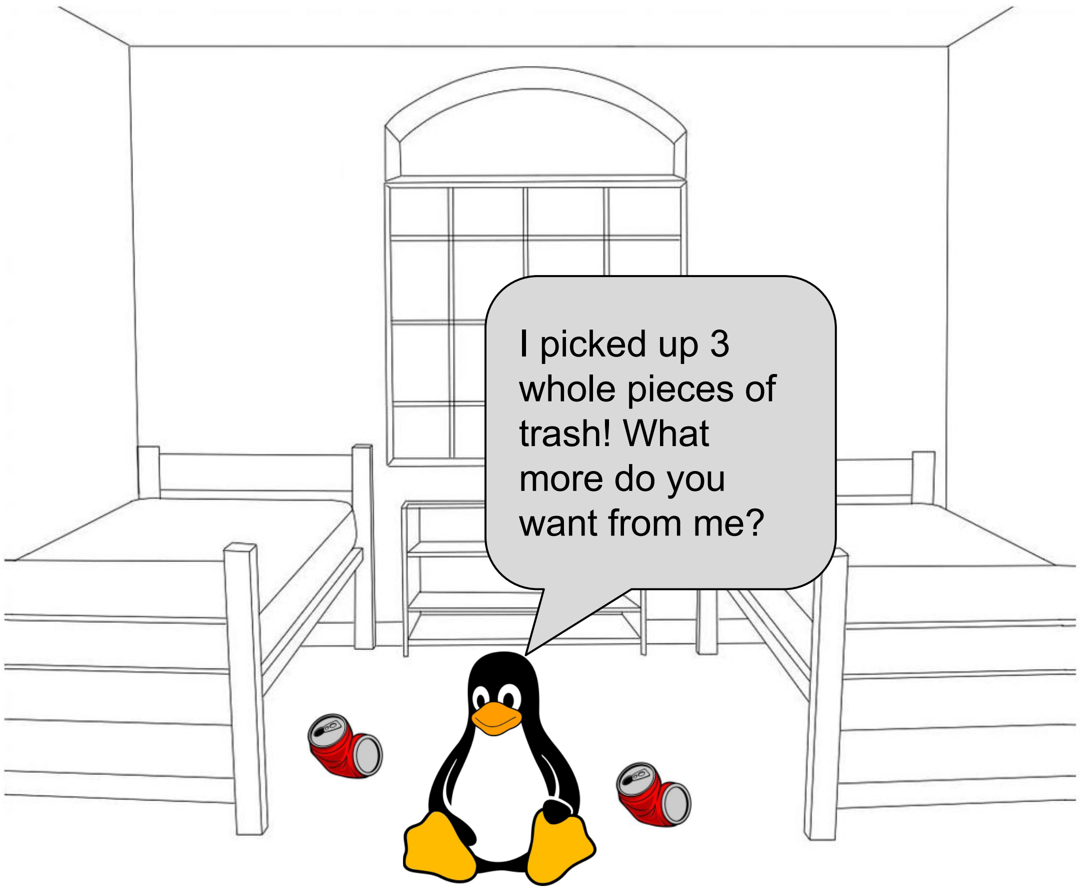


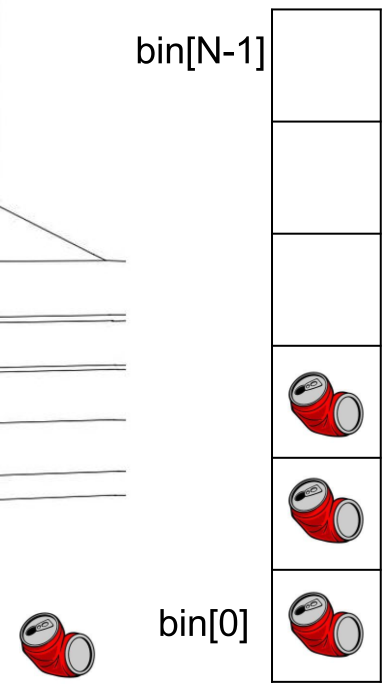
Stop asking them to clean the room! There's nothing to do.

How do we get the room clean?

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

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



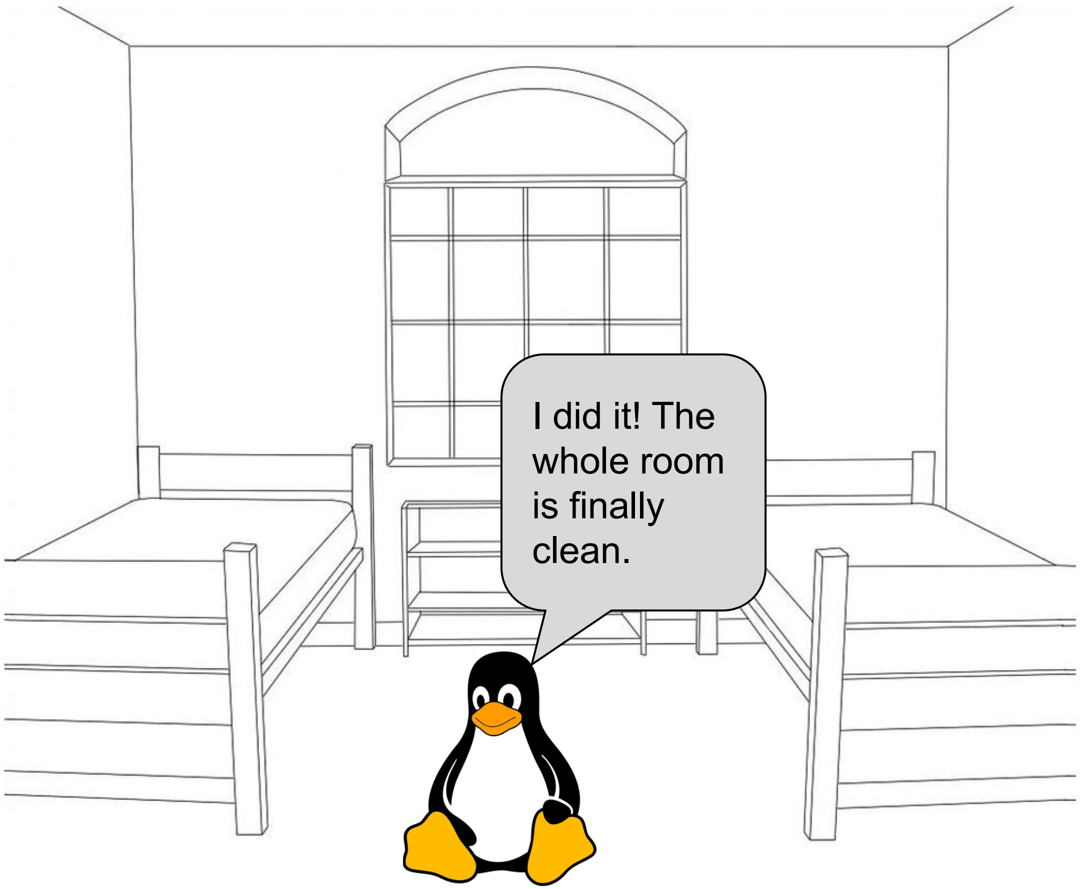


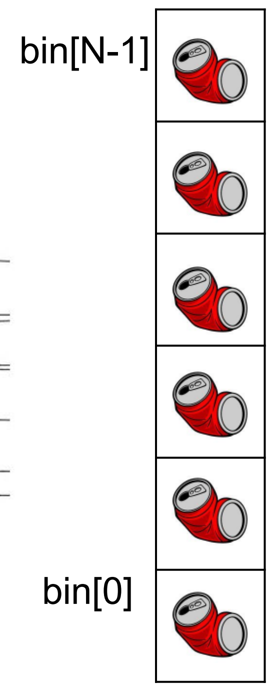
Ask them again to pick up the rest of it.

How do we get the room clean?

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

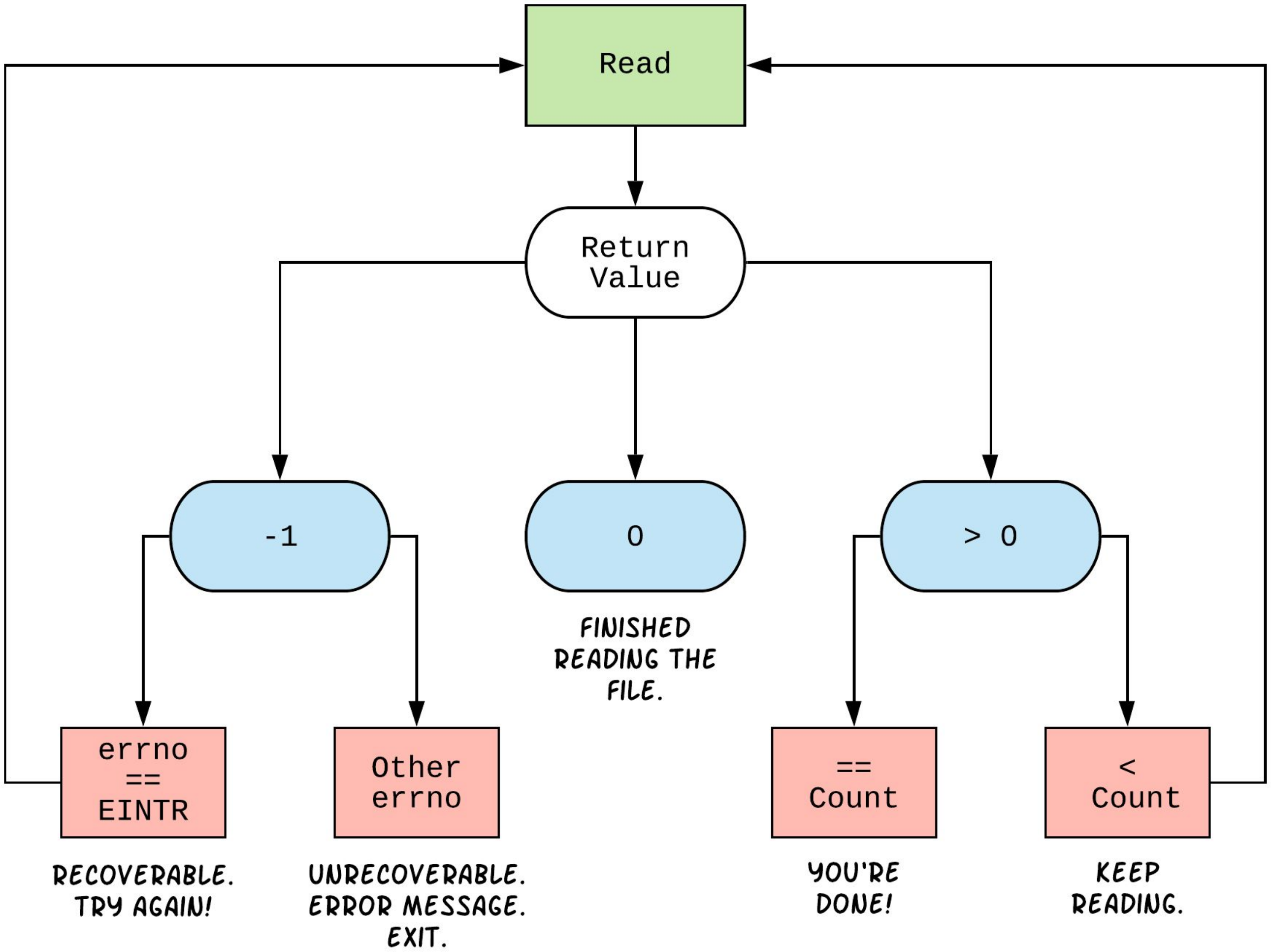
num_trash == -1, errno == excuse
num_trash == 0
num_trash < Amount
num_trash == Amount





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

Not fully comprehensive, please refer to the man pages



One way to read() n bytes

```
int fd = open(filename, O_RDONLY);
char* buf = ...;  // buffer of at least size n
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);
```

readN.c

Other Low-Level Functions

- Read man pages to learn more about POSIX I/O:
  - `write()` – write data
  - `fsync()` – flush data to the underlying device
    - Make sure you read the section 3 version (*e.g.* `man 3 fsync`)
- A useful shortcut sheet (from CMU): http://www.cs.cmu.edu/~guna/15-123S11/Lectures/Lecture24.pdf

Lecture Outline

- Reading and Writing Files
- **Reading and Writing Directories**
- System Calls Introduction

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++)
  - Basic operation is listing files/directories in a directory

POSIX Directory Basics

- Basic operations a lot like reading files
  - `opendir()` - Open a directory for reading
  - `readdir()` - Read the contents of a directory
  - `closedir()` - Close a directory when you're done
- Like C standard file I/O, but instead of `FILE *`, these use `DIR *`
  - `opendir()` returns a `DIR *`
  - `readdir()` and `closedir()` take a `DIR *`
- Instead of file bytes, reading a directory returns a `struct dirent`
  - describes a <u>dir</u>ectory <u>ent</u>ry

Full Prototypes

- `DIR *opendir(const char *name);`
- `struct dirent *readdir(DIR *dirp);`
- `int closedir(DIR *dirp);`

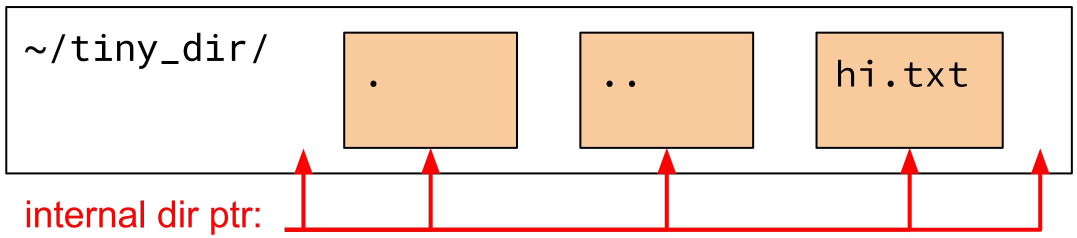
Return `-1` when it fails, and sets `errno`

Return `NULL` pointers when they fail, and set `errno`

Using `readdir()`

- The DIR * has **state**; it changes each time you read it.
- Each read returns one file or subdirectory, moves the DIR * to the next one
- After all directory contents have been read, returns NULL
  - Doesn't change errno if it's just the end of the directory

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
```

struct dirent

- Returned value from `readdir`
- Fields are "unspecified" (depends on your operating 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 (d_ino, d_off, d_reclen, d_type)

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

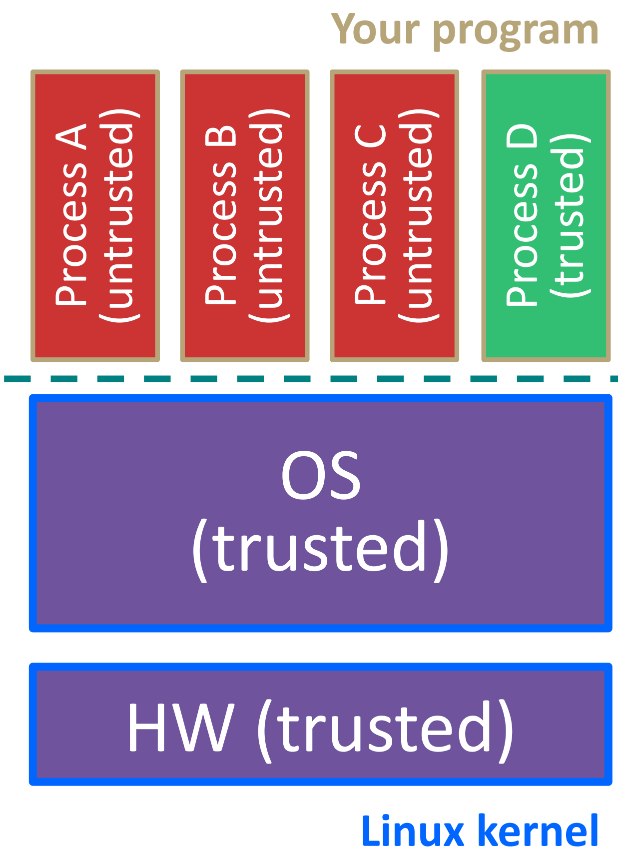
- Does *not* need to be "freed" or "closed"

Lecture Outline

- Reading and Writing Files
- Reading and Writing Directories
- **System Calls Introduction**

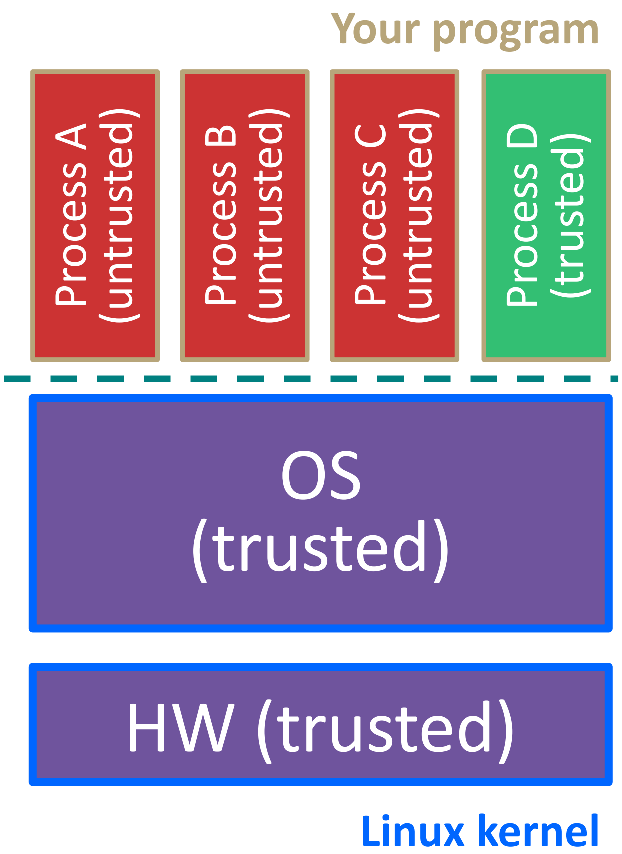
OS: Protection System

- OS isolates process from each other
  - But permits controlled sharing between them
    - Through shared name spaces (*e.g.* file names)
- OS isolates itself from processes
  - Must prevent processes from accessing the hardware directly



OS: Protection System

- The hardware has two important mechanisms to support OS functionality:
- Privileged Mode
  - Allows running special instructions that access the hardware directly
- Interrupts
  - Can be triggered by many kinds of events:
    - Timers, keypresses, etc.
  - Immediately causes the processor to jump to a pre-defined location, turns on privileged mode.

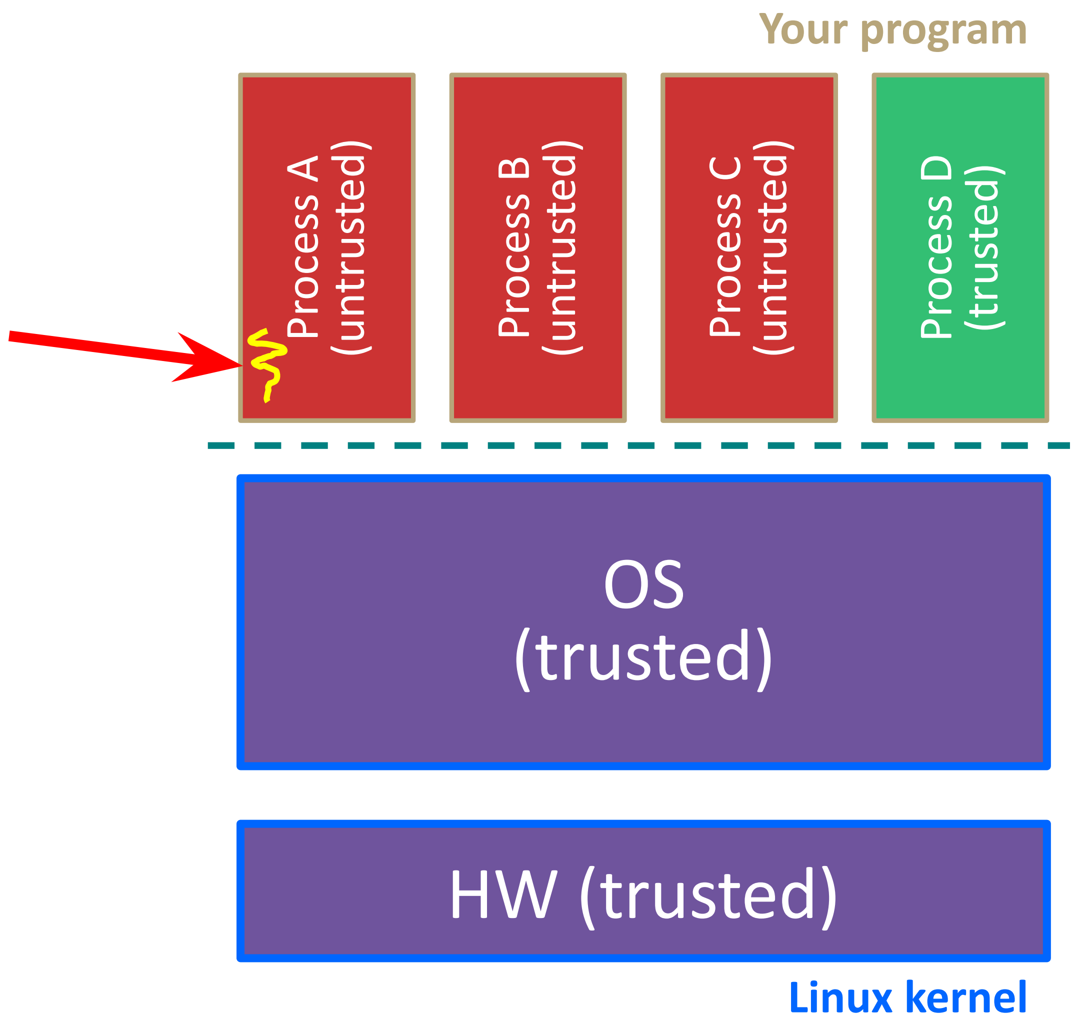


OS: Protection System

- User-level processes run with the CPU (processor) in unprivileged mode
- The OS runs with the CPU in privileged mode
- User-level processes invoke system calls by triggering an interrupt to safely enter the OS

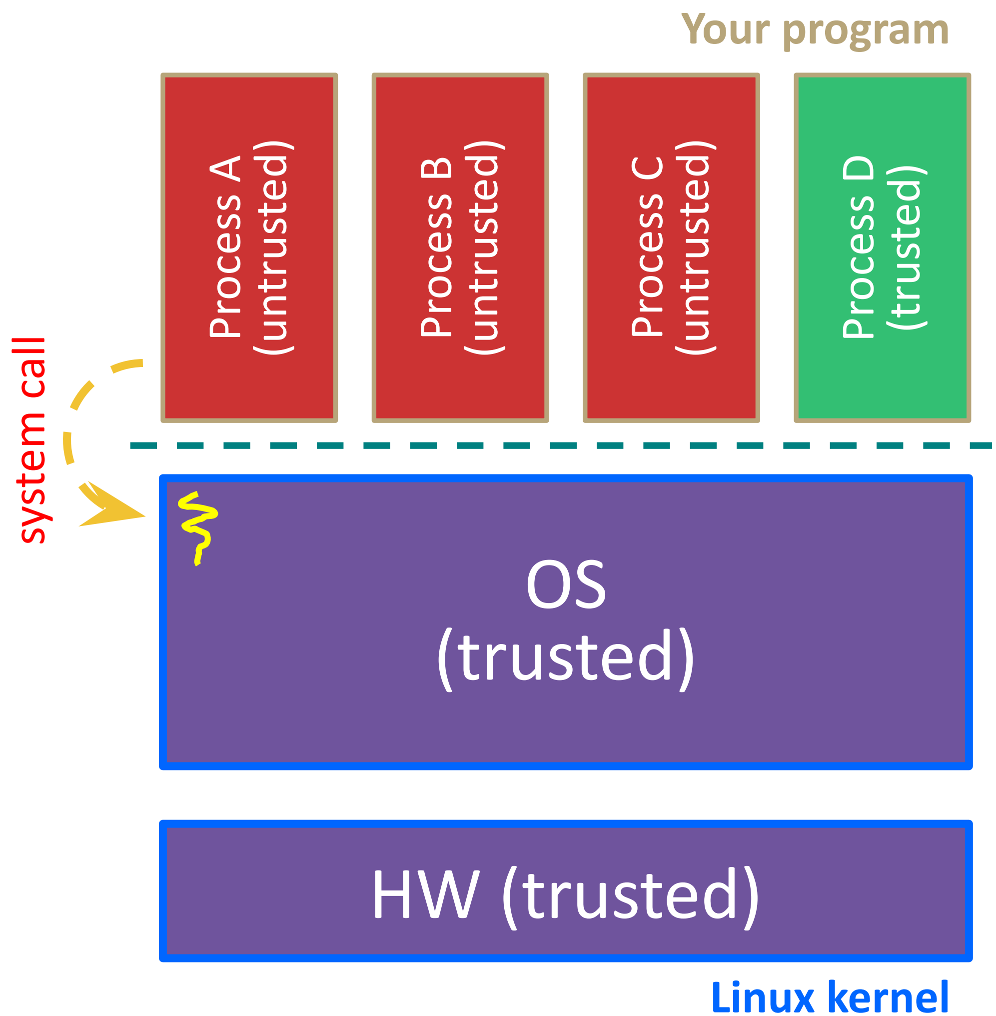
System Call Trace

A CPU (thread of execution) is running user-level code in Process A; the CPU is set to *unprivileged mode*.



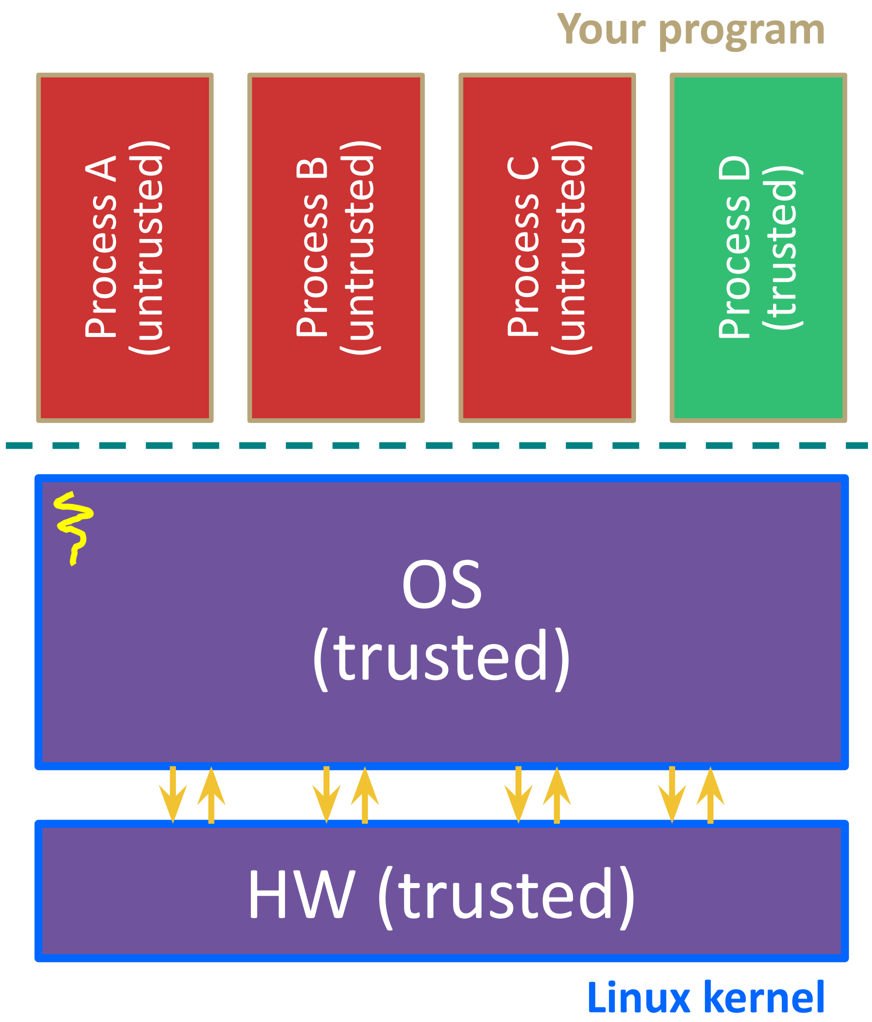
System Call Trace

Code in Process A invokes a system call; the hardware then:
(1) Sets the CPU to *privileged mode*
(2) Traps into the OS, which invokes the appropriate system call handler.



System Call Trace

Because the CPU executing the thread that's in the OS is in privileged mode, it is able to use *privileged instructions* that interact directly with hardware devices like disks.

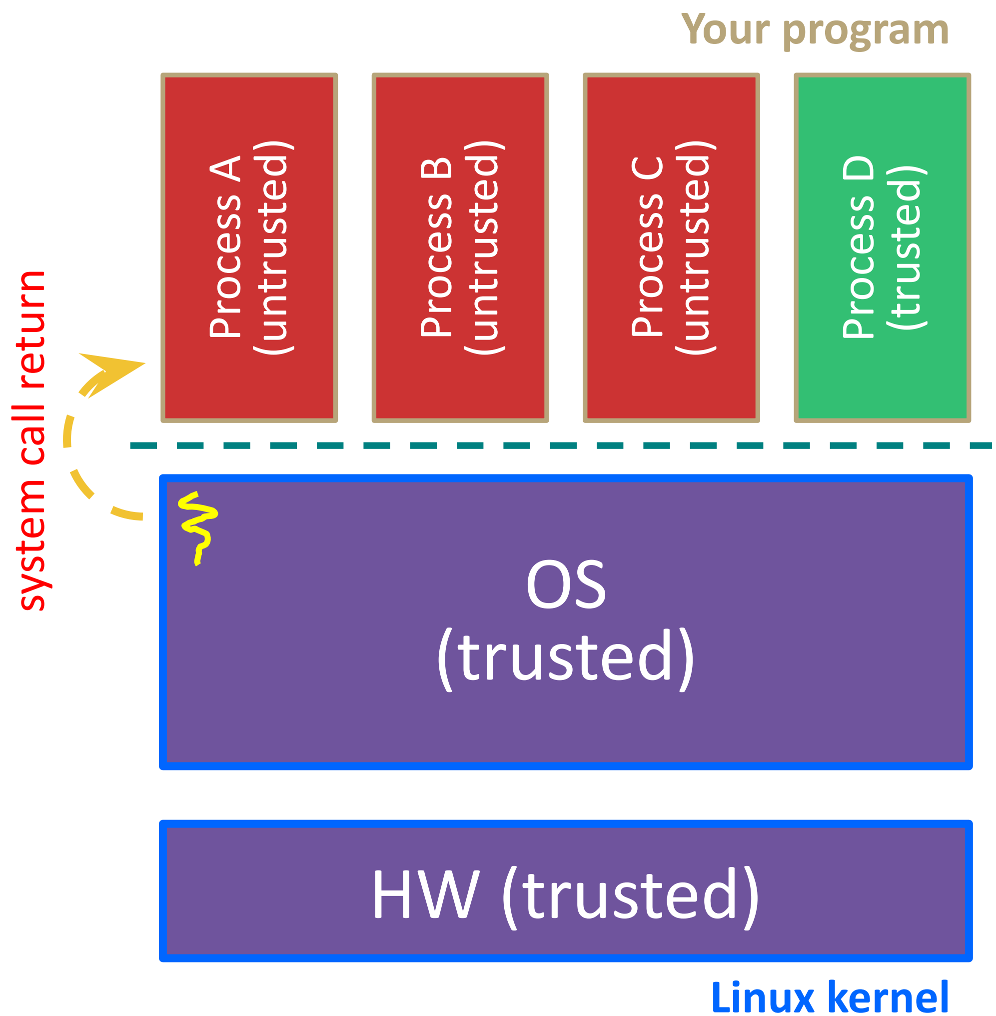


System Call Trace

Once the OS has finished servicing the system call, which might involve long waits as it interacts with HW, it:

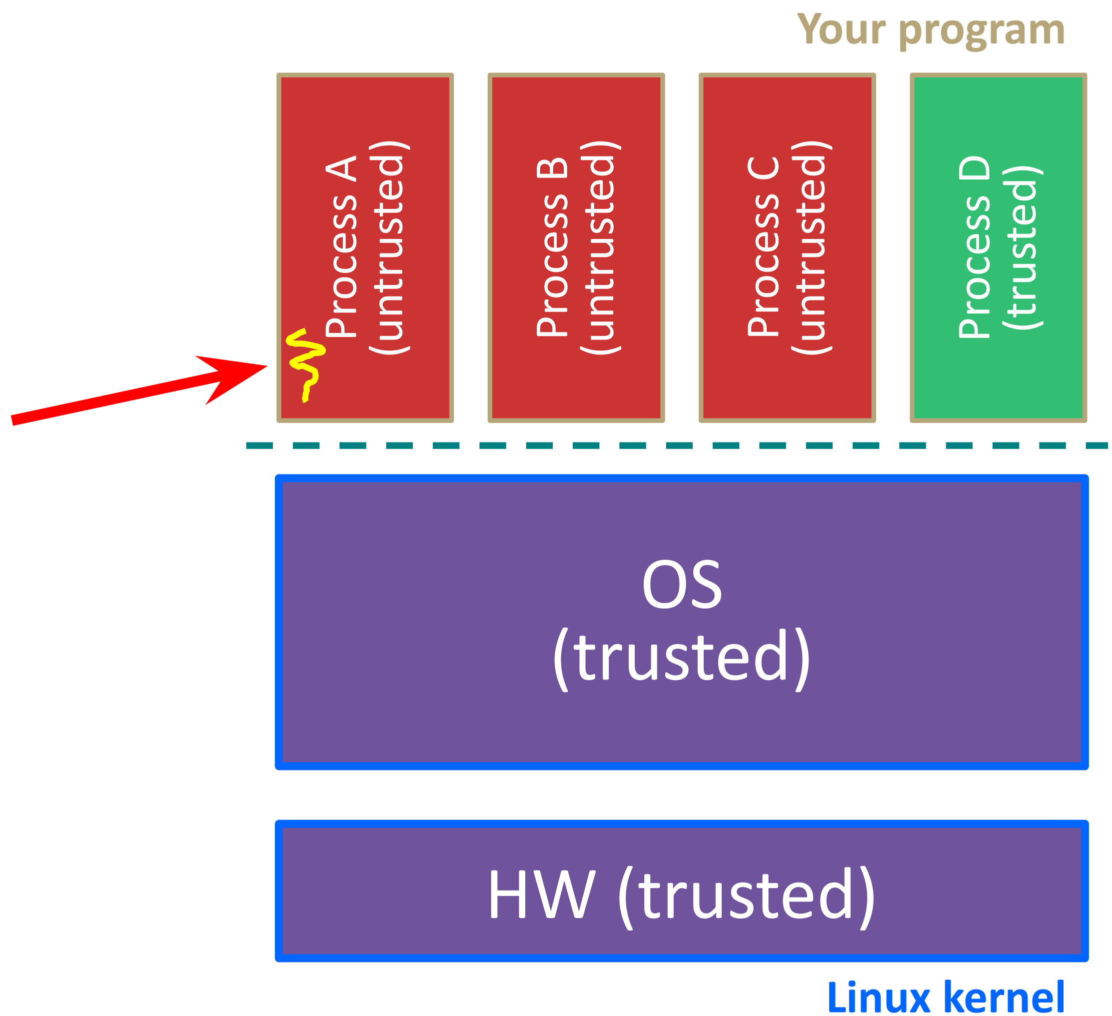
(1) Sets the CPU back to unprivileged mode and

(2) Returns out of the system call back to the user-level code in Process A.



System Call Trace

The process continues executing whatever code is next after the system call invocation.



Useful reference:
CSPP § 8.1–8.3
(the 351 book)

To do:

- HW1 due on **Tomorrow @ 11pm**
- Exercise 7 due Friday, but not out until tomorrow

Extra Exercise #1

- Write a program that:
  - Loops forever; in each loop:
    - Prompt the user to input a filename
    - Reads a filename from `stdin`
    - Opens and reads the file
    - Prints its contents to `stdout` in the format shown:

```
00000000 50 4b 03 04 14 00 00 00 00 00 9c 45 26 3c f1 d5
00000010 68 95 25 1b 00 00 25 1b 00 00 0d 00 00 00 43 53
00000020 45 6c 6f 67 6f 2d 31 2e 70 6e 67 89 50 4e 47 0d
00000030 0a 1a 0a 00 00 00 0d 49 48 44 52 00 00 00 91 00
00000040 00 00 91 08 06 00 00 00 c3 d8 5a 23 00 00 00 09
00000050 70 48 59 73 00 00 0b 13 00 00 0b 13 01 00 9a 9c
00000060 18 00 00 0a 4f 69 43 43 50 50 68 6f 74 6f 73 68
00000070 6f 70 20 49 43 43 20 70 72 6f 66 69 6c 65 00 00
00000080 78 da 9d 53 67 54 53 e9 16 3d f7 de f4 42 4b 88
00000090 80 94 4b 6f 52 15 08 20 52 42 8b 80 14 91 26 2a
000000a0 21 09 10 4a 88 21 a1 d9 15 51 c1 11 45 45 04 1b
... etc ...
```

- Hints:
  - Use `man` to read about `fgets`
  - Or, if you're more courageous, try `man 3 readline` to learn about `libreadline.a` and Google to learn how to link to it