# CS 344: Operating Systems I

# 02.01: I/O

M/W 12:00 – 1:50 PM (LINC #200)

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL

Secure AI Systems Lab

# Notice

- Announcements
  - Quiz answers will be available after all three attempts

# Notice

- Deadlines (~2 weeks)
  - (2/06 11:59 PM) Programming assignment 2
  - (2/13 11:59 PM) Midterm quiz 2

# Recap: Filesystem Structure Overview

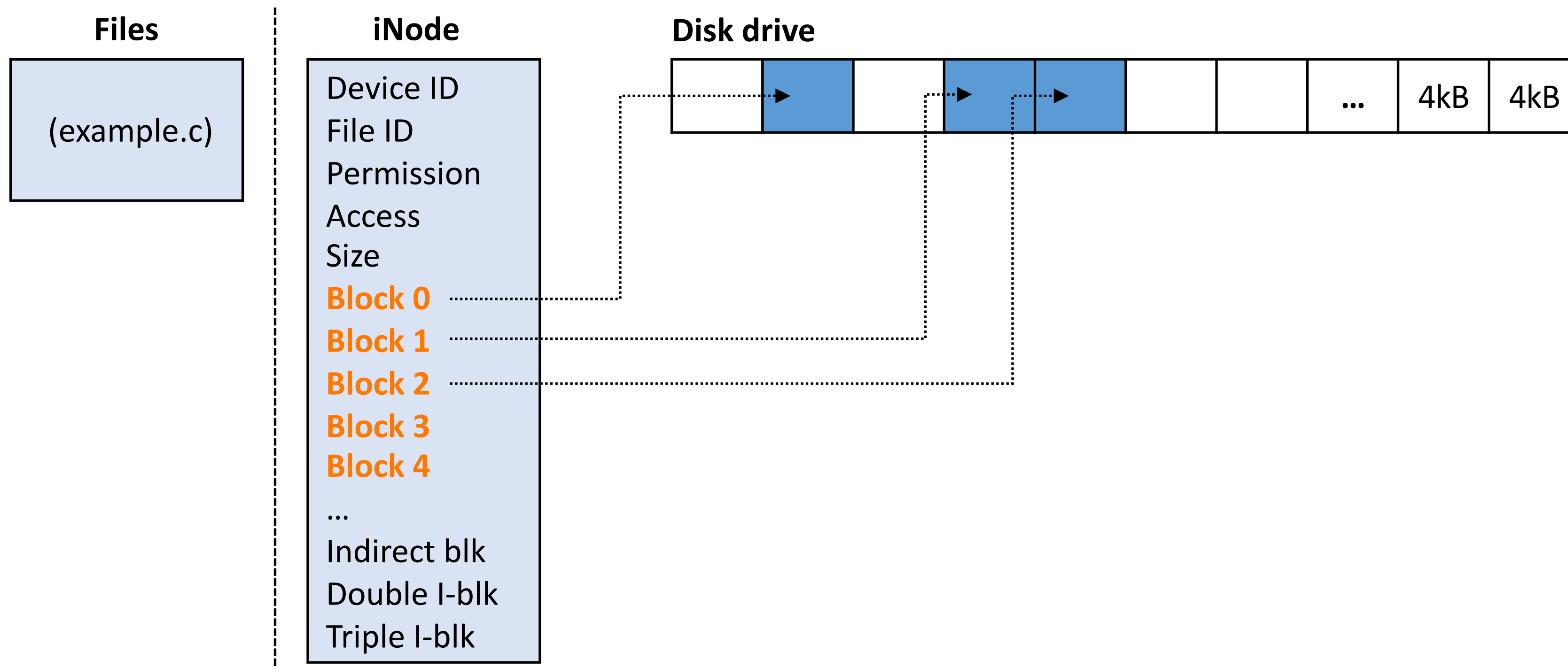
- Basic components
  - File : a named collection of data
  - Directory: a file that holds other files as data

- Access control, permission
  - Access control: user, group, and others (u, g, o)
  - Permission : read, write, and execute (r, w, x)

- Filesystem structure
  - iNode: a data-structure that describes a file-system object
  - Block : a unit of data storage, the size is defined by OS (e.g., 4kB)

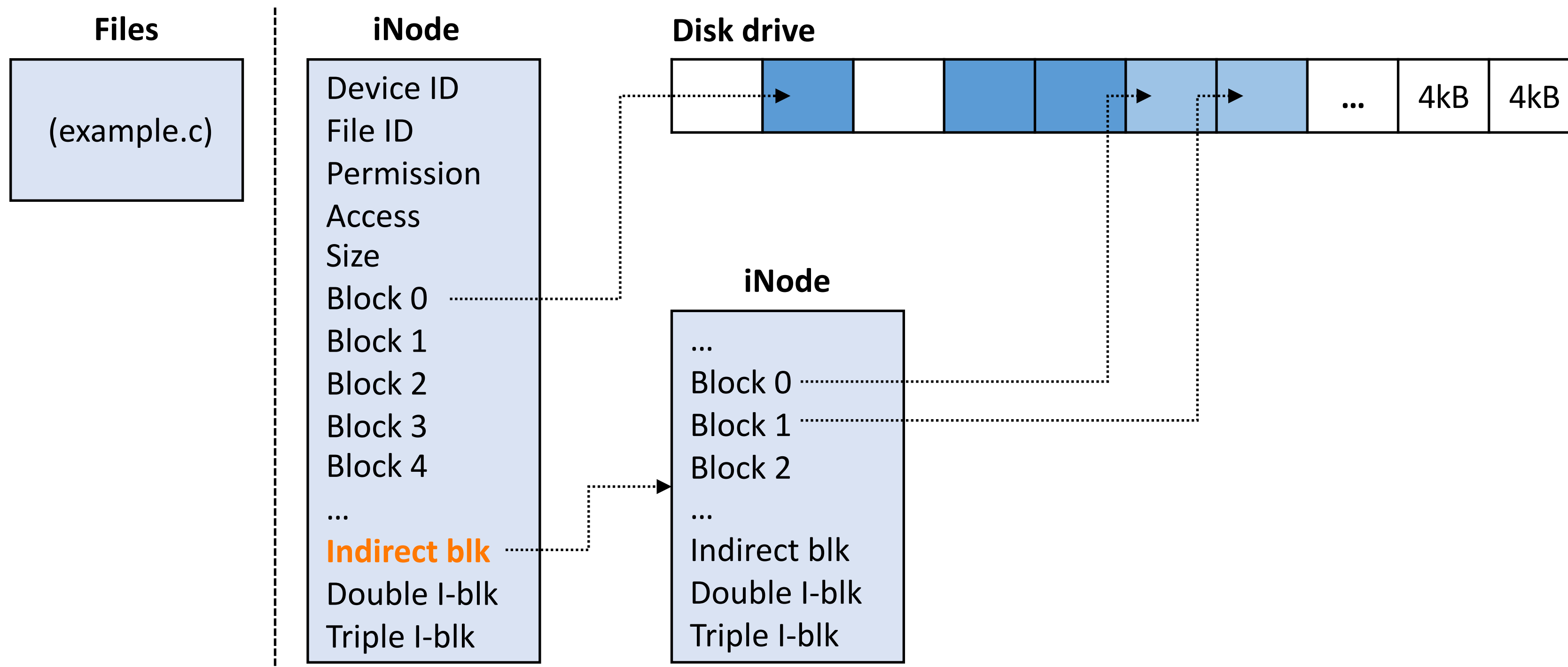
# Filesystem structure overview – cont'd

- A file stored in a filesystem (12 blocks ≈ 48kB)



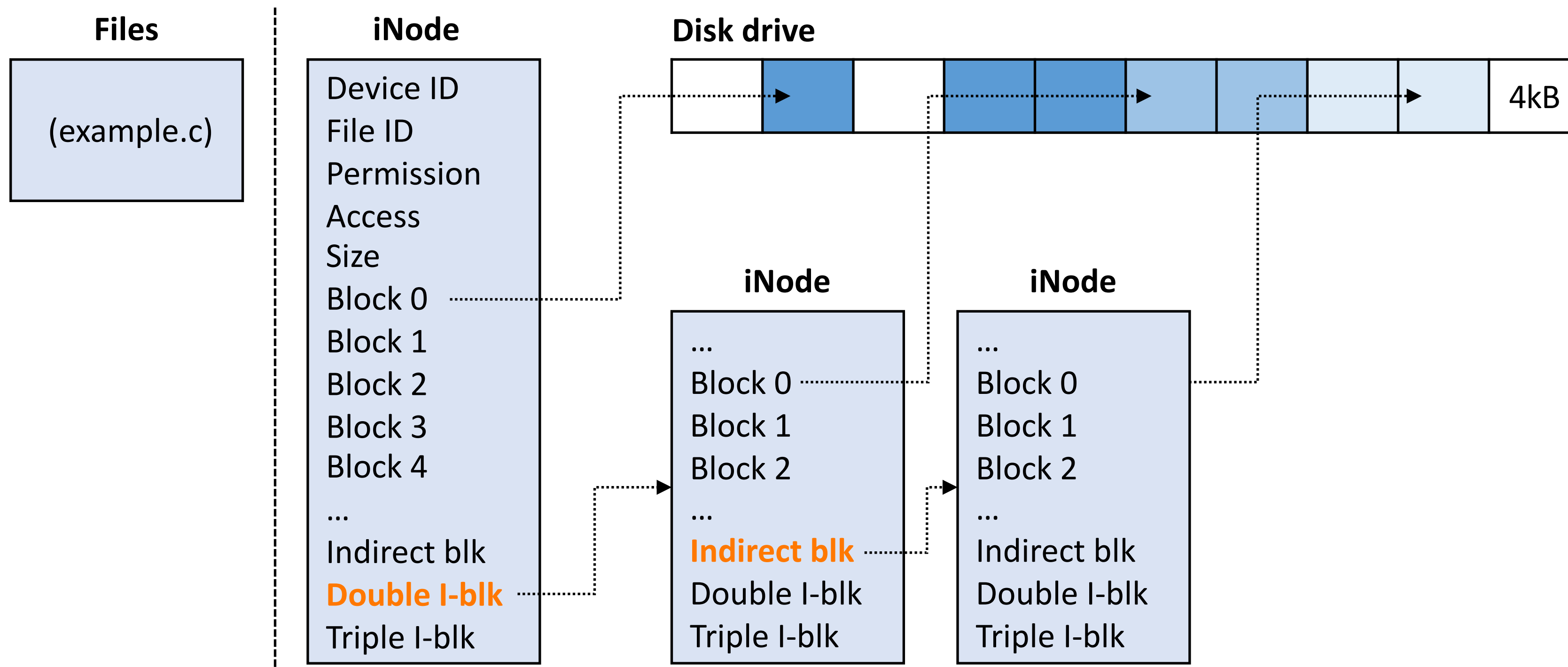
# FILESYSTEM STRUCTURE OVERVIEW – CONT'D

- A (larger) file stored in a filesystem (indirect block ≈ 4MB + 4kB)



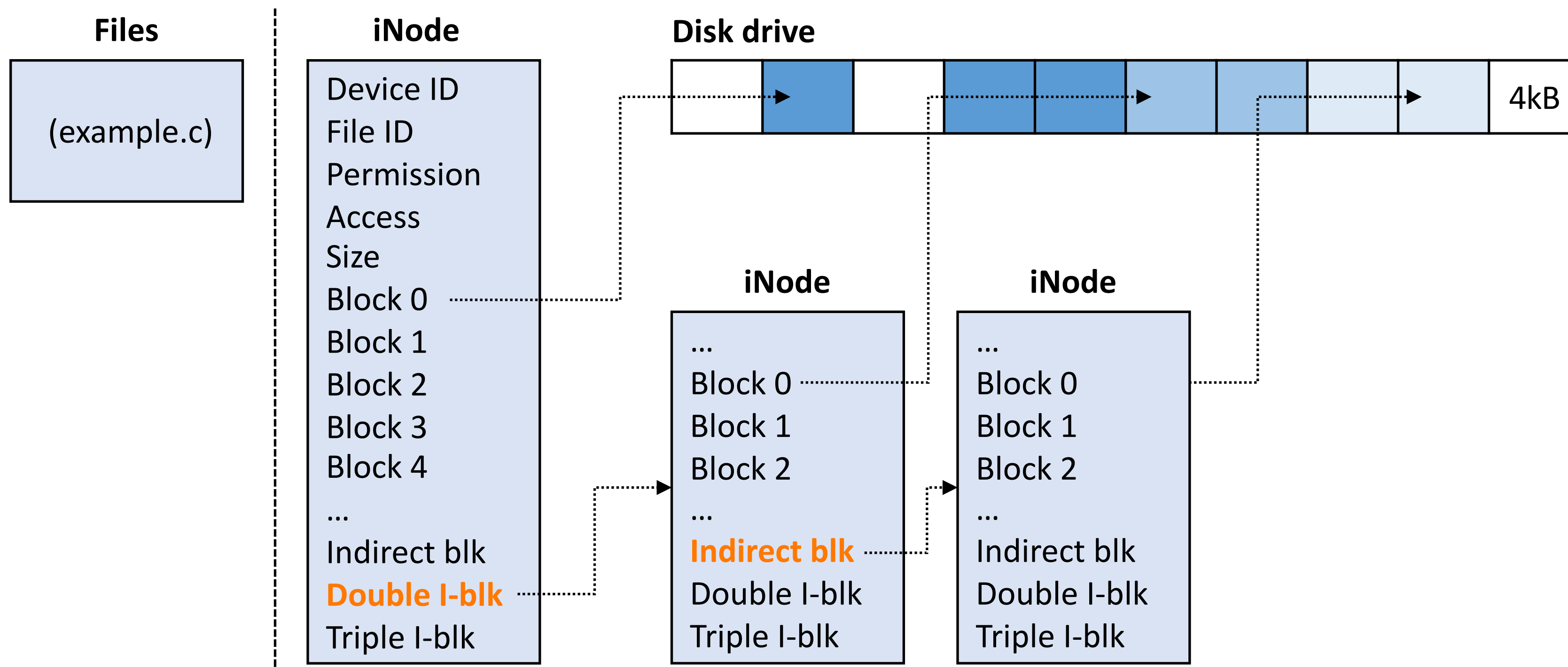
# Filesystem structure overview – cont'd

- A (larger) file stored in a filesystem (double I-blk ≈ 4GB +4MB +4kB)



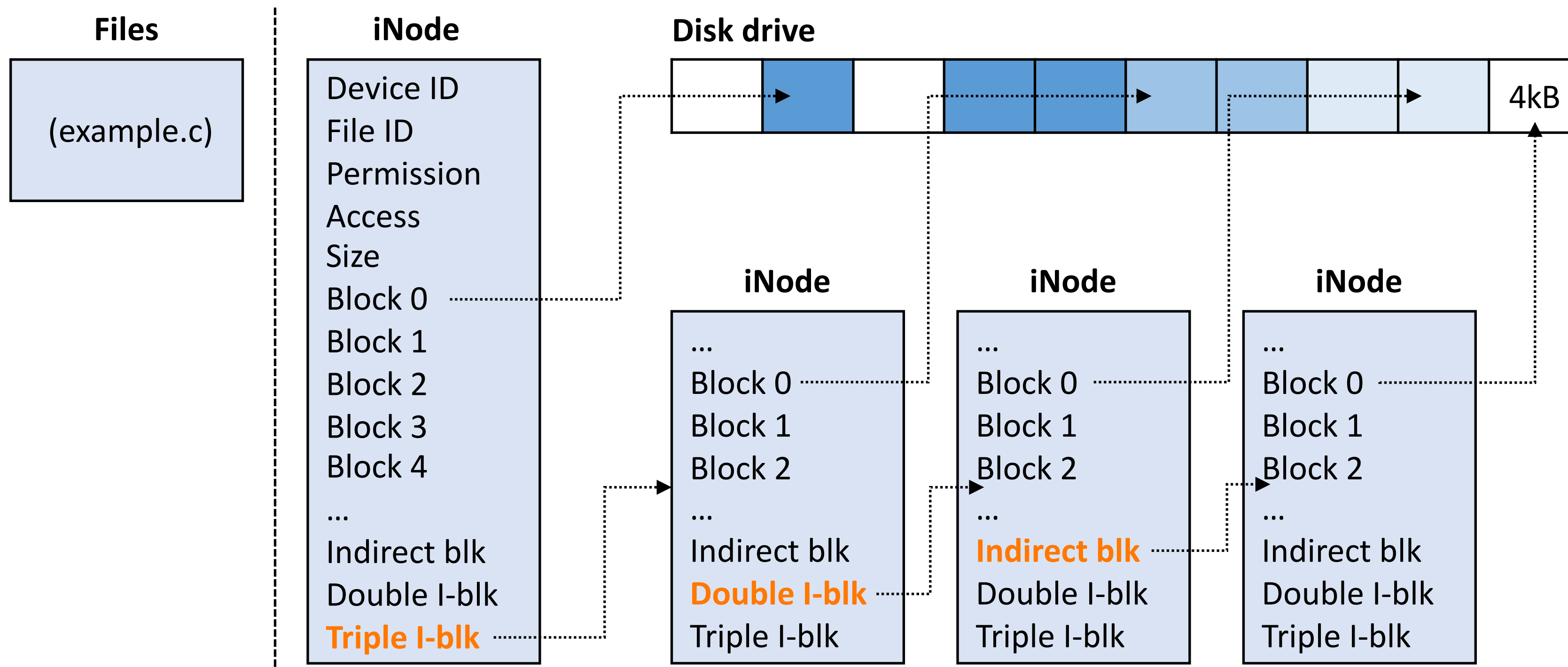
# Filesystem structure overview – cont'd

- A (larger) file stored in a filesystem (double I-blk ≈ 4GB +4MB +4kB)



# Filesystem structure overview – cont'd

- A (largest) file stored in a filesystem (triple I-blk ≈ 4TB +4GB +4MB +4kB)



# Filesystem structure overview – cont'd

- Design choices
  - FAT :
    - Index: Linked lists (iNode)
    - Data : Block
  - NTFS:
    - Index: Tree (iNode)
    - Data : Extent

# Topics for today

- Part II: I/Os
  - Provide abstractions
    - What is I/O?
  - Offer standard interface
    - How can we do low-level I/Os?
    - How can we do high-level I/Os?
  - Manage resources
    - How OS manages (file) I/O internally?

# Provide abstraction: what is I/O?

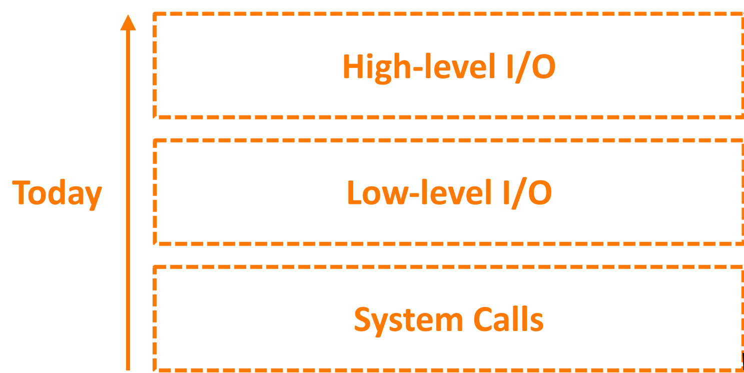
- I/O
  - **Definition :** input and output
  - **Def (*NIX):** any operation that
    read/write from/to system services
    (*NIX OS: everything is a file)

# Topics for today

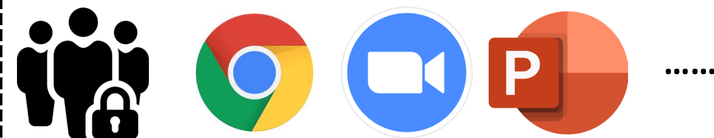
- Part II: I/Os
  - Provide abstractions
    - What is I/O?
  - Offer standard interface
    - How can we do low-level I/Os?
    - How can we do high-level I/Os?
  - Manage resources
    - How OS manages (file) I/O internally?

# Offer standard interface

- I/O
  - **Definition :** input and output
  - **Def (*NIX):** any operation that read/write system services (*NIX OS: everything is a file)




Users Run Applications




Standard Interfaces (Libraries)

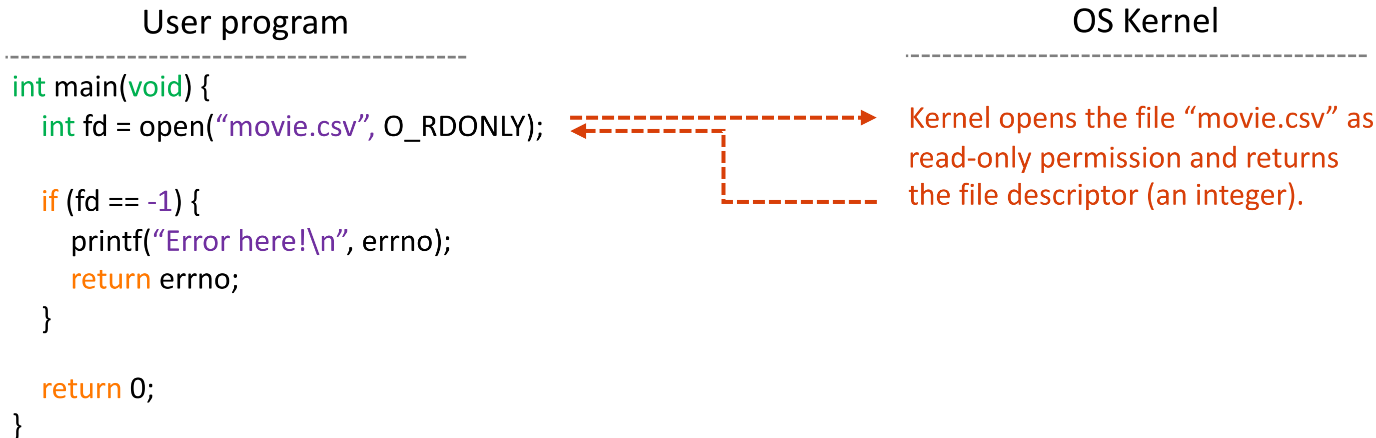
File System(s)

I/O Drivers

Hardware (CPU, GPU, Mem, …)

# Recap: System Call

- System call
  - **Definition:** a user-level function call to request a service from the OS
  - **I/O system calls:**
    - int open(const char **pathname*, int *flags*)
    - int creat(const char **pathname*, mode_t *mode*)
    - Int openat(int dirfd, const char **pathname*, int *flags*, mode_t *mode*)



# Offer standard interface: low-level I/O

- File descriptors (fd)
  - **Definition** : an integer that uniquely identifies an open file in Linux
  - **System calls:** (fctrl.h)
    - int open( const char *filename, int flags, mode_t *mode )
    - int create( const char *filename, mode_t *mode )
    - int close(int *fd )

  - **Standard file descriptors:**
    - STDIN_FILENO : **0**
    - STDOUT_FILENO: **1**
    - STDERR_FILENO : **2**

# Offer standard interface: low-level I/O

- File descriptors (fd)
  - **Definition** : an integer that uniquely identifies an open file in Linux
  - **System calls:**
    - int open( const char *filename, int flags, mode_t *mode )
      - Open the file and return a file descriptor
      - Returns error (link) if it fails to open the file
      - **flags** : access mode (O_RDONLY, O_APPEND, …)
      - **mode:** access permission (S_IRUSR, S_IRWXU, ...)
    - int create( const char *filename, mode_t *mode )
    - int close(int *fd )

# Offer standard interface: read from a file descriptor

- Basic functions
  - ssize_t read( int fd, void *buffer, size_t maxsize )

- Descriptions
  - read(): reads data from an open file using its file descriptor
    - Read **up to maxsize bytes**; returns less bytes if the data < maxsize
    - Return the number of bytes it read (0 means **EOF**, and negative values are errors)

# Offer standard interface: read from a file descriptor

- Basic functions
  - ssize_t read( int fd, void *buffer, size_t maxsize )
  - ssize_t write( int fd, const void *buffer, size_t size )

- Descriptions
  - read(): reads data from an open file using its file descriptor
    - Read **up to maxsize bytes**; returns less bytes if the data < maxsize
    - Return the number of bytes it read (0 means **EOF**, and negative values are errors)
  - write(): writes data to an open file using its file descriptor
    - Returns the number of bytes it wrote

# Offer standard interface: read from a file descriptor

- Basic functions
  - ssize_t read( int fd, void *buffer, size_t maxsize )
  - ssize_t write( int fd, const void *buffer, size_t size )
  - off_t lseek( int fd, off_t offset, int whence )

- Descriptions
  - read(): reads data from an open file using its file descriptor
    - Read **up to maxsize bytes**; returns less bytes if the data < maxsize
    - Return the number of bytes it read (0 means **EOF**, and negative values are errors)
  - write(): writes data to an open file using its file descriptor
    - Returns the number of bytes it wrote
  - lseek(): repositions the file offset within the kernel
    - (lseek != fseek) fseek holds a position in the FILE pointer

# Offer standard interface: read/write from a file descriptor

- Basic functions
  - ssize_t read( int fd, void *buffer, size_t maxsize )
  - ssize_t write( int fd, const void *buffer, size_t size )
  - off_t lseek( int fd, off_t offset, int whence )

**Data types** (size_t, off_t, …)**:**
C has many pre-defined data types. You can find them in <types.h>; a friendly version can be found in here (link)

- Descriptions
  - read(): reads data from an open file using its file descriptor
    - Read **up to maxsize bytes**; returns less bytes if the data < maxsize
    - Return the number of bytes it read (0 means **EOF**, and negative values are errors)
  - write(): writes data to an open file using its file descriptor
    - Returns the number of bytes it wrote
  - lseek(): repositions the file offset within the kernel
    - (lseek != fseek) fseek holds a position in the FILE pointer

# Offer standard interface: low-level I/O

- Example C code:

```
#include <stdio.h>
#include <unistd.h>
#include <fctrl.h>

#define BUFFER_SIZE     256

int main(void) {
   char *buffer = (char *) calloc(BUFFER_SIZE * sizeof(char));

   int fd = open("input.txt", O_RDONLY, S_IRUSR | S_IWUSR);

   ssize_t rlen = read(fd, buffer, BUFFER_SIZE);

   int err = close(fd);

   ssize_t wlen = write(STDOUT_FILENO, buffer, rlen);

   return 0;
}
```

**open() system call:**
It opens a file with the read-only permission.
A user can read/write from/to this file descriptor.

**read() system call:**
It reads at most, BUFFER_SIZE bytes from the opened file and returns the total bytes read (*rlen*).

**write() system call:**
It writes the contents in the buffer to the standard output (Term. screen). It will write *rlen* bytes.

# Offer standard interface: low-level I/O system calls

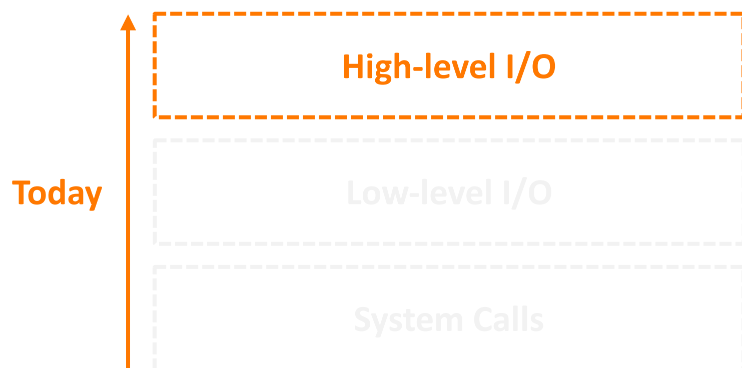
- Duplicating descriptors
  - int dup( int oldfd )
  - int dup2( int oldfd, int newfd )

# Offer standard interface: low-level I/O system calls

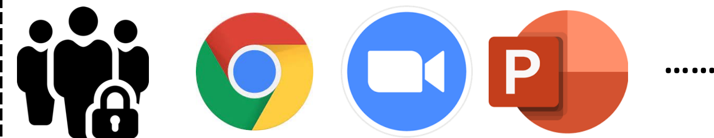
- Duplicating descriptors
  - int dup( int oldfd )
  - int dup2( int oldfd, int newfd )
- Modify configurations of a device file
  - int ioctl(int fd, unsigned long request, …)
- Inter-process communication
  - int pipe( int pipefd[2], … )
  - ex. Process A write to *pipefd[1]* and Process B reads from *pipefd[0]*
- …

# OFFER STANDARD INTERFACE: HIGH-LEVEL I/O

- I/O
  - **Definition :** input and output
  - **Def (*NIX):** any operation that read/write system services (*NIX OS: everything is a file)




Users Run Applications




Standard Interfaces (Libraries)

File System(s)

I/O Drivers

Hardware (CPU, GPU, Mem, …)

# Offer standard interface: high-level I/O

- File as a stream
  - **Definition:** an unformatted sequence of bytes **with a position**
  - **Functions :**
    - FILE *fopen( const char *filename, const char *mode )
    - int fclose( FILE *fp )
  - **Details :**
    - fopen() returns a stream represented by **a pointer** to a FILE data structure
    - Returns NULL if we have an error

# Offer standard interface: high-level I/O

- File as a stream
  - **Definition:** an unformatted sequence of bytes **with a position**
  - **Functions :**
    - FILE *fopen( const char *filename, const char *mode )
    - int fclose( FILE *fp )
  - **Details :**
    - fopen() returns a stream represented by **a pointer** to a FILE data structure
    - Returns NULL if we have an

| Mode | Descriptions |
|---|---|
| r | Open existing file for reading |
| w | Open for writing; create if not exists |
| a | Open for appending; create if not exists |
| r+ | Open existing file for reading and writing |
| w+ | Open for reading and writing; empty a file if exists |
| a+ | Open for reading and writing; read from the beginning and write as append |

# Offer standard interface: high-level I/O

- File as a stream
  - **Definition:** an unformatted sequence of bytes **with a position**
  - **Functions :**
    - FILE *fopen( const char *filename, const char *mode )
    - int fclose( FILE *fp )
  - **Standard streams:**
    - FILE *stdin   : normal source of input, can be redirected
    - FILE *stdout: normal source of output; redirection can be done
    - FILE *stderr : output errors

# Offer standard interface: high-level I/O

- File as a stream
  - **Definition:** an unformatted sequence of bytes **with a position**
  - **Functions :**
    - FILE *fopen( const char *filename, const char *mode )
    - int fclose( FILE *fp )

  - **Standard streams:**
    - FILE *stdin : normal source of input, can be redirected
    - FILE *stdout: normal source of output; redirection can be done
    - FILE *stderr : output errors

  - **Standard streams in Terminal:**
    - Each stream has numbers: 0 (stdin), 1 (stdout), 2 (stderr)
    - An example command : $ ./movie movie.csv > ./output 2>&1 &

# Offer standard interface: high-level I/O

- File as a stream
  - **Definition:** an unformatted sequence of bytes **with a position**
  - **Functions :**
    - FILE *fopen( const char *filename, const char *mode )
    - int fclose( FILE *fp )
  - **Standard streams:**
    - FILE *stdin : normal source of input, can be
    - FILE *stdout: normal source of output; redirection can be done
    - FILE *stderr : output errors
  - **Standard streams in Terminal:**
    - Each stream has numbers: 0 (stdin), 1 (stdout), 2 (stderr)
    - An example command : $ ./movie movie.csv > ./output 2>&1

Redirects the stdout from "./movie movie.csv" to "./output" file. "printf" outputs will be stored.

Errors won't be stored to "./output" "2>&1" redirects stderr output to stdin; stored to the file

# Offer standard interface: read/write from/to a stream

- Character(byte)-level API
  - int fputc( int c, FILE *fp )
  - int fputs( const char *s, FILE *fp )
  - int fgetc( FILE *fp )
  - char *fgets( char *buf, int n, FILE *fp )

# Offer standard interface: read/write from/to a stream

- Character(byte)-level API
  - int fputc( int c, FILE *fp )
  - int fputs( const char *s, FILE *fp )
  - int fgetc( FILE *fp )
  - char *fgets( char *buf, int n, FILE *fp )

- Block-level API
  - size_t fread( void *ptr, size_t size_of_elements, size_t number_of_elements, FILE *fp )
  - size_t fwrite( void *ptr, size_t size_of_elements, size_t number_of_elements, FILE *fp )

# Offer standard interface: read/write from/to a stream

- Character(byte)-level API
  - int fputc( int c, FILE *fp )
  - int fputs( const char *s, FILE *fp )
  - int fgetc( FILE *fp )
  - char *fgets( char *buf, int n, FILE *fp )

- Block-level API
  - size_t fread( void *ptr, size_t size_of_elements, size_t number_of_elements, FILE *fp )
  - size_t fwrite( void *ptr, size_t size_of_elements, size_t number_of_elements, FILE *fp )

- (More convenient) API allows formatting
  - int fprintf( FILE *restrict stream, const char *restrict format, … );
  - int fscanf( FILE *restrict stream, const char *restrict format, … );

# Offer standard interface: high-level I/O

- Example C code:

```
#define BUFFER_SIZE     256

int main(void) {
   FILE *input;
   char *buffer = (char *) calloc(BUFFER_SIZE * sizeof(char));
   size_t len = 0;

   input = fopen("input.txt", "r");
   if (input == NULL) {
      printf("Cannot open the input.txt file, abort.\n");
      return -ENOENT;
   }

   len = fread(buffer, BUFFER_SIZE, sizeof(char), input);
   while (len > 0) {
      printf("[CHAR] read: %c\n", buffer[--len]);
   }

   fclose(input);
   return 0;
}
```

**Macros (some predefineds):**
You can define any numbers, strings, etc.
Or you can use what C already defines

**fopen / fread system calls:**
Open a file and read the contents, 256 bytes,
The file will be open for reading-only. If the
contents are less than 256 bytes. It will return all

# Offer standard interface: high-level I/O

- Example C code:

```
#define BUFFER_SIZE     256

int main(void) {
   FILE *input;
   char *buffer = (char *) calloc(BUFFER_SIZE * sizeof(char));
   size_t len = 0;

   input = fopen("input.txt", "r");
   if (input == NULL) {
      printf("Cannot open the input.txt file, abort.\n");
      return -ENOENT;
   }

   len = fread(buffer, BUFFER_SIZE, sizeof(char), input);
   while (len > 0) {
      printf("[CHAR] read: %c\n", buffer[--len]);
   }

   fclose(input);
   return 0;
}
```

The next fread/fwrite will be performed from the new location!

# Offer standard interface: high-level I/O

- Example C code:

```
#define BUFFER_SIZE     256

int main(void) {
   FILE *input;
   char *buffer = (char *) calloc(BUFFER_SIZE * sizeof(char));
   size_t len = 0;

   input = fopen("input.txt", "r");
   if (input == NULL) {
      printf("Cannot open the input.txt file, abort.\n");
      return -ENOENT;
   }

   len = fread(buffer, BUFFER_SIZE, sizeof(char), input);
   while (len > 0) {
      printf("[CHAR] read: %c\n", buffer[--len]);
   }

   fclose(input);
   return 0;
}
```

**Good system programming practice**

Make your program returns proper errors in any cases; the error numbers are in here

# Offer standard interface: some additional APIs

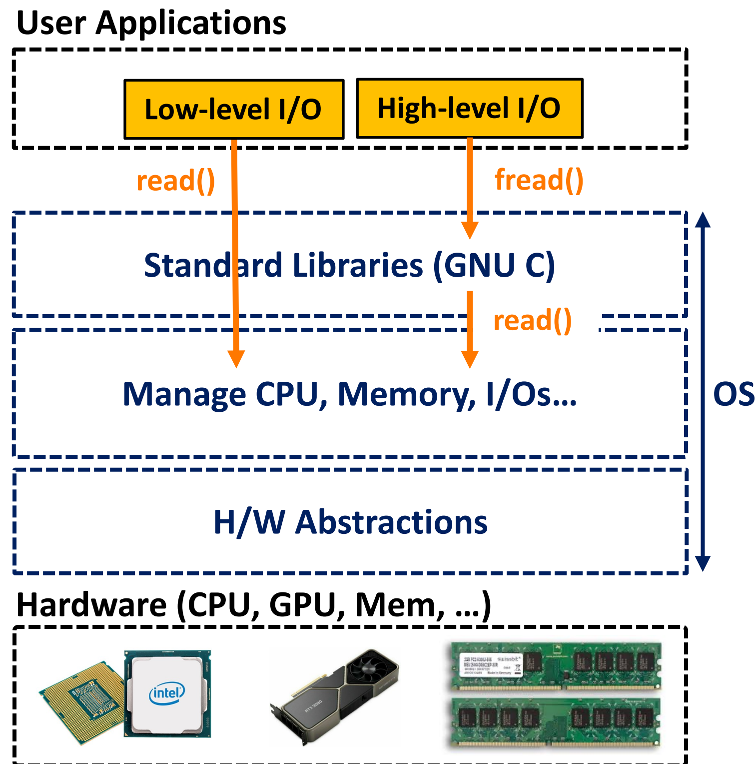
- Current working directory (CWD)
  - Each process has CWD (in their process context, i.e., *task_struct*)
  - int chdir( const char *path );
    - Set the CWD to path
    - Returns zero upon success; otherwise, returns -1

# Topics for today

- Part II: I/Os
  - Provide abstractions
    - What is I/O?
  - Offer standard interface
    - How can we do low-level I/Os?
    - How can we do high-level I/Os?
  - Manage resources
    - How OS manages (file) I/O internally?

# Manage resources: high-level vs. low-level I/Os

- Low-level I/O uses system calls, while high-level I/Os are **not**
  - **System calls**
    - They directly request OS services/resources
    - e.g., open(), read(), write(), and close()
  - **Standard libraries in C**
    - They are offered by C libraries
    - C libraries eventually do system calls
    - e.g., fopen(), fread(), fwrite(), and fclose()



# Manage resources: high-level vs. low-level I/Os

## High-level I/O calls

```
size_t fread(…) {
   You can do something at here!

   asm code … syscall <number> into %eax
   put <syscall args> into registers %ebx
   special trap instruction

   Kernel:
      get <syscal args> from %ebx
      dispatch to system func
      do the work to read from the file
      store return value in %eax

   get return values from regs

   You can do something at here!
}
```

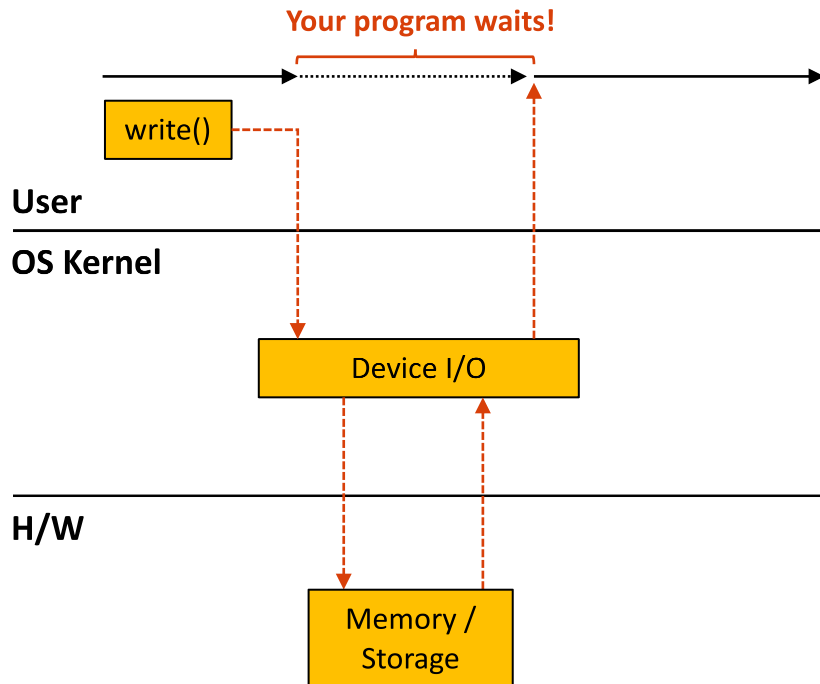
## Low-level I/O calls

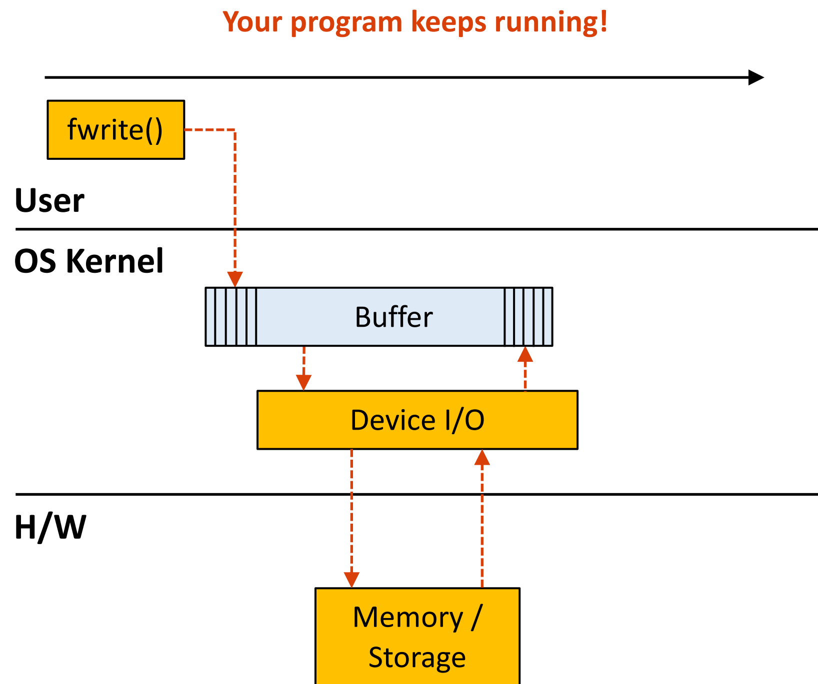
```
ssize_t read(…) {

   asm code … syscall <number> into %eax
   put <syscall args> into registers %ebx
   special trap instruction

   Kernel:
      get <syscal args> from %ebx
      dispatch to system func
      do the work to read from the file
      store return value in %eax

   get return values from regs

}
```

**High-level I/O calls also use system calls!**

# Manage resources: an example of "something"

- Kernel buffering



# Topics for today

- Part II: I/Os
  - Provide abstractions
    - What is I/O?
  - Offer standard interface
    - What OS provide us to do raw I/Os?
    - What OS provide us to do high-level I/Os?
  - Manage resources
    - How OS manages (file) I/O internally?

# Thank You!

M/W 12:00 – 1:50 PM (LINC #200)

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL

Secure AI Systems Lab