NOTICE

- Deadlines
  - (4/27 11:59 PM) Programming assignment 2

CS 374: Operating Systems I

Part II – I/O

M/W 12:00 – 1:50 PM (LINC #200)

Sanghyun Hong

sanghyun.hong@oregonstate.edu

RECAP: FILESYSTEM STRUCTURE OVERVIEW

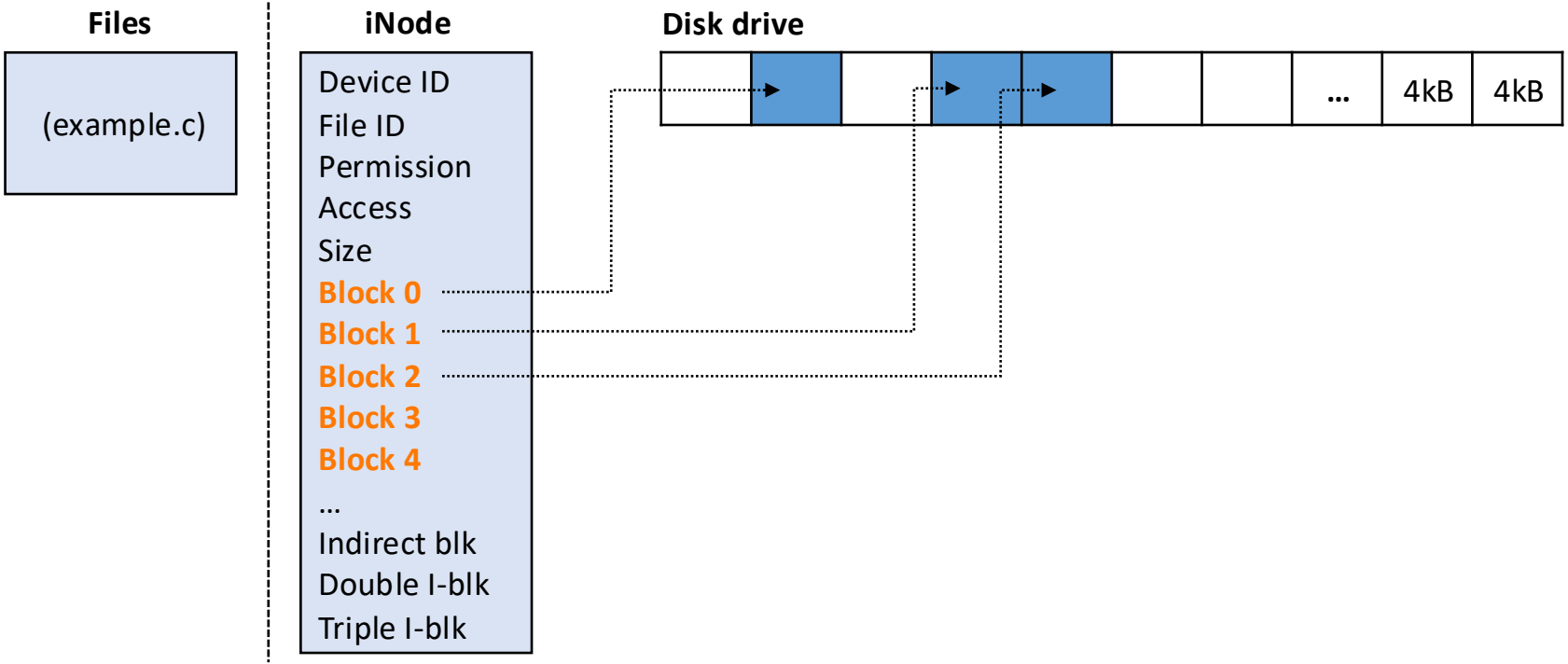
- Basic components
  - File : a named collection of data
  - Directory: a file that holds other files as data

- 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)

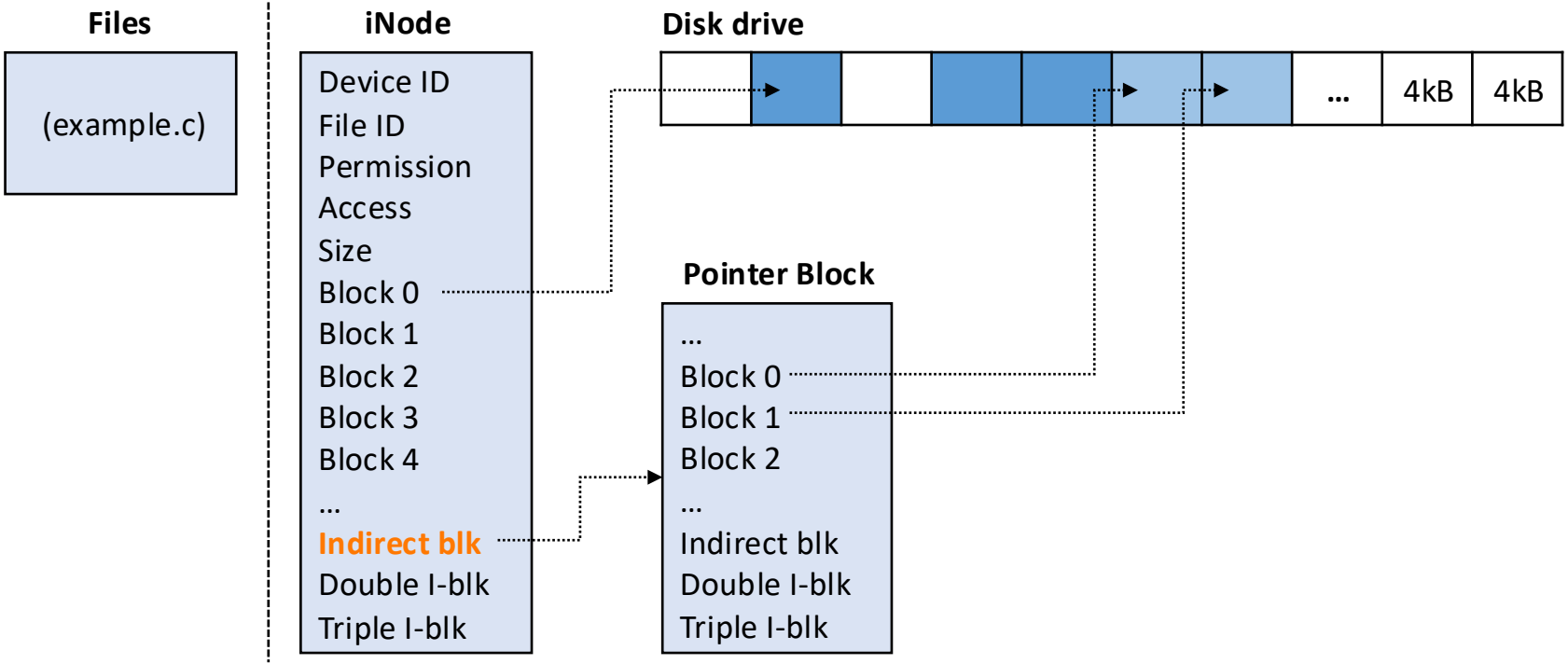
FILESYSTEM STRUCTURE (EXT2/3)

- An iNode can hold 12 blocks ≈ 48kB



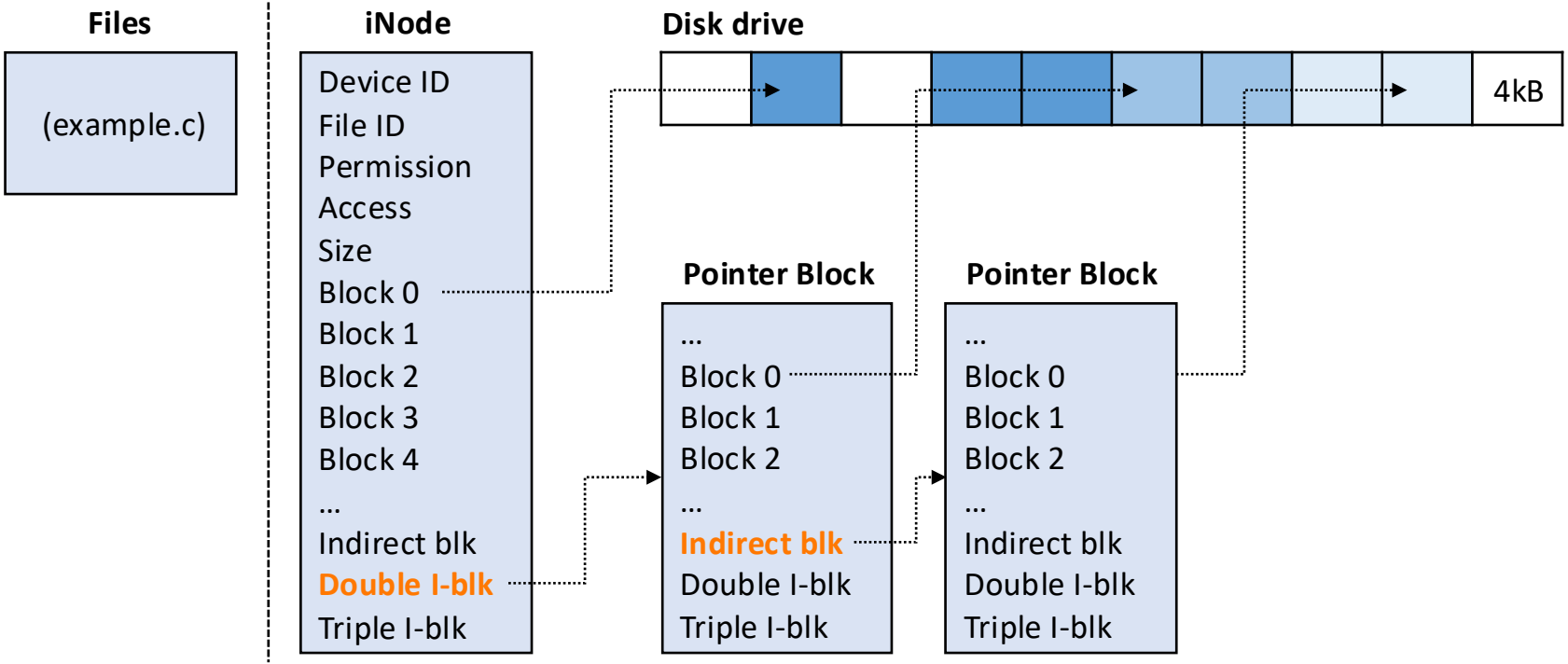
FILESYSTEM STRUCTURE: HOW TO STORE A LARGE FILE

- To store a large file, iNode supports "indirect block" ≈ 4MB (1024 pointers) + 48k



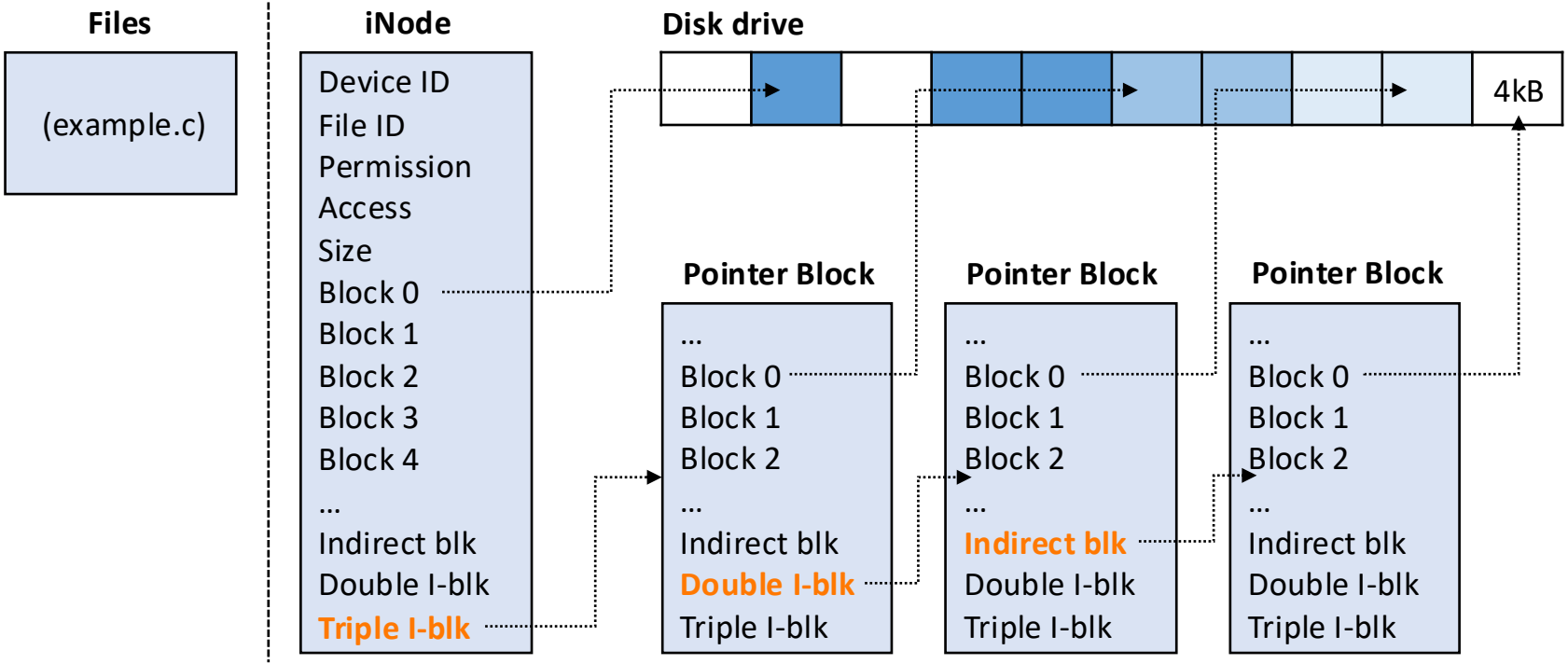
FILESYSTEM STRUCTURE: HOW TO STORE LARGER FILES

- To store a large file, iNode supports "double I-blk" ≈ 4GB (1024^2 pointers) + 4MB + 48kB



FILESYSTEM STRUCTURE: HOW TO STORE LARGER FILES

- Triple indirect blk ≈ 4TB (1024^3 pointers) + 4GB + 4MB + 48kB



FILESYSTEM STRUCTURE OVERVIEW – CONT'D

- Design choices
  - FAT:
    - Index: Linked lists (FAT Table)
    - Data : Cluster
  - ext2/3/4
    - Index: iNode (indirect block)
    - Data : Block
  - NTFS:
    - Index: MFT entry (B-tree)
    - Data : Extent (Data run)

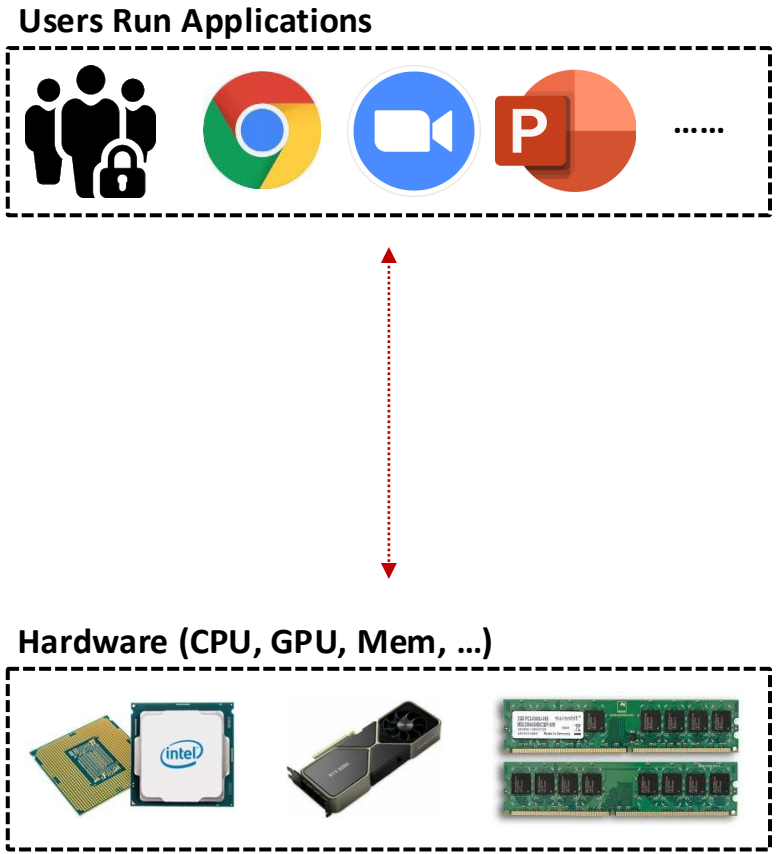
TOPICS FOR TODAY

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

PROVIDE ABSTRACTION

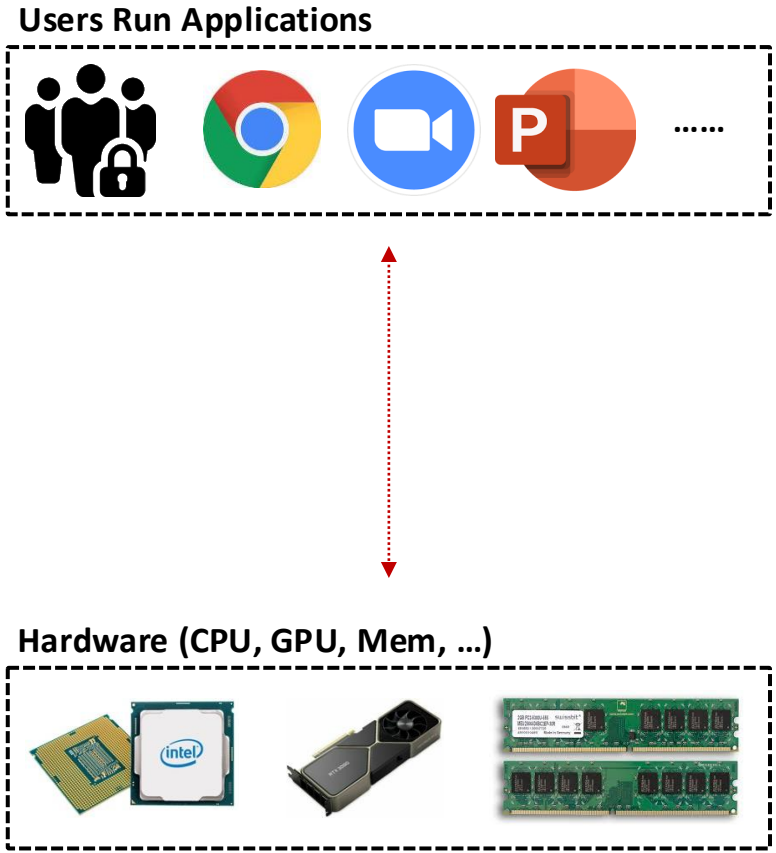
- H/W devices are diverse and complex
  - Disk, keyboard, USB, NIC, …
  - Each has different interfaces and protocols

- Imagine the world without abstraction
  - Directly control hardware registers
  - Be rewritten every time hardware changes
  - Know device-specific details (e.g., controller)
  - …



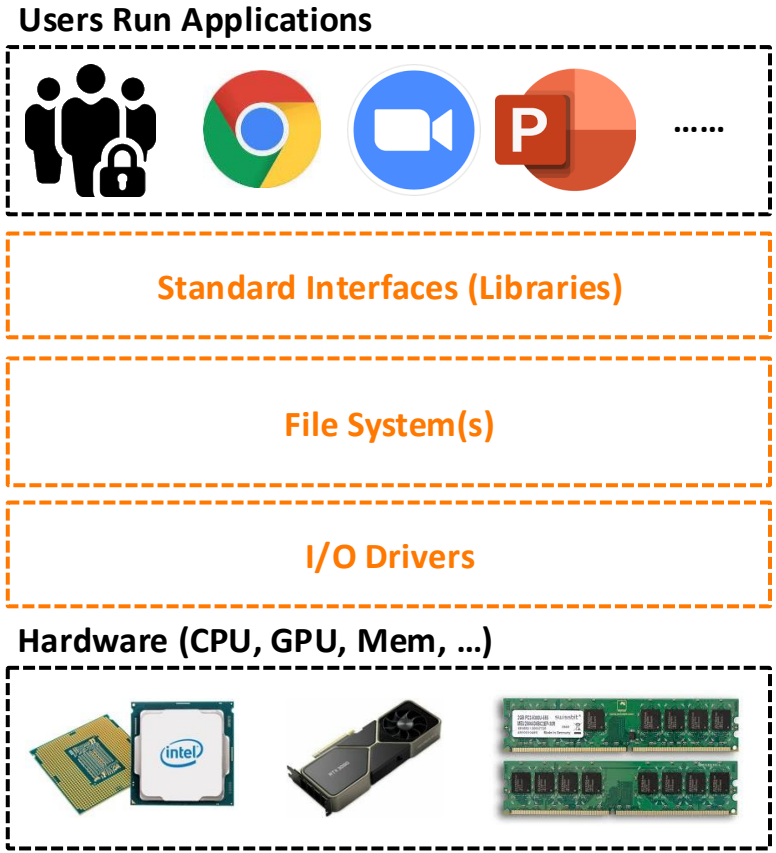
PROVIDE ABSTRACTION

- Without OS
  - No portability
  - No security (any app can access any H/W)
  - No sharing between processes



PROVIDE ABSTRACTION

- OS as a layer
  - **Hides** H/W complexity from apps
  - **Exposes** a clean, unified interface
  - **Controls** who can access what (security)



PROVIDE ABSTRACTION

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

  - Examples between resources and *NIX abstractions
    - Disk file — /home/user/file.txt
    - Terminal — /dev/tty
    - Keyboard — /dev/stdin
    - USB device — /dev/sdb
    - Network socket — socket fd
    - Pipe — pipe fd
    - …

PROVIDE ABSTRACTION

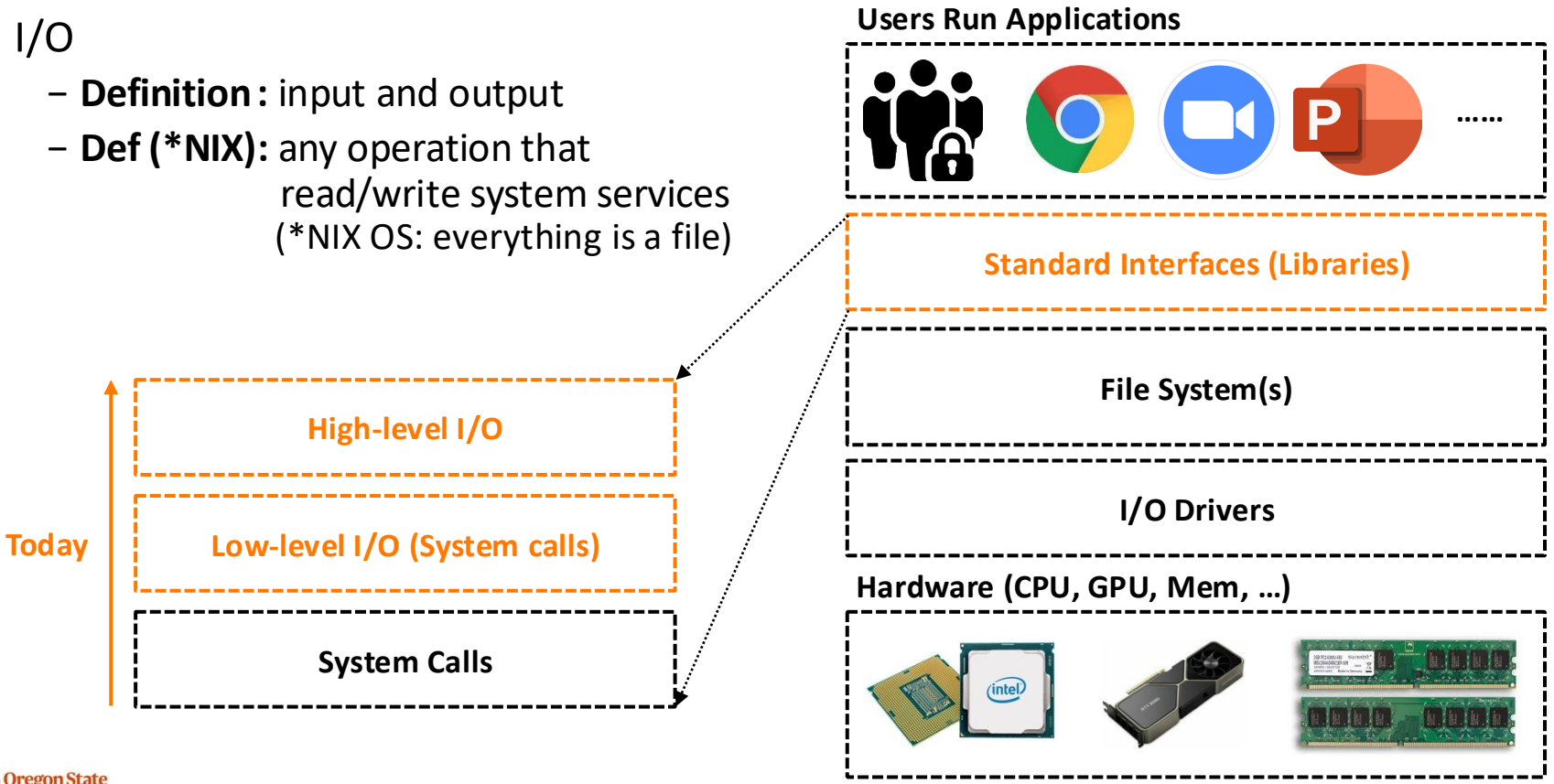
- Power of *NIX philosophy
  - All I/Os share the same interface
    - read(fd, buffer, size);
    - write(fd, buffer, size);
  - Benefits
    - Apps does *not* need to understand what's behind "fd"
    - H/W changes are *invisible* to applications

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?

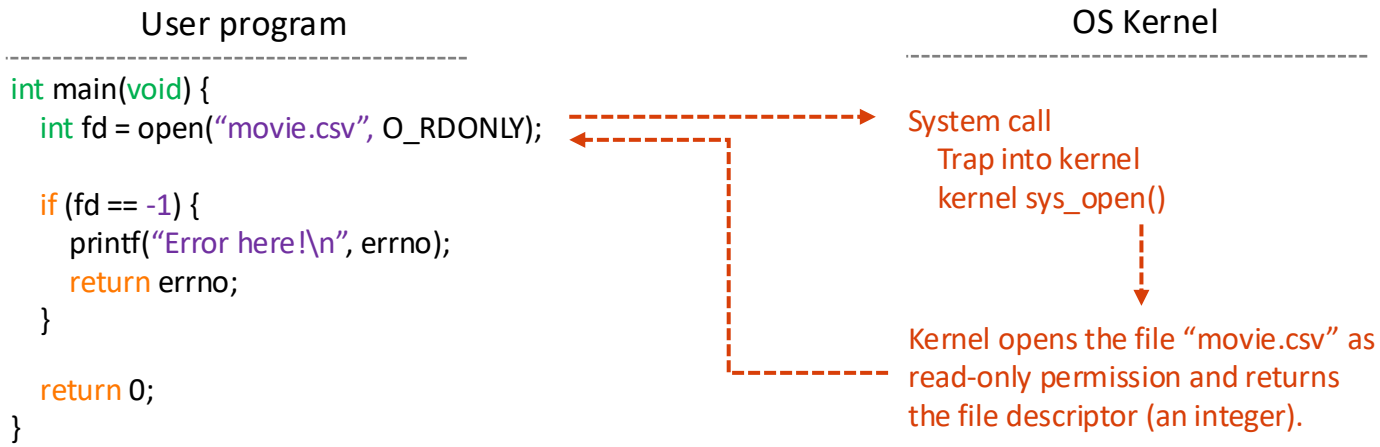
OFFER STANDARD INTERFACE

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



RECAP: SYSTEM CALL

- System call
  - **Definition:** a user-level function call to request a service from the OS
    - Enter into kernel mode from user mode via trap instruction
    - Arguments passed via registers (%eax, %ebx, …)
    - Kernel chooses the right function by syscall number
  - Low-level I/O (open, read, write) are system calls



OFFER STANDARD INTERFACE: OVERVIEW – LOW-LEVEL I/O

- Low-level I/O
  - **POSIX API** that wraps system calls
    - Simpler interface than raw system calls (e.g., open() instead of syscall)
    - Returns human-readable errors via errno
      - Note that raw system call returns a number
      - open() translates it to errno (e.g., ENOENT, EACCESS)

OFFER STANDARD INTERFACE: LOW-LEVEL I/O

- Key abstraction: file descriptors (fd)
  - **Definition:** an integer that uniquely identifies an open file in Linux
  - **POSIX API:** (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** (always open)**:**
    - 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
  - **POSIX API:**
    - int open(const char *filename, int flags, mode_t *mode)
      - Open the file and return a new fd
      - Returns error (link), e.g., -1, upon failure
      - **flags :** access mode (O_RDONLY, O_APPEND, …)
      - **mode:** access permission (S_IRUSR, S_IRWXU, ...)
    - int create(const char *filename, mode_t *mode)
      - Equivalent to open() with O_CREAT | O_WRONLY | O_TRUNC
    - int close(int *fd)
      - Releases the fd back to the OS
      - Must close fds opened – fd leak vulnerabilities

OFFER STANDARD INTERFACE: LOW-LEVEL I/O – READ AND WRITE

- Low-level read/write 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**
      - Negative values are errors

OFFER STANDARD INTERFACE: LOW-LEVEL I/O – READ AND WRITE

- Low-level read/write 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 written
    - Your program will wait until write completes (no buffering)

OFFER STANDARD INTERFACE: LOW-LEVEL I/O – READ AND WRITE

- Low-level read/write 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 (no buffering)
  - 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 – READ AND WRITE

- Low-level read/write 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

- Example C code:

```c
#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 – OTHERS

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

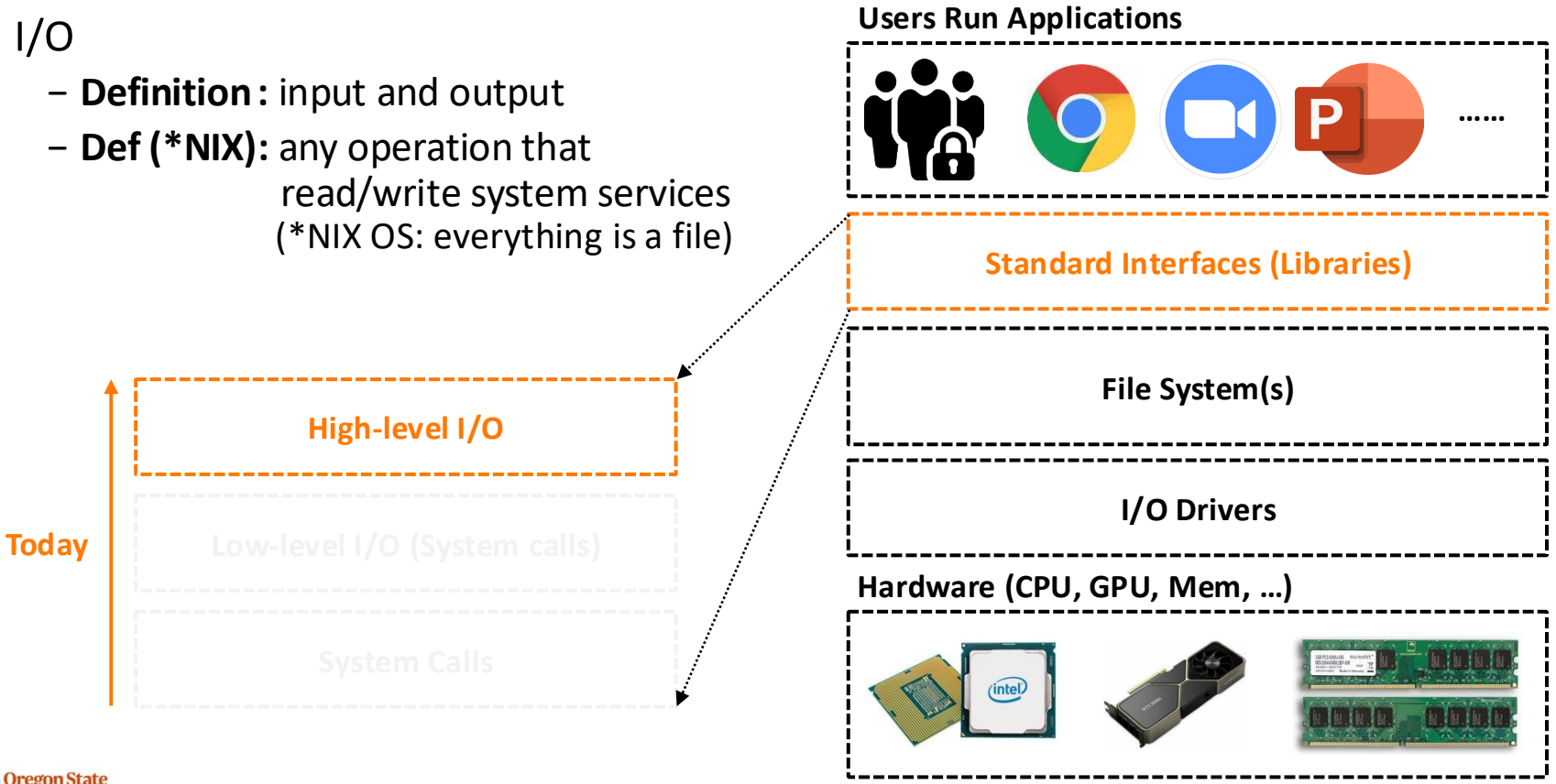
```
int main(void) {
   // Step 1: backup stdout (fd=1) before replacing it
   int stdout_backup = dup(1);
   // stdout_backup = 4 (OS automatically assigns lowest available fd)
   // fd=1 → terminal | fd=4 → terminal (same open file table entry)

   // Step 2: replace stdout with a file
   int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
   dup2(fd, 1);
   close(fd);
   // fd=1 → output.txt | fd=4 → terminal (backup still alive)

   // Step 3: printf now goes to output.txt
   printf("This goes to output.txt\n");

   // Step 4: restore stdout using the backup
   dup2(stdout_backup, 1);
   close(stdout_backup);
   // fd=1 → terminal (restored!)

   // Step 5: printf goes back to terminal
   printf("This goes to terminal again\n");

   return 0;
}
```

OFFER STANDARD INTERFACE: LOW-LEVEL I/O – SOME MORE

- 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], …)
  - pipefd[0]: read end
  - pipefd[1]: write end
  - ex. Process A write to *pipefd[1]* and Process B reads from *pipefd[0]*
  - ex. ls | grep <token>
- …

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)



OFFER STANDARD INTERFACE: HIGH-LEVEL I/O

- Key abstraction: 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
    - FILE wraps a fd with extra information (e.g., buffer, position, error flags)
    - Returns NULL if we have an error

OFFER STANDARD INTERFACE: HIGH-LEVEL I/O

- Key abstraction: 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
    - FILE wraps a fd with extra i
    - 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

- Key abstraction: 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

- Key abstraction: 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

- Key abstraction: 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 b
    - 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) formatted APIs
  - 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:

```c
#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;
}
```

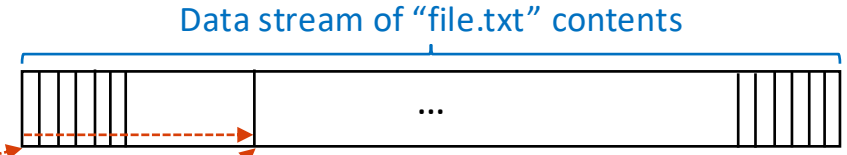
Macros (some predefineds):
You can define any numbers, strings, etc.
Or you can use what C already defines

fopen / fread:
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:

```c
#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:

```c
#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: HIGH-LEVEL I/O – SOME MORE

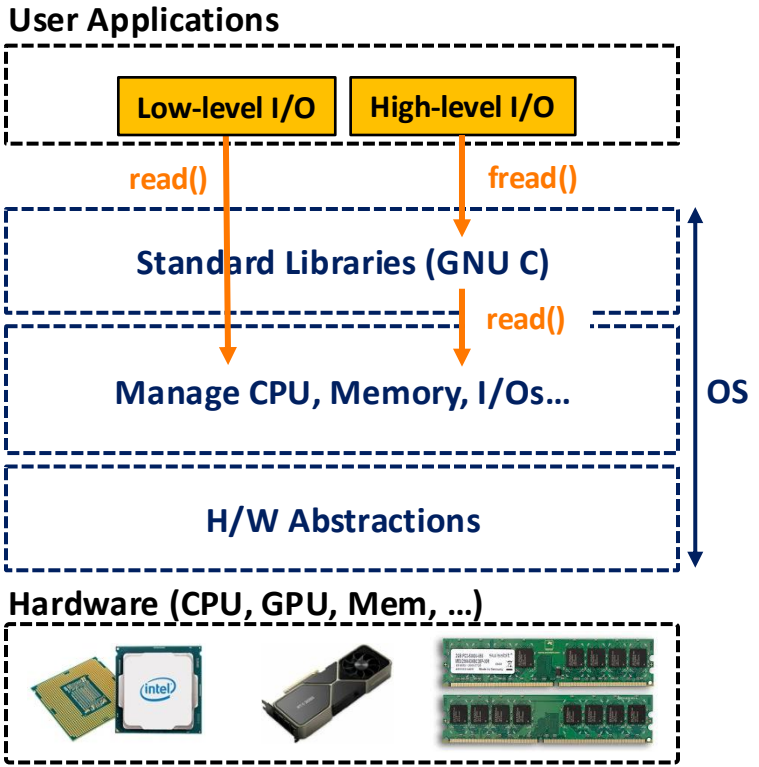
- Current working directory (CWD)
  - Each process has CWD (in their process context, i.e., *task_struct*)
  - int chdir(const char *path);
    - Set the CWD to path
    - Returns zero upon success; otherwise, returns -1

TOPICS FOR TODAY

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

MANAGE RESOURCES: HIGH-LEVEL VS. LOW-LEVEL I/O

- 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!
}
```

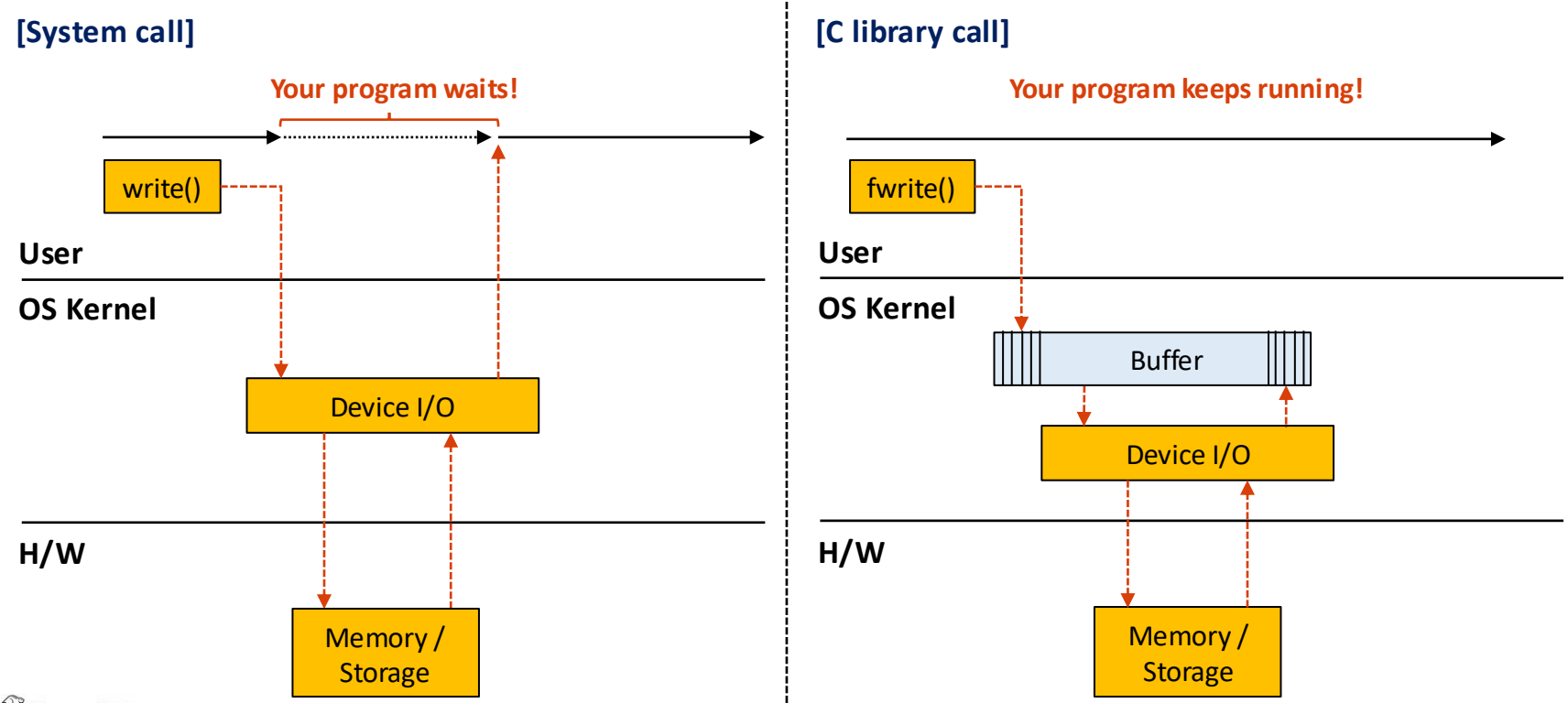
Low-level I/O calls

```
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/O?
    - What OS provide us to do high-level I/O?
  - Manage resources
    - How does OS manage (file) I/O internally?

Thank You!

M/W 12:00 – 1:50 PM (LINC #200)

Sanghyun Hong

sanghyun.hong@oregonstate.edu