

# CS 344: OPERATING SYSTEMS I

## 02.06: FILESYSTEM INTERNALS

M/W 12:00 – 1:50 PM (LINC #200)

Sanghyun Hong

[sanghyun.hong@oregonstate.edu](mailto:sanghyun.hong@oregonstate.edu)



**Oregon State**  
University

**SAIL**  
Secure AI Systems Lab

# NOTICE

---

- Announcements
  - No class on the 8<sup>th</sup>
  - No Sanghyun's office hours on the 10<sup>th</sup>

# NOTICE

---

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

# RECAP

---

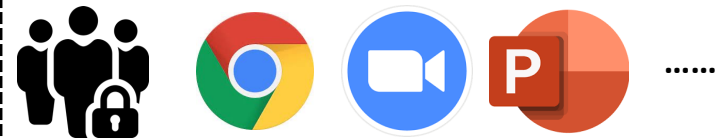
- Part II: I/Os
  - Provide abstractions
    - What is I/O?
  - Offer standard interface
    - How can we do low-level I/Os?
    - How can we do high-level I/Os?
  - Manage resources
    - How OS manages (file) I/O internally?

# OFFER STANDARD INTERFACE: 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 )
      - fopen() returns a stream represented by **a pointer** to a **FILE data structure**
      - Returns **NULL** if we have an error
    - **int** fclose( FILE \*fp )

# 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 )
      - fopen() returns a stream represented by **a pointer** to a **FILE data structure**
      - Returns **NULL** if we have an error
    - **int** fclose( FILE \*fp )



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 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`

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 predefined):

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
```

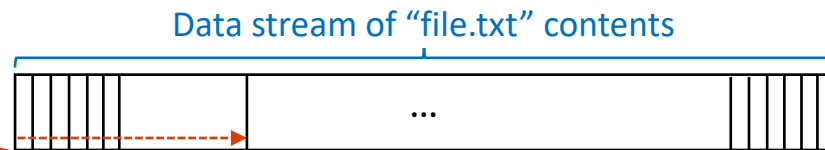
```
int main(void) {  
    FILE *input;  
    char *buffer = (char *) calloc(BUFFER_SIZE * sizeof(char));  
    size_t len = 0;
```

```
    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

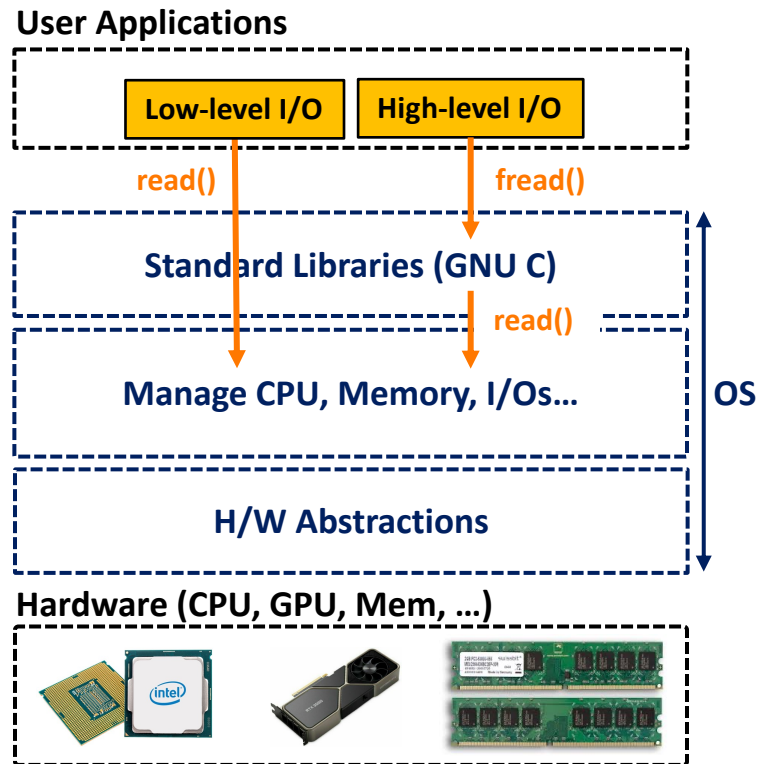
# TOPICS FOR TODAY

---

- 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 <syscall 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 <syscall 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!**

# HIGH-LEVEL API INTERNALS: WHY?

---

- Given the functionalities we've learned:
  - `fopen()` anyway uses `open()` system call
  - `fopen()` may make users (or developers) more confusing which one to use (`open?`, `fopen?`)
- **Problem**
  - System calls are **25x slower** than the standard function call
  - Solutions?
- **Kernel buffering**
  - Create a buffer (in user-space or kernel-space?)
  - Read/write data asynchronously
    - Read whatever amount of data in the buffer
    - Write the data to devices when the buffer is full

## Recall the `read()` system call:

```
ssize_t read(...) {  
    asm code ... syscall <number> into %eax  
    put <syscall args> into registers %ebx  
    special trap instruction
```

## Kernel:

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

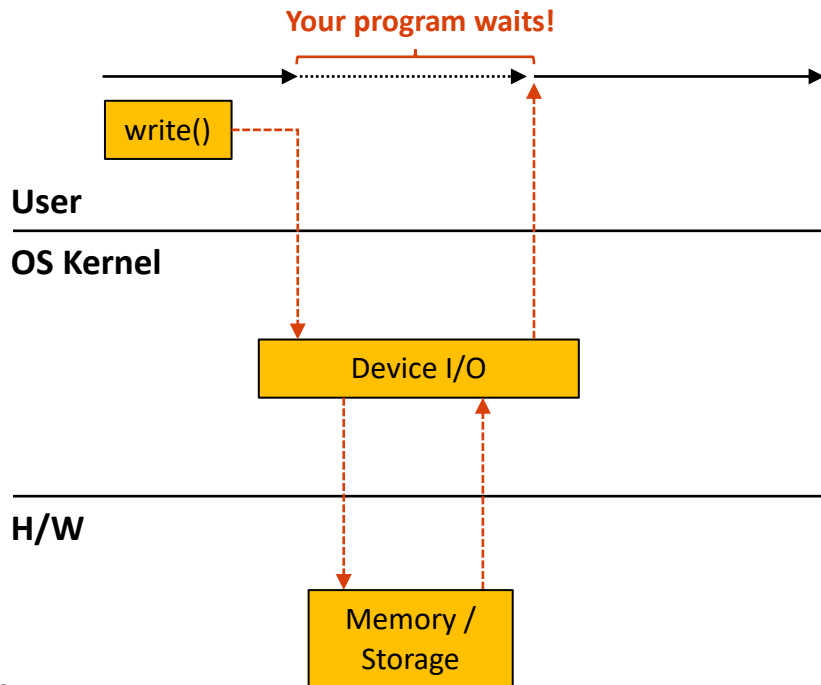
```
    get return values from regs
```

```
}
```

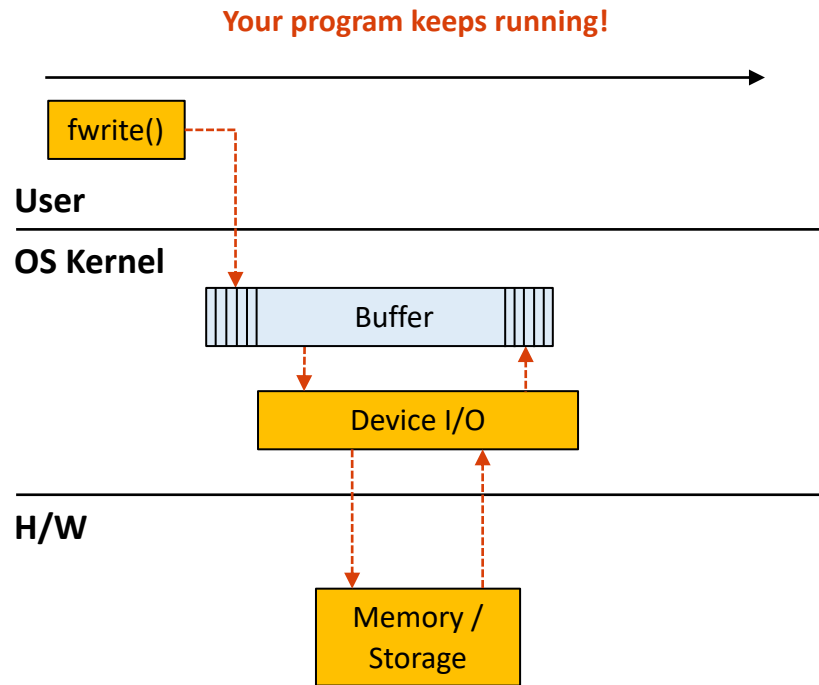
# HIGH-LEVEL API INTERNALS: AN EXAMPLE OF “SOMETHING”

- Kernel buffering

[System call]



[C library call]



# HIGH-LEVEL API INTERNALS: AN EXAMPLE OF “SOMETHING”

---

- When fwrite flushes the buffer?
  - When we write data to the buffer, but it is full
  - When we close the stream, *i.e.*, fclose(FILE \*fp)
  - When the program that has called fwrite() finished its execution (*i.e.*, terminated)
  - When a new line (*i.e.*, \n) is written to the buffer
  - When a program reads data from a file (not from the buffer)
  - ...
- Or if you explicitly call fflush()
  - `int fflush( FILE *fp );`

# TOPICS FOR TODAY

---

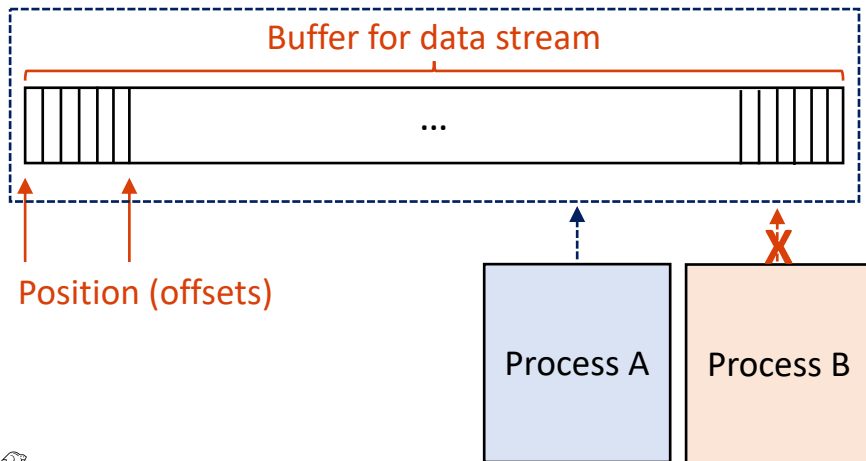
- Part II: Filesystem internals
  - Manage resources
    - How OS manages high-level I/O internally?
    - How OS manages low-level I/O internally?



# HIGH-LEVEL API INTERNALS: HOW?

- FILE data structure contents ([code](#))
  - File descriptor (**fd**), from `open()`
  - **Offsets**, a position to read/write data
  - **Buffer** (an array of bytes to read/write data)
  - **Lock** (only one process can access data)

- FILE (Diagram)



```
49 struct _IO_FILE
50 {
51     int _flags;          /* High-order word is _IO_MAGIC; rest is flags. */
52
53     /* The following pointers correspond to the C++ streambuf protocol. */
54     char *_IO_read_ptr; /* Current read pointer */
55     char *_IO_read_end; /* End of get area. */
56     char *_IO_read_base; /* Start of putback+get area. */
57     char *_IO_write_base; /* Start of put area. */
58     char *_IO_write_ptr; /* Current put pointer. */
59     char *_IO_write_end; /* End of put area. */
60     char *_IO_buf_base; /* Start of reserve area. */
61     char *_IO_buf_end; /* End of reserve area. */
62
63     /* The following fields are used to support backing up and undo. */
64     char *_IO_save_base; /* Pointer to start of non-current get area. */
65     char *_IO_backup_base; /* Pointer to first valid character of backup area */
66     char *_IO_save_end; /* Pointer to end of non-current get area. */
67
68     struct _IO_marker *_markers;
69
70     struct _IO_FILE *_chain;
71
72     int _fileno;
73     int _flags2;
74     __off_t _old_offset; /* This used to be _offset but it's too small. */
75
76     /* 1+column number of pbase(); 0 is unknown. */
77     unsigned short _cur_column;
78     signed char _vtable_offset;
79     char _shortbuf[1];
80
81     _IO_lock_t *_lock;
82
83     __off64_t _offset;
84
85     /* Wide character stream stuff. */
86     struct _IO_codecvt *_codecvt;
87     struct _IO_wide_data *_wide_data;
88     struct _IO_FILE *_freeres_list;
89     void *_freeres_buf;
90     size_t __pad5;
91     int _mode;
92
93     /* Make sure we don't get into trouble again. */
94     char _unused2[15 * sizeof (int) - 4 * sizeof (void *) - sizeof (size_t)];
95 };
```

# HIGH-LEVEL API INTERNALS: AN EXAMPLE OF “SOMETHING”

---

- When fwrite flushes the buffer?
  - When we write data to the buffer, but it is full
  - When we close the stream, *i.e.*, fclose(FILE \*fp)
  - When the program that has called fwrite() finished its execution (*i.e.*, terminated)
  - When a new line (*i.e.*, \n) is written to the buffer
  - When a program reads data from a file (not from the buffer)
  - ...
- Or if you explicitly call fflush()
  - `int fflush( FILE *fp );`

# HIGH-LEVEL API INTERNALS: HOW?

- Exercise

... many #include ...

```
int main(void) {  
    char name[8] = "Sanghyun";  
    char desc[28] = "is an instructor of CS 344\n";  
  
    fwrite(name, sizeof(char), strlen(name), stdout);  
    sleep(10);  
    fwrite(desc, sizeof(char), strlen(desc), stdout);  
    return 0;  
}
```

... many #include ...

```
int main(void) {  
    char name[8] = "Sanghyun";  
    char desc[28] = "is an instructor of CS 344\n";  
  
    write(STDOUT_FILENO, name, strlen(name));  
    sleep(10);  
    write(STDOUT_FILENO, desc, strlen(desc));  
    return 0;  
}
```

- Before the sleep(10), what message you'll see in your terminal?
- After the sleep(10), what message you'll see in your terminal?

# HIGH-LEVEL API INTERNALS: HOW?

---

- Exercise

```
... many #include ...
```

```
int main(void) {  
    char x = "S";  
  
    FILE *fp1 = fopen("input.txt", "w");  
    fwrite("H", sizeof(char), 1, fp1);  
  
    FILE *fp2 = fopen("input.txt", "r");  
    fread(&x, sizeof(char), 1, fp2);  
  
    printf("I read %c\n", x);  
    return 0;  
}
```

- Case I

- “H” is written to the file by fwrite()
- fread() will read “H” from the file
- Print “H”

- Case II

- “H” is in the kernel buffer
- fread() won’t read anything from the file
- Print “S”

# HIGH-LEVEL API INTERNALS: HOW?

- Exercise

```
... many #include ...
```

```
int main(void) {  
    char x = "S";  
  
    FILE *fp1 = fopen("input.txt", "w");  
    fwrite("H", sizeof(char), 1, fp1);  
    fflush(fp1);  
  
    FILE *fp2 = fopen("input.txt", "r");  
    fread(&x, sizeof(char), 1, fp2);  
  
    printf("I read %c\n", x);  
    return 0;  
}
```

- Case I

- “H” is written to the file by fwrite()
- fread() will read “H” from the file
- Print “H”

- Case II

- “H” is in the kernel buffer
- fread() won’t read anything from the file
- Print “S”

- Case with fflush()

- “H” is written to the buffer
- It will be flushed to the file by fflush()
- fread() will read “H” from the file
- Print “H”

# TOPICS FOR TODAY

---

- Part II: Filesystem internals
  - Manage resources
    - How OS manages high-level I/O internally?
    - How OS manages low-level I/O internally?

# LOW-LEVEL API INTERNALS

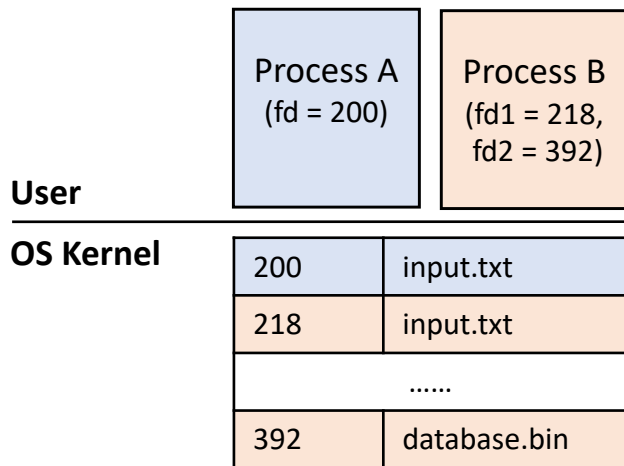
---

- File descriptors (fd)
  - **Definition** : an integer that uniquely identifies an open file in Linux
  - **System calls:** (fcntl.h)
    - `int open( const char *filename, int flags, mode_t *mode )`
  - Magic behind the open()
    - open() creates an *open file descriptor table* for each process (fd is here)
    - open() also creates an *entry* in system-wide table of open files (offset are here)
    - *open file description object* in the kernel represents an *instance of an actual open file*

# LOW-LEVEL API INTERNALS: HOW?

- File descriptor ([code](#)) in Linux kernel
  - **iNode**, a structure that holds data on disk
  - **Offsets**, a position to read/write data
  - **No buffer**

- File descriptor (Diagram)



```
932 struct file {
933     union {
934         struct llist_node    fu_llist;
935         struct rcu_head      fu_rcuhead;
936     } fu;
937     struct path              f_path;
938     struct inode             *f_inode; /* cached value */
939     const struct file_operations *f_op;
940
941     /*
942      * Protects f_ep, f_flags.
943      * Must not be taken from IRQ context.
944      */
945     spinlock_t              f_lock;
946     atomic_long_t           f_count;
947     unsigned int            f_flags;
948     fmode_t                 f_mode;
949     struct mutex             f_pos_lock;
950     loff_t                  f_pos;
951     struct fown_struct       f_owner;
952     const struct cred        *f_cred;
953     struct file_ra_state     f_ra;
954
955     u64                     f_version;
956 #ifdef CONFIG_SECURITY
957     void                    *f_security;
958 #endif
959     /* needed for tty driver, and maybe others */
960     void                    *private_data;
961
962 #ifdef CONFIG_EPOLL
963     /* Used by fs/eventpoll.c to link all the hooks to this file */
964     struct hlist_head       *f_ep;
965 #endif /* #ifdef CONFIG_EPOLL */
966     struct address_space     *f_mapping;
967     errseq_t                 f_wb_err;
968     errseq_t                 f_sb_err; /* for syncfs */
969 } __randomize_layout
970 __attribute__((aligned(4))); /* lest something weird decides that 2 is OK */
971
```



# LOW-LEVEL API INTERNALS: HOW?

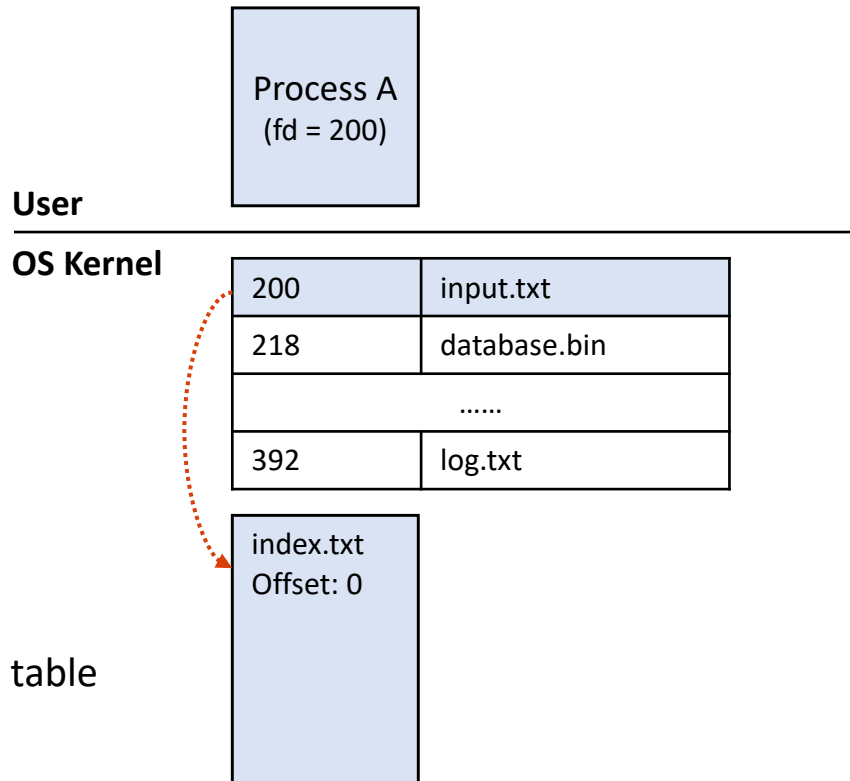
- Let's check with the following program

```
... many #include ...
#define BSIZE 100

int main(void) {
    char buf1[BSIZE];
    char buf2[BSIZE];
    int fd = open("input.txt", O_RDONLY); ←
    read(fd, buf1, BSIZE);
    read(fd, buf2, BSIZE);
    return 0;
}
```

- Note

- Process A opens a file “input.txt”
- OS Kernel opens the file, offset is 0
- OS Kernel create an entry to the descriptor table
- OS Kernel returns fd = 200



# LOW-LEVEL API INTERNALS: HOW?

- Let's check with the following program

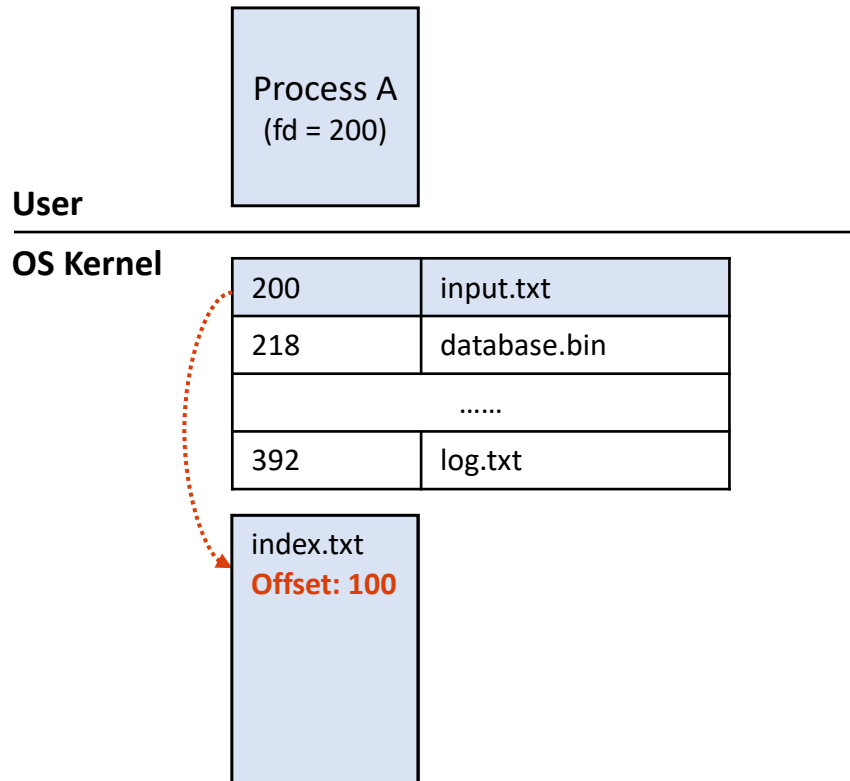
```
... many #include ...
#define BSIZE 100

int main(void) {
    char buf1[BSIZE];
    char buf2[BSIZE];
    int fd = open("input.txt", O_RDONLY);

    read(fd, buf1, BSIZE);
    read(fd, buf2, BSIZE);
    return 0;
}
```

- Note

- Process A read the file
- OS Kernel reads the file, 100 bytes
- OS Kernel moves the offset to 100
- OS Kernel returns the data to Process A



# LOW-LEVEL API INTERNALS: HOW?

- Let's check with the following program

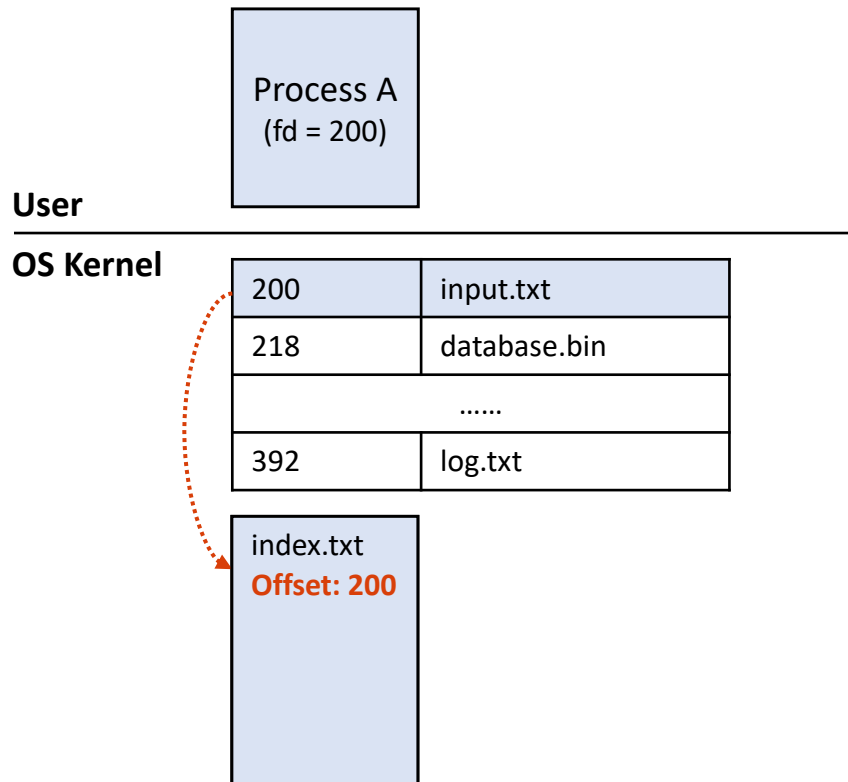
```
... many #include ...
#define BSIZE 100

int main(void) {
    char buf1[BSIZE];
    char buf2[BSIZE];
    int fd = open("input.txt", O_RDONLY);

    read(fd, buf1, BSIZE);
    read(fd, buf2, BSIZE);
    return 0;
}
```

- Note

- Process A read the file
- OS Kernel reads the file, 100 bytes
- OS Kernel moves the offset to 100
- OS Kernel returns the data to Process A



# LOW-LEVEL API INTERNALS: HOW?

- Let's do more exercise

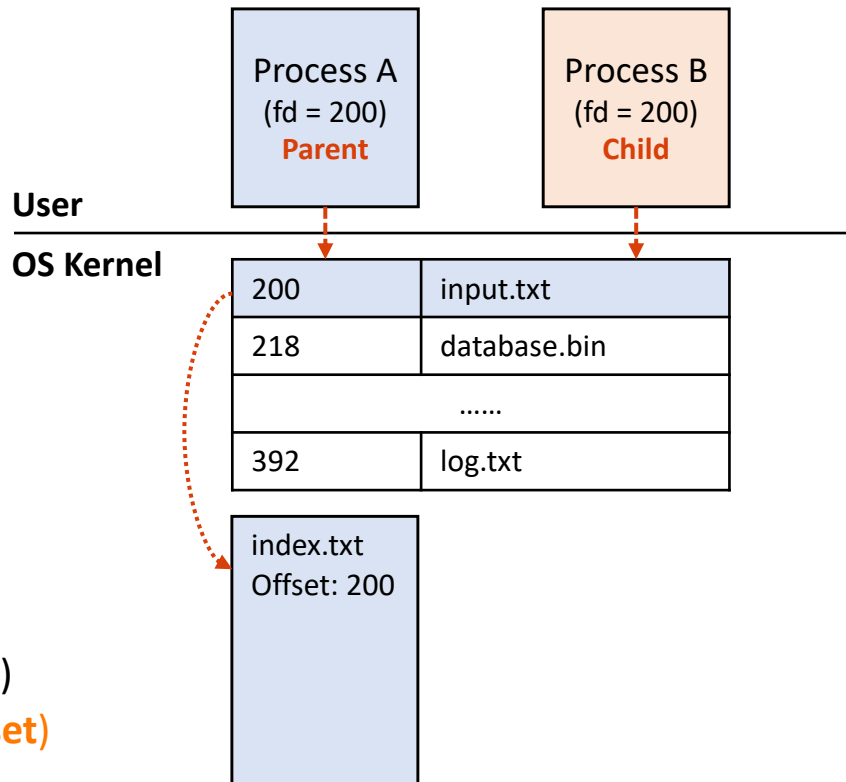
```
... many #include ...
#define BSIZE 100

int main(void) {
    char buf1[BSIZE];
    char buf2[BSIZE];
    int fd = open("input.txt", O_RDONLY), pid;

    read(fd, buf1, BSIZE);
    read(fd, buf2, BSIZE);

    switch (pid = fork()) {
```

- Note
  - Process A fork()!
  - Process B is created (a child)
  - Process B has the same file descriptor (200)
  - The fd is **copied** and **aliased** (share the **offset**)



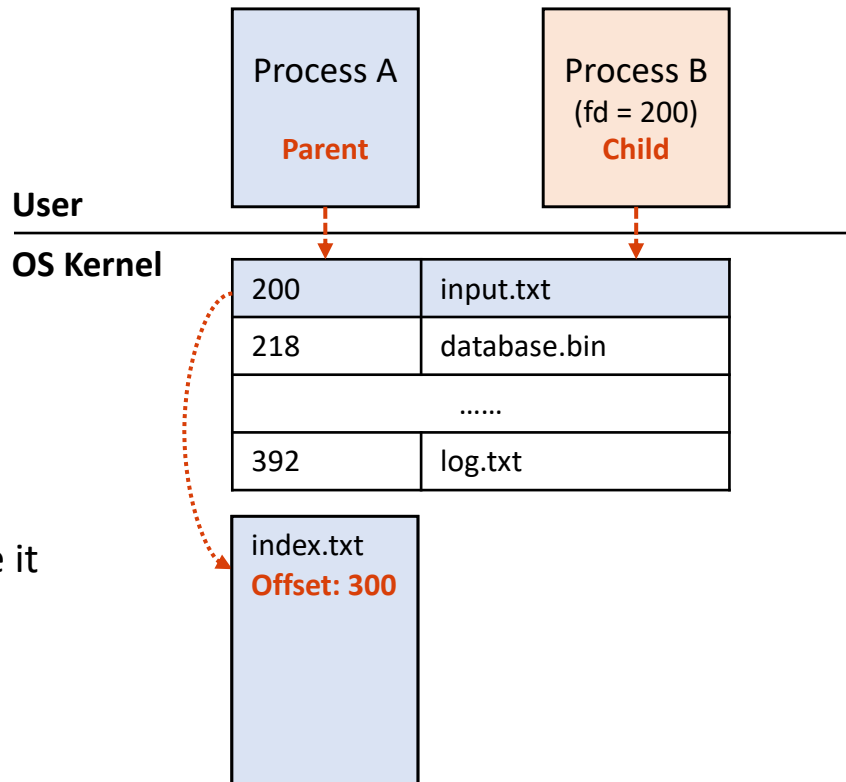
# LOW-LEVEL API INTERNALS: HOW?

- Let's do more exercise

```
... many #include ...
#define BSIZE 100

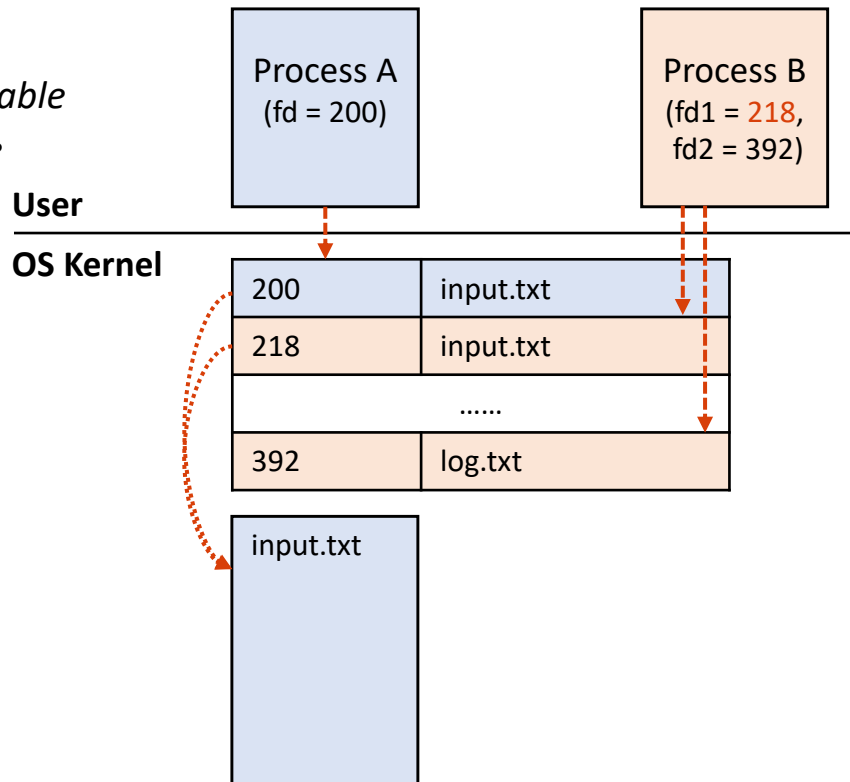
int main(void) {
    ...
    switch (pid = fork()) {
        case 0:
            sleep(3); read(fd, buf1, BSIZE);
            break;
        default:
            read(fd, buf1, BSIZE);
            close(fd);
    }
}
```

- Note
  - Proc A (parent) read data from fd and close it
  - The fd will **remain available** to Proc B



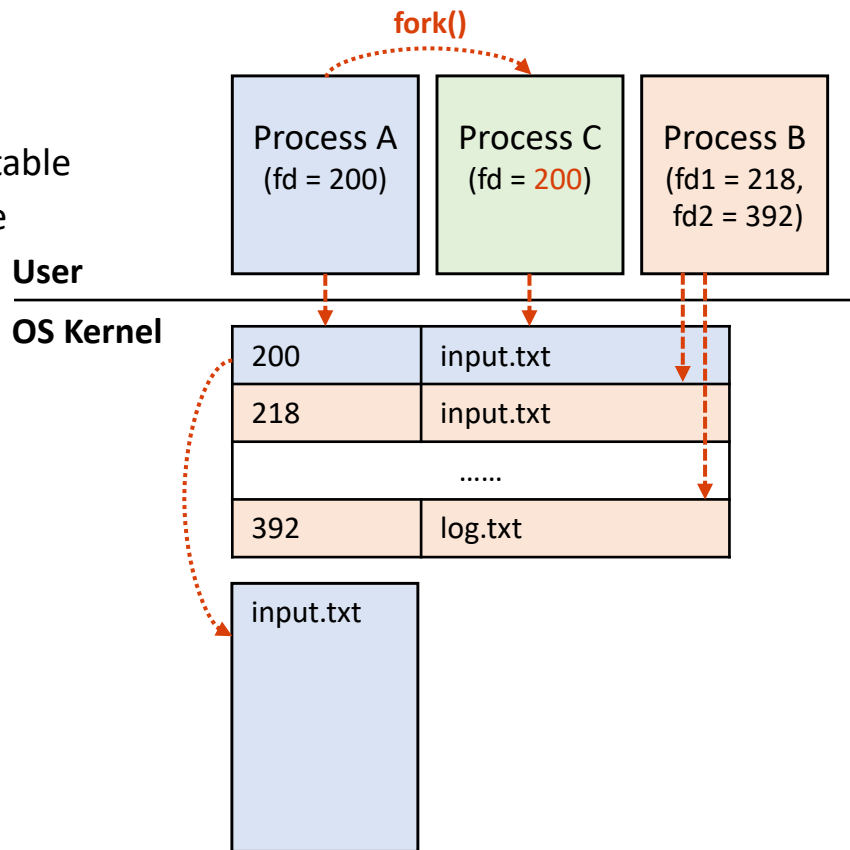
# LOW-LEVEL API INTERNALS: SUMMARY

- File descriptors (fd)
  - A unique identifier for an open file
    - Each process has an *open file descriptor table*
    - OS also has a *system-wide descriptor table*
  - Properties of file descriptors
    - The fd can **point to the same file**



# LOW-LEVEL API INTERNALS: SUMMARY

- File descriptors (fd)
  - A unique identifier for an open file
    - Each process has an open file descriptor table
    - OS also has a system-wide descriptor table
  - Properties of file descriptors
    - The fd can **point to the same file**
    - The fd can be **copied and aliased**
      - Proc A and C share the offset
      - Proc A and B do not



# TOPICS FOR TODAY

---

- Part II: Filesystem internals
  - Manage resources
    - How OS manages high-level I/O internally?
    - How OS manages low-level I/O internally?



# Thank You!

M/W 12:00 – 1:50 PM (LINC #200)

Sanghyun Hong

[sanghyun.hong@oregonstate.edu](mailto:sanghyun.hong@oregonstate.edu)



**Oregon State**  
University

**SAIL**  
Secure AI Systems Lab