

File I/O Syscalls & StdLib

Topics

- What **syscalls** can we use to **access files** (like `write()`)?
- Why are there **stdio functions**, like `fprintf()`, **in addition to `write()`**?
- Why do writes sometimes **not happen right away**?

Basic I/O System Calls

File Offset

- File offset

..

- Offset is where both `read()` and `write()` occur (one pointer).

- Move it to an arbitrary position using `lseek()`

- `read()` and `write()` automatically increments the offset:

..

IO Syscalls

- 5 basic system calls for file I/O
 - open
 - read
 - write
 - close
 - fcntl - File control

open()

- `open()` receives 2 or 3 parameters:
 - `int open(const char *pathname, int flags);`
 - `int open(const char *pathname, int flags, mode_t mode);`
- `flags`: ..
 - Must be one of: `O_RDONLY`, `O_WRONLY`, or `O_RDWR`
Read only, write only, read/write
- **Flags can also be bitwise-or'd with others such as:**
 - `O_RDWR | O_APPEND`: All write actions happen at end of file
 - `O_WRONLY | O_CREAT`: If file does not exist, then create it.
 - `O_RDWR | O_TMPFILE`: Create an unnamed temporary file
 - `O_WRONLY | O_TRUNC`: Truncate file when opened for writing
- **Bitwise-or sets individual bits in a bit vector,**
e.g., `O_RDWR | O_CREAT`

open() cont

- Recall:

- `int open(const char *pathname, int flags);`

- `int open(const char *pathname, int flags, mode_t mode);`

- mode

- ..

- (flags `O_CREAT` or `O_TMPFILE`) – see [man 2 chmod](#) to learn bit masks

- `S_IRWXU`: User can read/write/execute

- `S_IRUSR | S_IWUSR`: ..

- Return Value

- ..

- a handle for the file to read and write:

- it's a small non-negative integer (`int`)

- It could change every time you open the file.

write()

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

- `write()` writes `buf` to a file descriptor and ..
- `man 2 write` important points:
 - ..
 - insufficient space on disk
 - call interrupted by a signal handler
- Writing takes place **at the file offset**, and offset is incremented by the number of bytes actually written.
- If unsuccessful, return **-1**

read()

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

- `read()` reads from a file descriptor and ..
- ``man 2 read`` **important points:**
 - read operation commences at the file offset, which is **incremented** by the number of bytes read.
 - If file offset is at or past the end of file, ..
 - Not an error if **# bytes read < # bytes requested**
 - fewer bytes available right now (near end-of-file or reading pipe/terminal)
 - or `read()` was interrupted by a signal
 - If unsuccessful, return **-1**

close()

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

- closes the file descriptor.
- Writes any remaining buffered data to file.
- Returns **zero** on success. On error, **-1** is returned, and errno is set to indicate the error.

lseek()

```
off_t lseek(int fd, off_t offset, int whence);
```

- Manually adjust the file offset.

– `man lseek`

- `whence`

– ..

- `SEEK_SET`: Start of file
- `SEEK_CUR`: Current offset
- `SEEK_END`: End of file (1st byte after end of data)

– `offset` is always added.

– Can seek past end of file:
next write will extend file with 0's.

ABCD: lseek

- Suppose a file has 6 bytes (i.e., EOF is at 6) and the current file offset is 4:

Index	0	1	2	3	4	5	6	7	...
Content	H	e	l	l	o	!	<EOF>		



–Note: <EOF> does not actually appear *in* the file.

- What character is read when doing a `read()` of 1 byte after the following calls (in isolation)?

- 1) `lseek(fd, 4, SEEK_SET)`
- 2) `lseek(fd, -1, SEEK_CUR)`
- 3) `lseek(fd, -1, SEEK_END)`

- (a) l
- (b) o
- (c) !
- (d) none

fcntl()

```
int fcntl(int fd, int op, ... /* arg */ );
```

- File control

- man fcntl

- It can do many things, such as

- modify flags and mode used when file was opened:

- `op = F_SETFL` (set flag)

Activity: Files

- Write a program that:
 - Creates a new file named “tmp” in current folder
 - Writes X bytes to a file
 - Write a string like “Hello World!”; your choice!
 - Moves the file offset backward by $X/2$ bytes
 - Reads and prints out from the offset to EOF
 - Closes the file

Buffered I/O

Categories of File Functions

- **Syscalls**

- I/O functions that are system calls:
`write()`, `read()`, etc. (previous slides)

- **..**

- All I/O functions that start with f:

- `fprintf()`, `fscanf()`, `fputs()`, `fgets()`, `fput()`, `fget()`, etc.

- The same functions without f:

- `printf()`, `scanf()`, `puts()`, `gets()`, etc.

- **What's the difference?**

- Let's look at `write()`, `fprintf()`, and `printf()`

write() vs fprintf()

- `write()` directly sends data to the kernel, `fprintf()` ..
 - Uses `write()` under the hood.
 - Because of this, ..
 - `fprintf()` generates fewer syscalls, which gives better performance (less overhead).
- **File Descriptor vs FILE stream**
 - Syscalls like `write()` take..
`ssize_t write(int fd, const void *buf, size_t count);`
 - Library functions like `fprintf()` take.. a stream (`FILE *`)
`int fprintf(FILE *stream, const char *format, ...);`

Stream vs File Descriptor

- What is a Stream

FILE *stream

- Convenient wrapper around a file descriptor.

Used by the stdio functions.

- Think of this as

..

- Converting Stream \Leftrightarrow File Descriptor

- You can get the file stream from a file descriptor with:

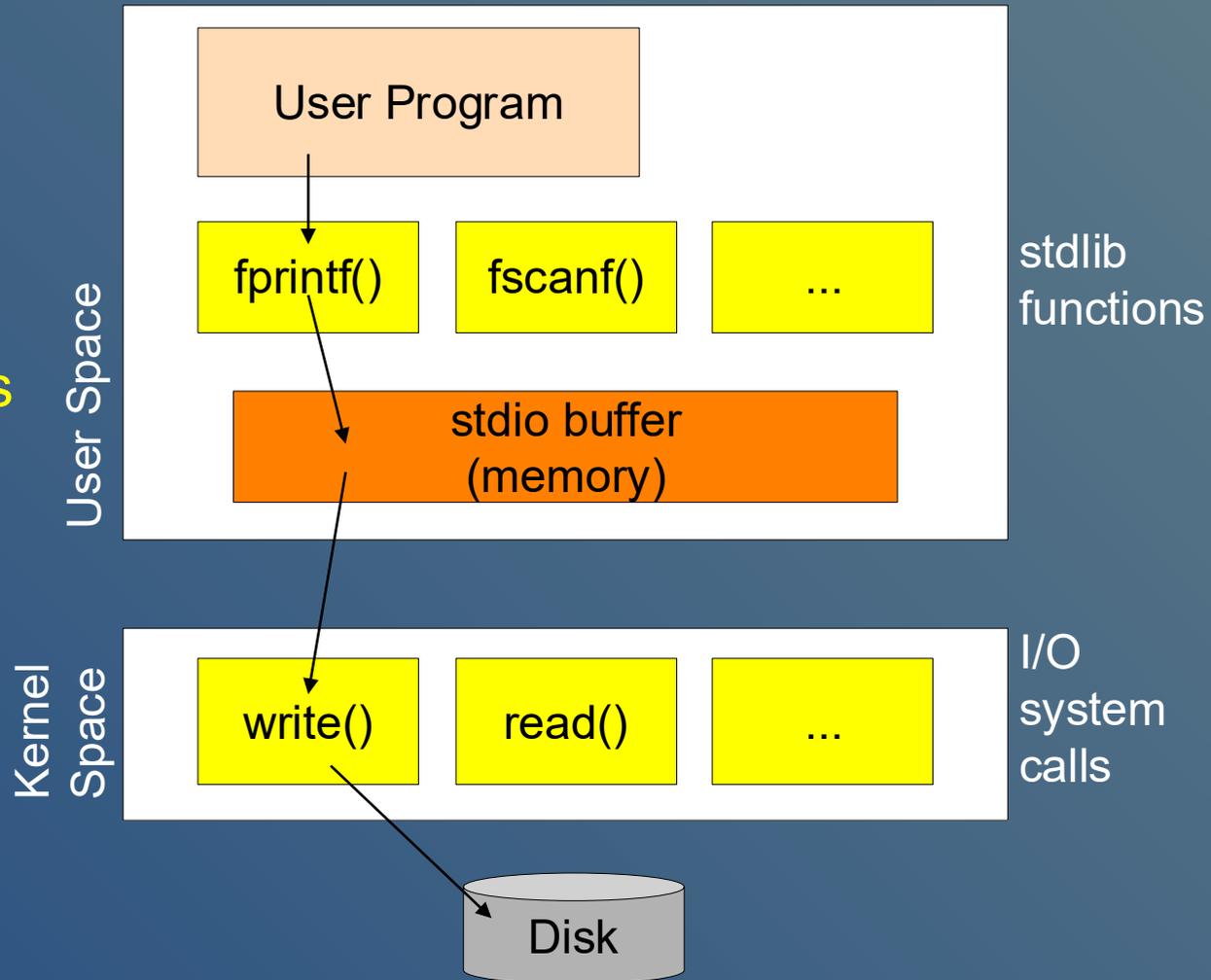
..

- You can get the file descriptor from a file stream with:

..

Relationship

- User program has **data** (in memory) to write.
- It **calls library function**.
- Data written into **library's buffer**.
- Later executes **syscall to write** to kernel.
- Kernel will **write to disk**.



Activity: Kernel Write

- Write a program that will:
 - `open()` a file named `tmp`,
 - `write()` a string (your choice) to `tmp`,
 - infinite loop that calls `sleep()` for 30 seconds each loop.
- Experiment
 - Run it in the background
 - Did it write to the file `tmp`. Check with `cat`.
(It should.)
- When done, delete `tmp` from the command line.

Activity: Library print

- Write another program that will
 - `fopen()` a file named `tmp`,
 - `fprintf()` a string to `tmp`,
 - infinite loop that calls `sleep()` for 30 seconds each iteration.
- Run It
 - Run it in the background
 - Did it write to the file `tmp`. Check with `cat`.
(It should not!)
- Experiment
 - Change to close file after writing. Repeat running it.
It should write to file.

Buffering

- Explain Behaviour

- Why did `fprintf()` not write to the file when the file is left open?

- Why did it write when we closed?

- Answer:..

- `fflush()` immediately sends the buffered data to the kernel.

- Calling `setbuf()` with NULL as the buffer automatically does flushing.

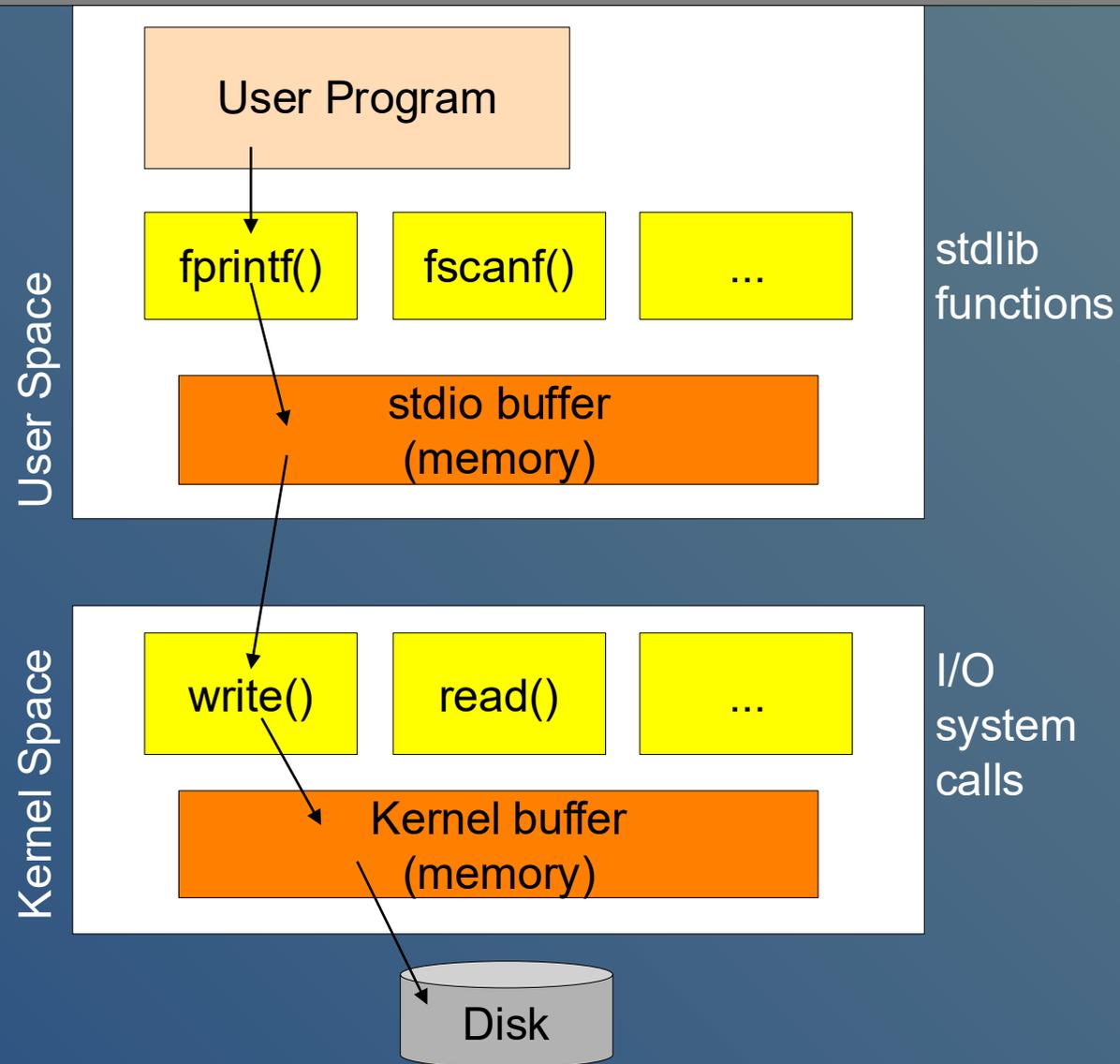
- Read ``man setbuf`` for more details.

Activity: fflush()

- Change Previous Program with fprintf():
 - Add fflush() call after printing
- Run it and see if it writes to tmp. (It should.)

Kernel Buffering

- Kernel has **read/write buffers** too.
- Even kernel does **not immediately** write to disk.



Kernel Buffering

- Can force kernel to flush buffer with `fsync()`
 - Using `O_SYNC` when with `open()` automatically does `fsync()`.
- Parallel between user buffering and kernel buffering
 - `fflush()` and `fsync()`: both flush their buffer.
 - `setbuf()` with a `NULL` buffer and `O_SYNC`:
 - ..

Blocking vs Non-Blocking I/O

- **Blocking call**

..

–**E.g.**, a blocking `read()` call doesn't return until there's something to read.

- **Non-blocking call**

–**O_NONBLOCK** flag

(either with `open()` or with `fcntl()` & `F_SETFL`)

–If an operation can't be done immediately, then

..

Summary

- 5 Syscalls for File Access

- int **open**(const char *pathname, int flags);

- int **open**(const char *pathname, int flags, mode_t mode);

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

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

- int **close**();

- off_t **lseek**(int fd, off_t offset, int whence);

- Syscalls vs Library functions

- write**() vs **fprintf**()

- Non-buffered vs buffered IO

- File descriptor (**int**) vs stream (**FILE***)