File I/O:
Syscalls and StdLib

Adapted by Joseph Lunderville
from slides by Dr. Brian Fraser
and course material by Dr. Steve Ko

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:***

  a pointer that points to a byte in the file where you operate

  - Offset is used by `read()` and `write()` (one pointer)
  - Move it to an arbitrary position using `lseek()`
  - `read()` and `write()` automatically move the offset: subsequent calls can just continue with the next data
  - Like a tape head (...because in olden days, it was)

I/O Syscalls

- *5 basic system calls for file I/O*
  - `open`
  - `read`
  - `write`
  - `close`
  - `fcntl` (“file control”)

open()

- *`open()` receives 2 or 3 parameters (depending on `flags`)*
  - `int open(const char *pathname, int flags);`
  - `int open(const char *pathname, int flags, mode_t mode);`
- *`flags`: access mode and a creation mode*
  - Must be one of: `O_RDONLY`, `O_WRONLY`, or `O_RDWR` (read only, write only, read/write) – never `O_RDONLY|O_WRONLY`!
- *`flags` can also be bitwise-or'd with others such as*
  - `O_RDWR|O_APPEND`: All write actions happen at end of file
  - `O_WRONLF|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'd)

- *Recall*
  - int open(const char *pathname, int flags);
  - int open(const char *pathname, int flags, mode_t mode);
- *mode: sets file permissions when creating file*
  (flags `O_CREAT` or `O_TMPFILE`)
  - `S_IRWXU`: User can read/write/execute
  - `S_IRUSR|S_IWUSR`: User can read/write
- *Return Value*
  - **File descriptor:** 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()

- *write() writes buf to a file descriptor and returns the number of bytes written*
  - `ssize_t write(int fd, const void *buf, size_t count);`
- man 2 write important points:
  - Writing takes place *at the file offset,* and offset is incremented by the number of bytes actually written
  - Number of bytes written *may be less than count:*
    - insufficient space on disk
    - call interrupted by a signal handler

read()

- read() reads from a file descriptor and returns the number of bytes read
  - `ssize_t read(int fd, void *buf, size_t count);`
- `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, no bytes are read, and read() returns zero.
  - Not an error if bytes read less than count:
    - fewer bytes available right now (near end-of-file, reading pipe/terminal)
    - read() was interrupted by a signal

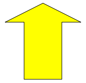
close()

- *closes the file descriptor*
  - `int close(int fd);`
  - Writes any remaining buffered data to file

`lseek()`

- *Manually adjusts the file offset*
  - off_t lseek(int fd, off_t offset, int whence);
- *whence: from which location we want to adjust the file offset*
  - SEEK_SET: Start of file
  - SEEK_CUR: Current offset
  - SEEK_END: End of file (byte after last byte in file)
- *Important points*
  - `offset` is always added
  - Can seek past end of file (write will extend file with 0’s: “sparse files”)

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



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

- *File control*
  - int fcntl(int fd, int op, … /* arg */);
- Can do many things, such as modify `flags` and `mode` specified when file was opened:
  - op = F_SETFL (set flag)
  - Note this (anti?) pattern in software: “util”, “misc”, “thingy”…

Activity - Files

- Write a program that (15m):
  - Creates a new file named “tmp” in current folder
  - Writes *X* bytes to a file
    - Suggestion: write a string like “Hello World!”
    - Your choice, content and length not important
  - 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)
- *Standard library (stdio) functions*
  - 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()` (usually) manages a buffer in memory, and writes to the buffer*
  - Uses `write()` under the hood if buffer full, file closed, `fflush()`
  - Because of this, `fprintf()` is sometimes called buffered I/O
  - Generates fewer syscalls, which gives better performance…
    - When making lots of small writes, at least!
    - Kernel also buffers writes (but this gets complicated *fast:* filesystem integrity, write ordering issues)
- *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:
    int fprintf(**FILE *stream**, const char *format, ...);

Stream vs. File Descriptor

- *What is a Stream?*

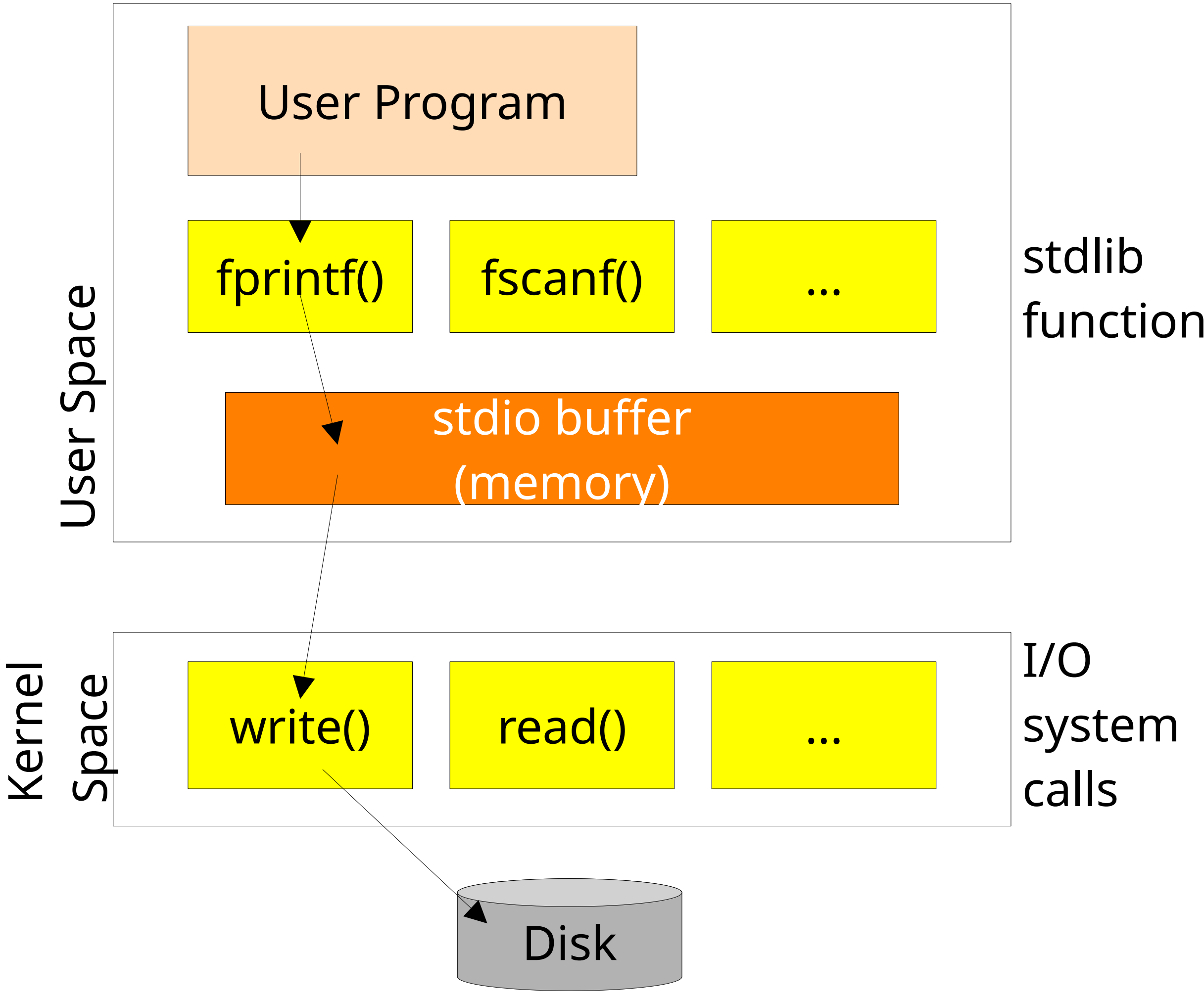
  FILE *stream

  - Convenient wrapper around a file descriptor
  - Used by the stdio functions
  - Think of this as a file descriptor plus a buffer backing it up
    - Also about platform portability… not our problem, since we’re explicitly targeting Linux (or maybe POSIX)

- *Converting Stream ⟺ File Descriptor*
  - You can get the *stream* from a *file descriptor* with `fdopen()`
  - You can get the *file descriptor* from a *stream* with `fileno()`

Relationship

- User program has data (in memory) to write…
- It calls a library function
- Data is (initially) written into library's buffer
- During some library call, executes syscall to write to kernel
- Kernel will write to disk (eventually, but that's not our problem)



Activity - Kernel Write

- Write a program that will (15m):
  - `open()` a file named “tmp”
  - `write()` a string (of your choice, like before) to tmp
  - Wait forever: call `sleep(10)` in a loop
- *Experiment*
  - Run it in the background (“`./a.out` **`&`**”)
  - 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 (15m):
  - `fopen()` a file named “tmp”
  - `fprintf()` a string to tmp
  - Wait forever: call `sleep(10)` in a loop
- *Experiment 1*
  - Run it in the background (“`./a.out` **&**”)
  - Did it write to the file tmp? Check with cat (it should not!)
- *Experiment 2*
  - Change it to `fclose()` after writing and repeat above procedure
  - Now it should write to tmp!

Buffering

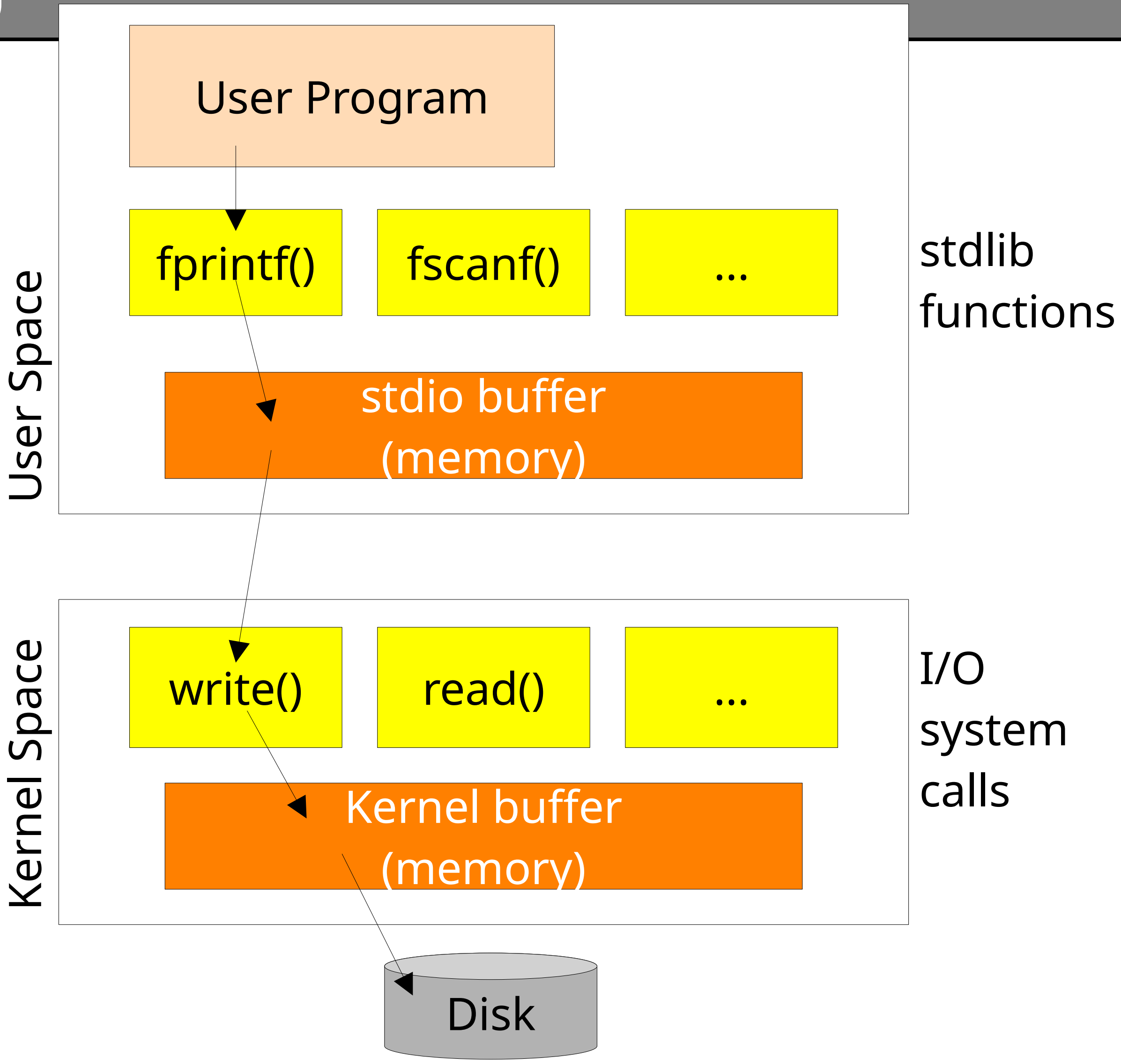
- Explain Behaviour
  - Why did `fprintf()` not write to the file when the file is left open?
  - Why did it write when we closed it?
  - If you use python: “with open(‘tmp’) as f: …” (or “del f”)
- `fflush()` immediately sends the buffered data to the kernel
  - Calling `setbuf()` with `NULL` as the buffer also automatically flushes
  - 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
- But the kernel will write when it can: not limited to calls
- The kernel must compromise between performance and potential for catastrophe
- Filesystem metadata usually journaled so you don't lose whole files
- Consistency within a file usually left to the application



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`
    - both automatically perform no buffering

Blocking vs. Non-Blocking I/O

- ***Blocking call***
  - Doesn't return until the operation can be done
  - E.g., a blocking `read()` call doesn't return until there's something *to* read
- ***Non-blocking call***
  - `O_NONBLOCK` flag (either passed to `open()` or with `fcntl()` and `F_SETFL`)
  - If an operation can't be done immediately, then the call returns an error, typically `EAGAIN`
  - *Completing* still does not mean it does everything you asked!
    - E.g., `read()` doesn’t necessarily read *all* `count` bytes… just more than zero
    - Has to return `EAGAIN` as an error to distinguish from EOF!

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()`
  - Unbuffered vs buffered IO
  - File descriptor (`int`) vs stream (`FILE*`)