CSCI 4061 - Intro to Low-Level I/O

# Threads and Processes

* Both are ways to enable multiple streams of execution at once
* Threads share and live in the same address space- see the same virtual memory
  + Easy cooperation, better efficiency, but less isolation
* Every process has its own address space- different memory in all of the virtual memory addresses
  + We saw this with fork()- parent and child could store basically different data at the same address & coexist at the same time; independent copy
  + Better isolation, but less efficient
  + Need more memory

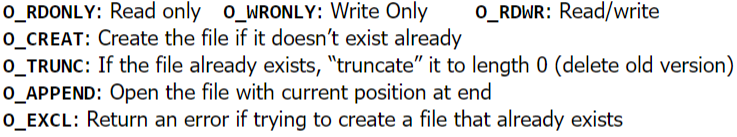
# Moving from stdio to I/O System Calls

* Major Changes
  + No longer a leading f, e.g., fopen() -> open()
  + We have to get more specific on how file is opened
    - O\_CREA|O\_TRUNC|O\_WRONLY as opposed to “w”
    - S\_IRUSR|S\_IWUSR - what’s this?
  + Sizes to read and write are directly in bytes as opposed to “elements”
  + Each write we invoke corresponds directly to a system call

## The open System Call

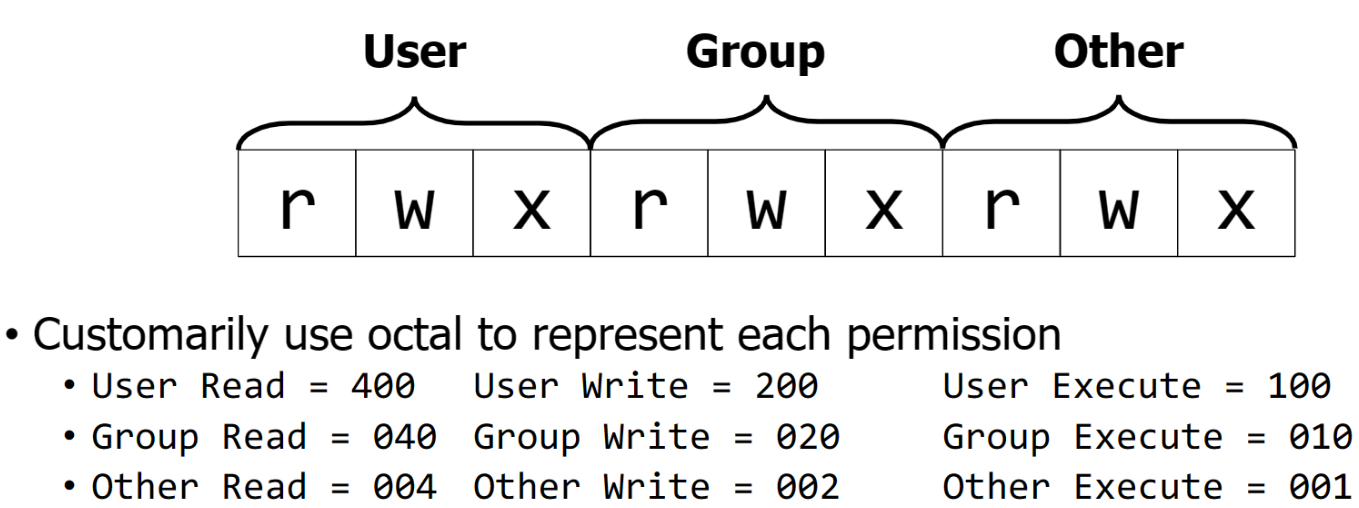
int open(const char \*pathname, int flags);

int open(const char \*pathname, mode\_t mode);

* + Returns a **file descriptor**- unique id for each open file
  + First argument is the name of the file, just as before
  + flags is a bit vector indicating which options we want enabled (file permissions)
    - Each option has a predefined name, combine them with bitwise OR
  + Options for open(): 
    - These give you much more control than fopen()- fopen() with “w” mode is equivalent to O\_CREAT | O\_WRONLY | O\_TRUNC

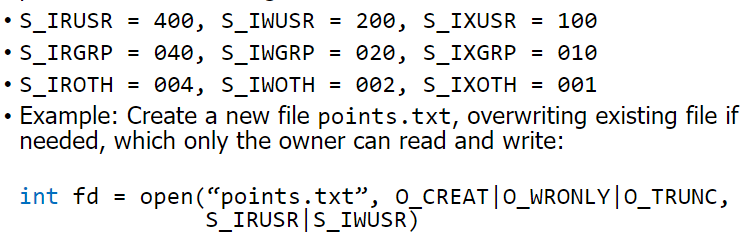
## Unix-Style File Permissions

* + Three operations to enable/disable on a file
    - Read
    - Write
    - Execute
  + Three different entities to enable/disable these operations for:
    - User- the owner of the file
    - Group- any user designated as part of the file’s assigned group
    - Other: Everyone else
  + Permissions are encoded as a bit vector; customarily use octal to represent each permission- bitwise OR-ing everything together enables a combination of permissions
  + The chmod command allows one to modify a file’s permissions
    - Chmod 700 secret\_script.py: owner had full permissions, nothing else
    - Chmod 640 project\_grades.csv: owner can read/write, group can read
    - Although, it is often easier to “add” or “subtract” permissions



## Back to open: Setting Initial Permissions

* + When you call open with the O\_CREAT flag, you can specify the file’s permissions as the third argument



## read: Consuming Data From Files

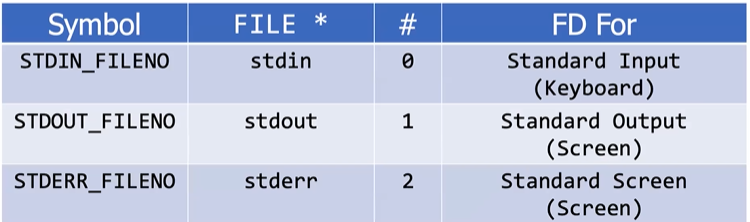
ssize\_t read(int fd, void \*buf, size\_t count);

* + - ssize\_t: signed size\_t value
    - fd: file descriptor of file to read from
    - buf: memory location to store file contents
    - count: number of bytes to read
  + Like fread(): Memory location to store data, returns amount read
  + Not like fread()- size to read and return value given as number of bytes, not “elements”

## Write: adding data to files

ssize\_t write(int fd, const void \*buf, size\_t count);

* Good Practice is to read and write in Chunks`
* Built-In Files
  + Every process comes with these three file descriptors automatically



# Buffering

* A **buffer** (noun) is a temporary chunk of memory where we store information that was just read in or is about to be sent out
* **Buffering** (verb) is the technique of accumulating pending I/O operations until there is a chuck to read/write all at once
  + **Full buffered**: defer completing a sequence of read/writes until a buffer (maintained within kernel) is completely full
  + **Line Buffered**: data written only when newline provided
  + **Unbuffered**: data written immediately
* Stdout vs Stderr
  + Stdout is line buffered: defer write until ‘\n’ appears
  + Stderr is not: write to screen immediately
* write() is buffered by OS
  + When you call write() to modify a file and it returns, guaranteed that your data has been copied into a kernel-maintained buffer
  + Not guaranteed that your bytes are actually reflected on disk
  + To force all changes to a file to be reflected on disk: int fsync(int fd);
    - Almost never do this- OS buffers for a reason
    - Disk devices can only write data in chunks (often 4096 bytes)
* fwrite() is buffered by stdio
  + When you call fwrite() to modify a file and it returns, guaranteed that your data

has been copied into a library-maintained buffer

* + Not guaranteed that your bytes are copied to kernel buffer
  + Process suddenly terminates (no cleanup)-> your data is gone
  + To force all changes to a file to be copied to kernel buffer:

Int fflush(FILE \*f);

* + Should you do this? Almost never - Stdio buffers for a reason
  + A system call is a very slow operation (hundreds of nanoseconds)
  + Lots of work to switch into OS code, modify protection settings
  + Again: coalesce lots of operations into one system call to improve performance
* Buffering Principles

