Last Updated: 2-2-23

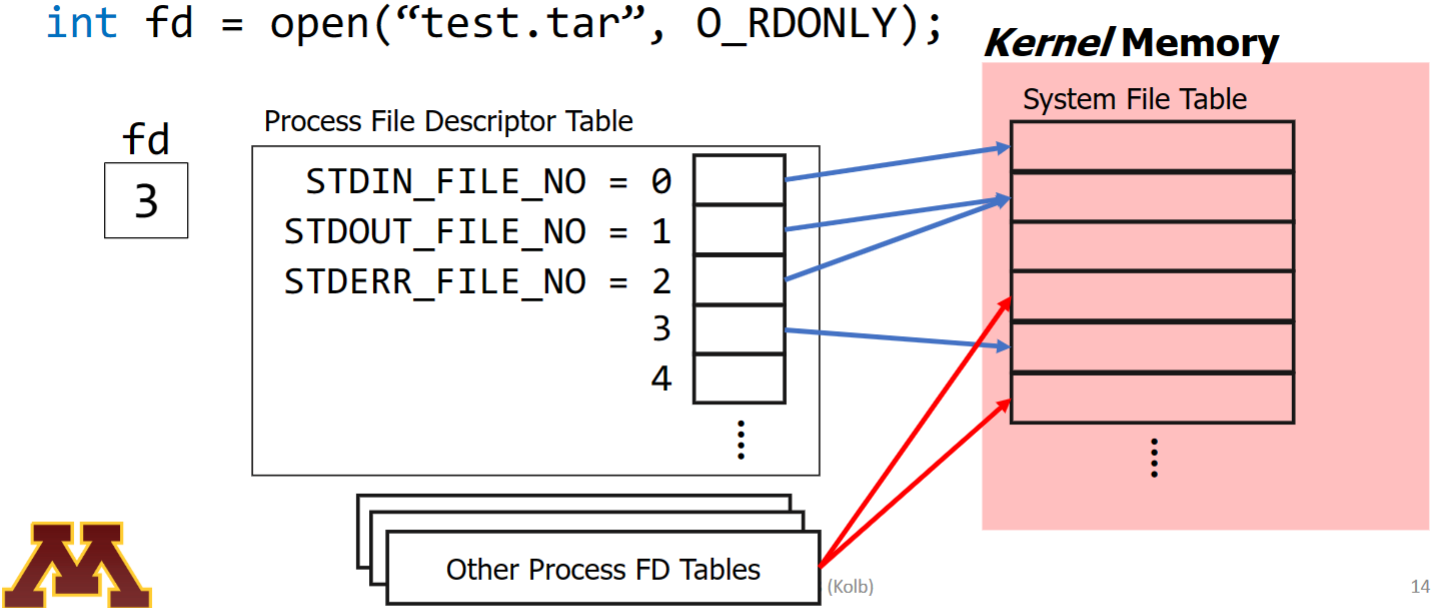
Advance I/O Redirection

# Anatomy of a FILE Structure (stdio)

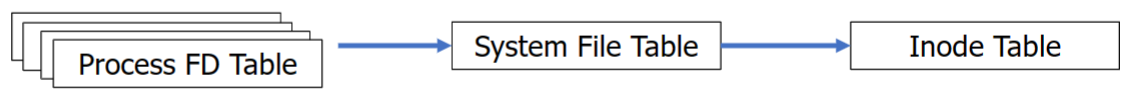
* The FILE\* data type that is returned from fopen()- what does it contain?
  + A buffer (or at least, a pointer to a buffer) that holds things in userspace that are accumulated after repeated calls to fwrite()
    - This buffer is passed to a write() system call that will write everything out all at once
  + A file descriptor which corresponds to an open file from the OS perspective
    - This file descriptor is passed to the write() system call as well

# File Descriptor, System File, and Inode Tables

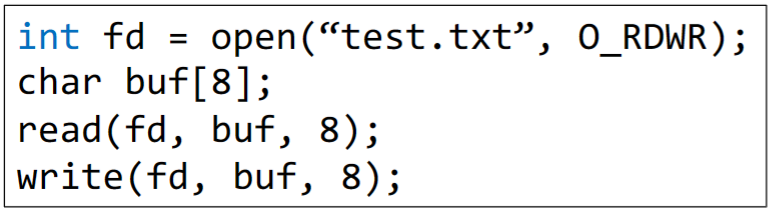
* The (type integer) file descriptor is basically an index into some table (or basically an array data structure) - one of these per running process that keeps track of all open files for that process; this is a **file descriptor table**



* The **system file table** (or global file table) is only changed or modified by system calls by the OS with its own code- meaning that my own C code could not modify it; each entry in a file descriptor table points to this globally shared file table
  + There is one entry per actively used file (after “open”, before “close”)- what does each entry contain?
    - File “offset” - the position we’ve been talking about all this time
    - Access mode - (read, write, read/write)
    - Number of FILE\* pointing to this entry (reference count)
    - Pointer to inode-table entry for this file
  + For now, assume that reference count starts at 1 - if two processes call open on the same file, they each get a sys file table entry, both entries have a reference count == 1
* Any file on the hard drive has an inode devoted to keeping track of the contents & metadata of that file; the inodes for all files are stored on the **Inode table**; each entry contains file metadata and reference count

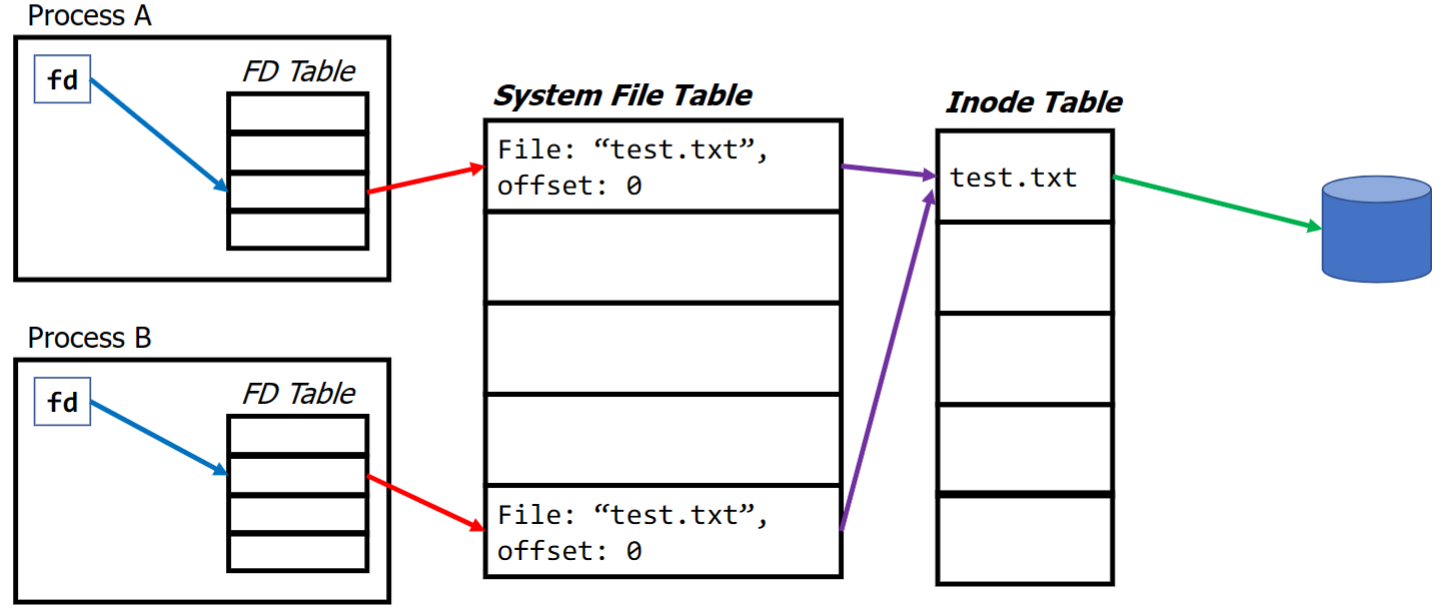


* In-Class Exercise (1) - two unrelated processes (neither parent or child) run the following code at the same time (assume the file contents at the start are as follows: ABCDEFGH):

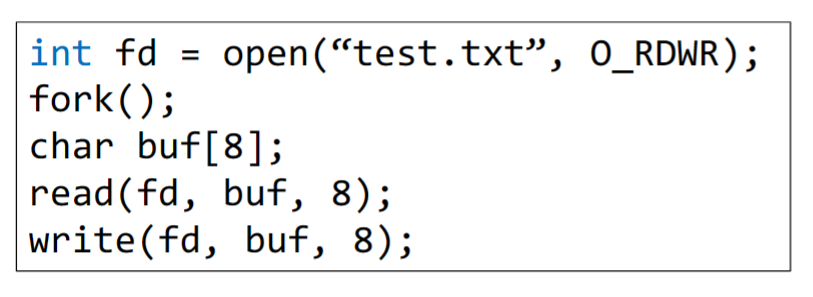


* + How many process file descriptor table entries are created?
    - 2 - two processes each open a file, a new FTE entry needs to be added in the file descriptor tables for each file
  + How many system file table entries are created?
    - 2 - while they are opening the same file, they open it independently (unrelated processes), so they are going to get their own entries in the global file table
  + How many inode table entries are created?
    - 1 - there is only one file that is being manipulated, it has only one inode, so there is only one inode table entry; however, if some other process happened to have this file already open, there would already be an inode table entry present, so no new entry would be created
  + What does “test.txt” contain after the code is executed?
    - Offsets are stored in the Global File Table for each entry- because there are two entries (one for each process); the change in offset for one process will not be reflected in the other process, so with an initial offset of 0, test.txt will contain:

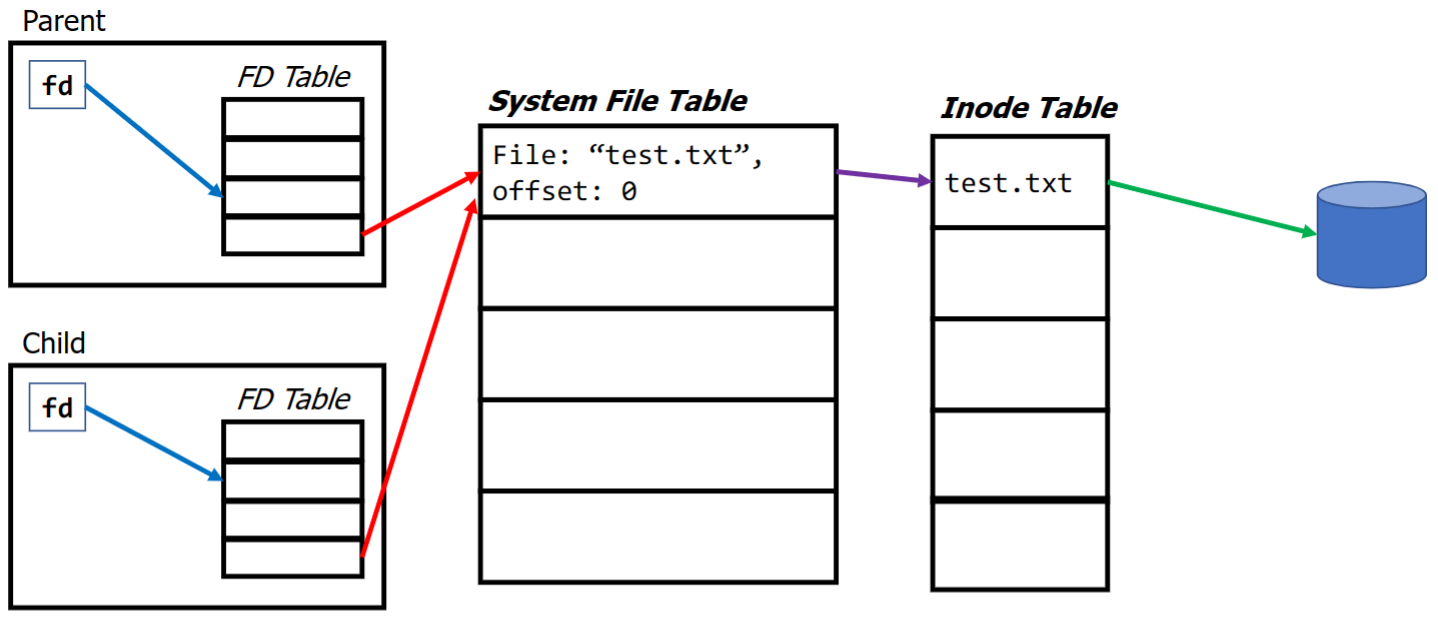
ABCDEFGABCDEFG



* In-Class Exercise (2) - Now assume that we call fork(), how does each instance change?



* + How many process file descriptor table entries are created?
    - 2 - the parent has a file descriptor table, when fork() is create, the child inherits a copy of the parent’s file descriptor table which results in two file descriptor tables with one entry each
  + How many system file table entries are created?
    - 1 - when the child inherits the file descriptor table from the parent, the reference to the system file table remains the same
  + How many inode table entries are created?
    - 1 - this pretty much is the same as the previous exercise
  + What does “test.txt” contain after the code is executed?
    - Because both processes’ file descriptor table entry reference the same entry in the global file table, the offset is “shared” between processes, meaning that when one process writes and updates the offset, the starting offset for the other process will be that new offset; with this, there is a possible sequence of events:
      * Process ‘A’ reads 8 bytes, offset = 8 (EOF)
      * Process ‘A’ writes 8 bytes, offset = 16 (EOF)
      * Process ‘B’ attempts to read 8 bytes from EOF and fails
      * Process ‘B’ writes 8 bytes of uninitialized content (garbage) from buf to file
    - An additional Sequence of events
      * Process ‘A’ reads 8 bytes, offset = 8 (EOF)
      * Process ‘B’ reads 8 bytes & fails because at EOF
      * Process ‘A’ writes 8 bytes, offset = 16 (EOF)
      * Process ‘B’ writes 8 bytes of garbage



# Concurrent File Access

* If you opened after for in Exercise (2), the files would be opened independently and the scenario would end up like in Exercise (1)
* Child Processes Inherit File Descriptors
* Actions in one process is reflected in the other due to the same reference to the system file table
* Files are a globally shared resource
  + All processes see the same set of files
* **POSIX FILE SEMANTICS** dictate what happens
  + Once a write system call has completed, its effects are immediately visible to any read on the file’s data

## Atomic Operators

* + - An operation is “atomic” if it is either executed to full completion or not executed at all; “all or nothing”
    - No partial effects
    - Huge theme in systems and systems programming
    - Ideal world- write is atomic
      * One write will be completed before another write executes; prevents overlap/interleave
      * The write order, however, is unpredictable
  + In theory, while processes are not guaranteed to be atomic (read/writes under 4096 bytes are close), threads are guaranteed to be completely atomic; in practice, this is not the case

# Files for Interprocess Communication

* Send data to other processes by writing to a file; receive data from other processes by reading from a file
* Bad idea:
  + If you need temporary storage, why use disk space that will eat up the hard drive
  + How does one process know when to read from the file (when the other process has completed writing to file); some sort of timing needs to happen
    - **Polling**: process repeatedly keep checking and react if the write has occurred
    - **Interrupts**: set up a notification that will alert the process that a write has occurred
* Motive idea for **pipes**