CS 3214: Computer Systems
Lecture 5: File Descriptors and Pipes

Instructor: Huaicheng Li

Sept 3 2024

VIRGINIA TECH™

Administrivia

- Congrats on ex0 submission!
- ex1 is up, due on September 17, 2024 11:59 PM

Recap & Today's Topics

- Processes manages many resources …
  - And one key aspect is the file descriptors they own
- Let's learn how file descriptors are managed by the OS
  - Why do we need it?
  - How does it work?
  - How do we use it?
  - Cute demos!

Linux/Unix: everything is a file ...

Unix File Descriptors

- A file descriptor is a **handle** that allows user processes to refer to files, which are sequences of bytes
- Unix represents many different kernel abstractions as files to abstract I/O devices, *e.g., disks, terminals, network sockets, IPC channels (pipes), etc.*
- Provide a uniform API, no matter the kind of the underlying object
  - read(2), write(2), close(2), lseek(2), dup2(), and more
  - May maintain a read/write position if seekable
  - But note: not all operations work on all kinds of file descriptors

Various Aspects of File Descriptors

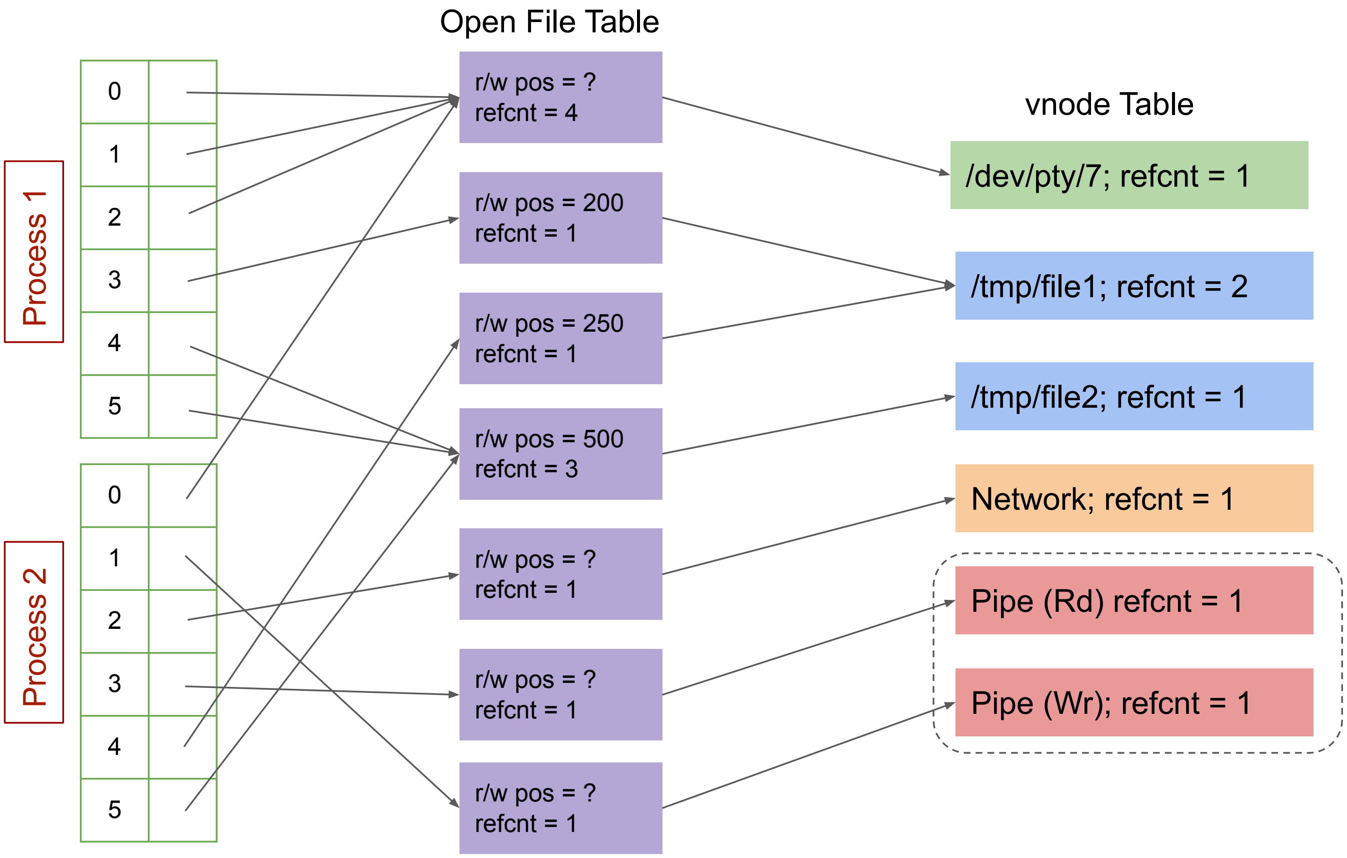
- Are represented using integers obtained from syscalls such as open()
- Are considered low-level I/O
- Are inherited/cloned by a child process upon fork()
- Are retained when a proces exec()'s another program
- Are closes when a process exit()s or is killed

Standard Streams

- By convention, stdin (0), stdout (1), stderr (2)
- Programs do not have to open any files; they are preconnected; thus programs can use them without needing any additional information
- Control programs (shell), or the program starting a program can set those up to refer to some regular file, terminal device, or something else
- When used, they access they underlying kernel object in the same way as if they'd open it themselves
- Programs should, in general, avoid changing their behavior depending on the specific type of object their standard streams are connected
  - Exceptions exist, e.g., flushing strategy of C's stdio depends on whether standard output is a terminal or not
  - Python 2 sys.stdout.encoding fiasco

File Descriptors – The Subtle Parts

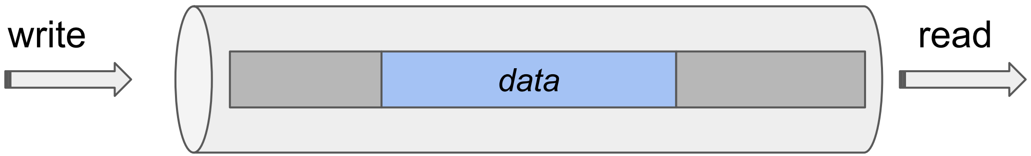
- To properly understand file descriptors, must understand their implementation inside the kernel
- File descriptors use 2 layers of indirection, both of which involve reference counting
  - (integer) file descriptors in a per-process table point to entries in a global open file table
  - per-process file descriptor table has a limit on the number of entries
  - each open file table entry maintains a read/write offset (or position) for the file
  - entries in the open file table point to entries in a global "vnode" table, which contains specialized entries for each file-like object
- File descriptor tables are (generally) per-process, but processes can duplicate and rearrange entries


Open File Table
vnode Table
Process 1
Process 2
0
1
2
3
4
5
0
1
2
3
4
5
r/w pos = ?
refcnt = 4
r/w pos = 200
refcnt = 1
r/w pos = 250
refcnt = 1
r/w pos = 500
refcnt = 3
r/w pos = ?
refcnt = 1
r/w pos = ?
refcnt = 1
r/w pos = ?
refcnt = 1
/dev/pty/7; refcnt = 1
/tmp/file1; refcnt = 2
/tmp/file2; refcnt = 1
Network; refcnt = 1
Pipe (Rd) refcnt = 1
Pipe (Wr); refcnt = 1

File Descriptor Manipulation

- *dup(int fd):* create a new file descriptor referring to the same file descriptor as fd, increment refcount
- *dup2(int fromfd, int tofd):* if tofd is open, close it. Then, assign tofd to the same open file entry as fromfd, increment refcount
- *close(fd):*
  - clear entry in file descriptor table, decrement refcount in open file table
  - if zero, deallocate entry in open file table and decrement refcount in vnode table
  - if zero, deallocate entry in vnode table and close underlying object
  - for certain objects (pipes, socket), closing the underlying object has important side effects that occur only if all file descriptors referring to it have been closed
- *lseek(fd, offset, …)*
- *opendir(), closedir(), readdir(), …*
- On fork(), the child inherits a copy of the parent's file descriptor table (and the reference count of each open file table entries is incremented)
- On exit() (or abnormal termination), all entries are closed

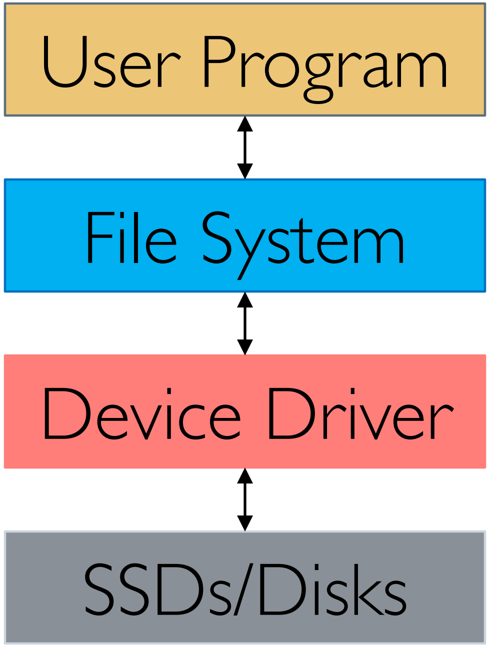
Pipes



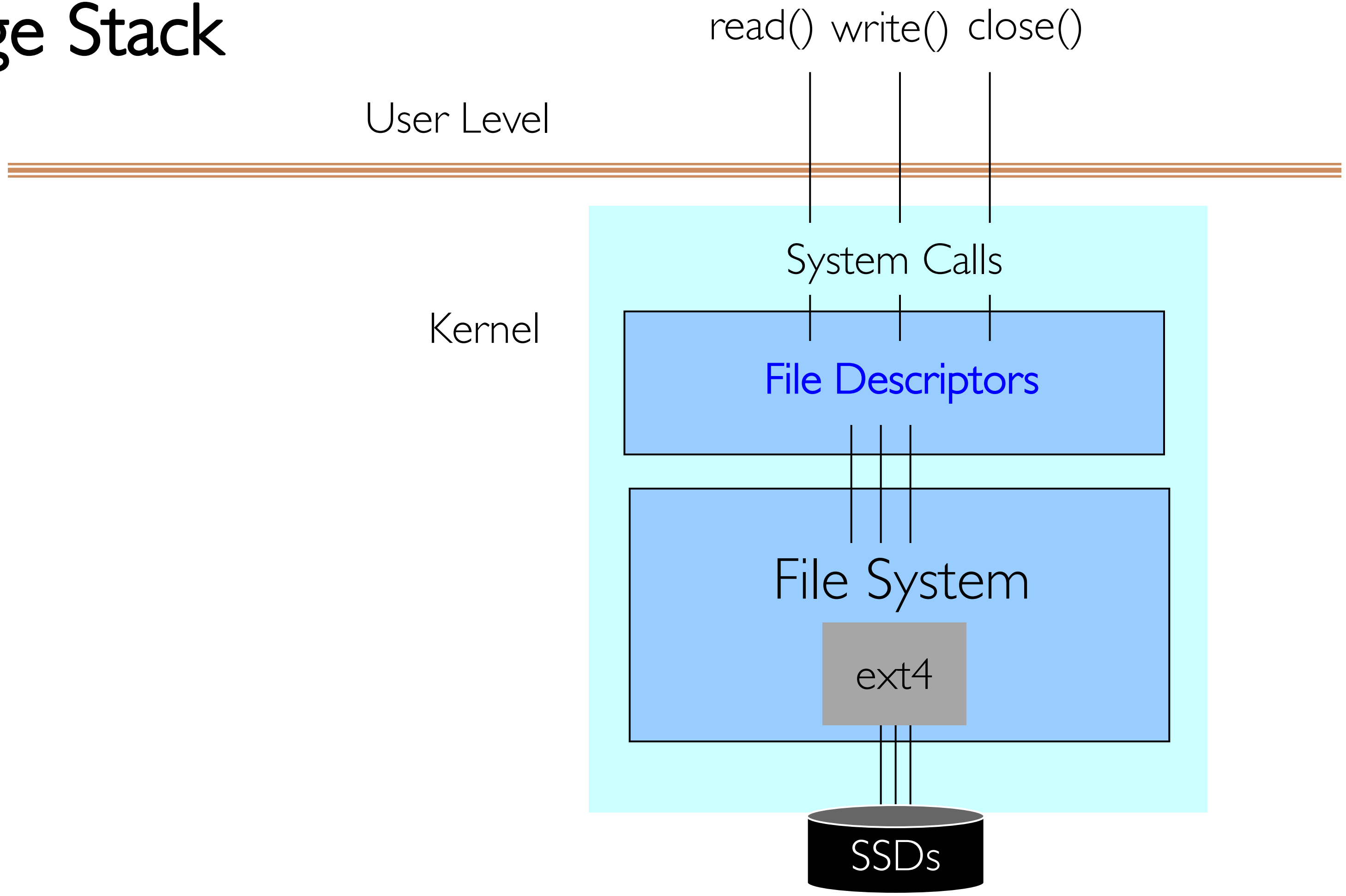
- **Writers:**
  - can store data in the pipe as long as there is space
  - blocks if pipe is full until reader drains pipe
- **Readers:**
  - drains pipe by reading from it
  - if empty, blocks until writer writes data
- Pipes provide a classic "bounded buffer" abstraction that
  - **is safe:** no race conditions, no shared memory, handled by kernel
  - **provides flow control that automatically controls relative progress:** e.g., if writer is BLOCKED, but reader is READY, it'll be scheduled. And vice versa.
  - **Created unnamed;** file descriptor table entry provide for automatic cleanup

More for Reference

- File path
  - Absolute path (e.g., /usr/bin/ls)
  - Relative path (e.g., ./a.out)
- File types
  - Regular
  - Block / character
  - Socket
  - Directory
  - Links
  - …
- File/Storage Stack



The Storage Stack



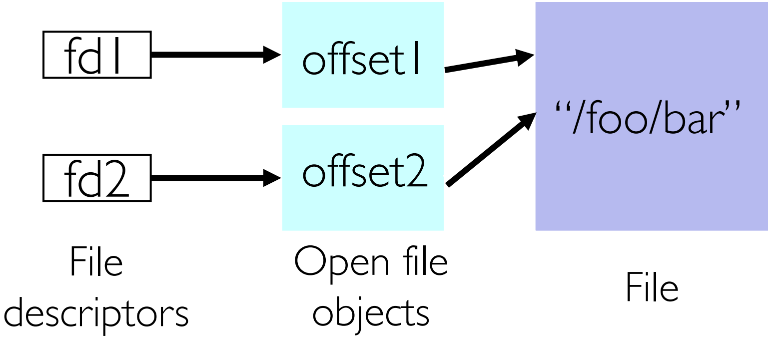
- *int fd = open(const char *path, int oflag, ...);*

  File Descriptor

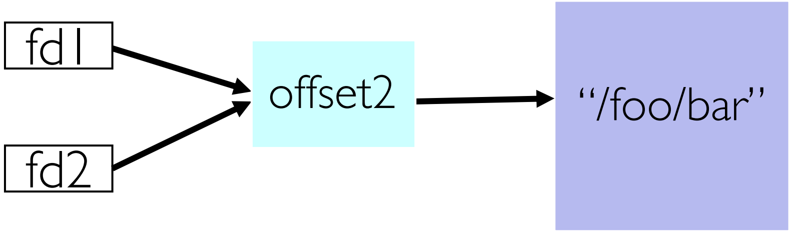
- *ssize_t ret = write(int fd, void *buf, size_t nbyte);*
- *ssize_t ret = read(int fd, void *buf, size_t nbyte);*
- *ssize_t ret = close(int fd);*

Accessing Open Files

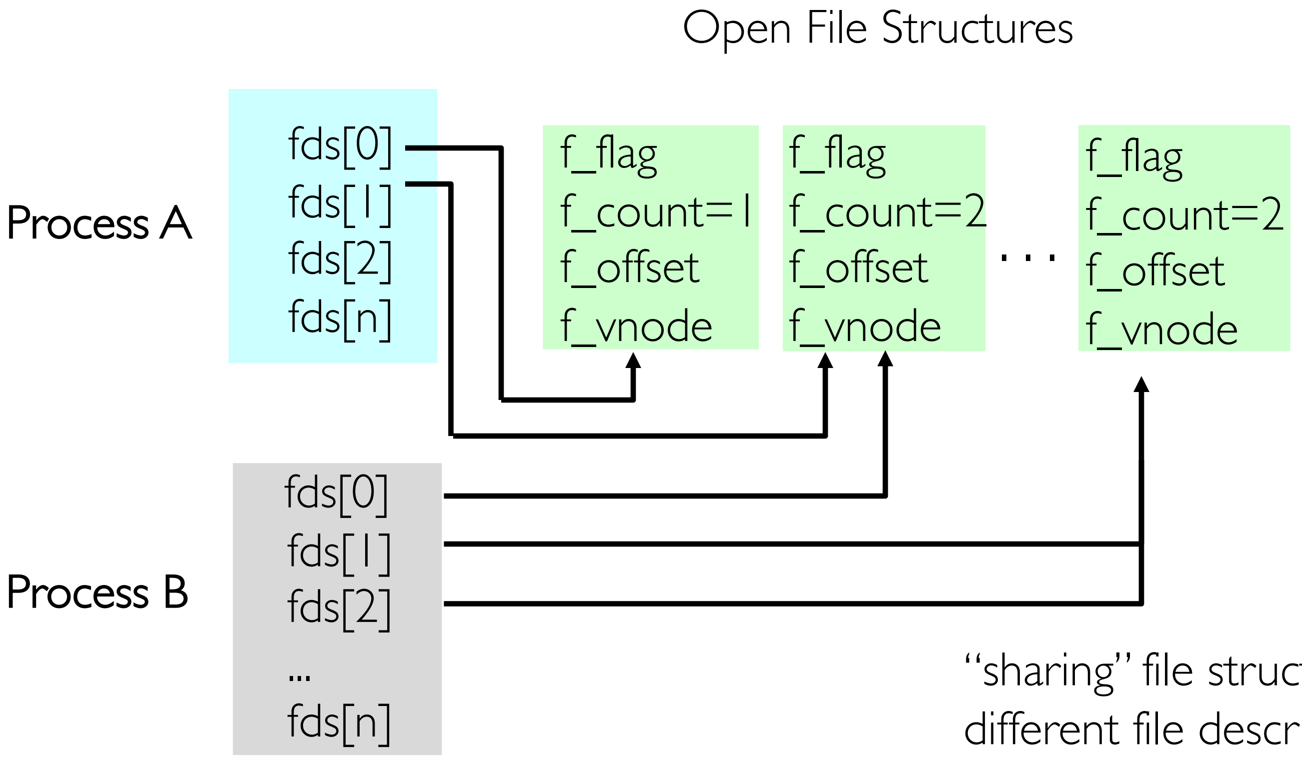
- Two opens of the same file yield independent sessions



- Two opens of the same file yield independent sessions

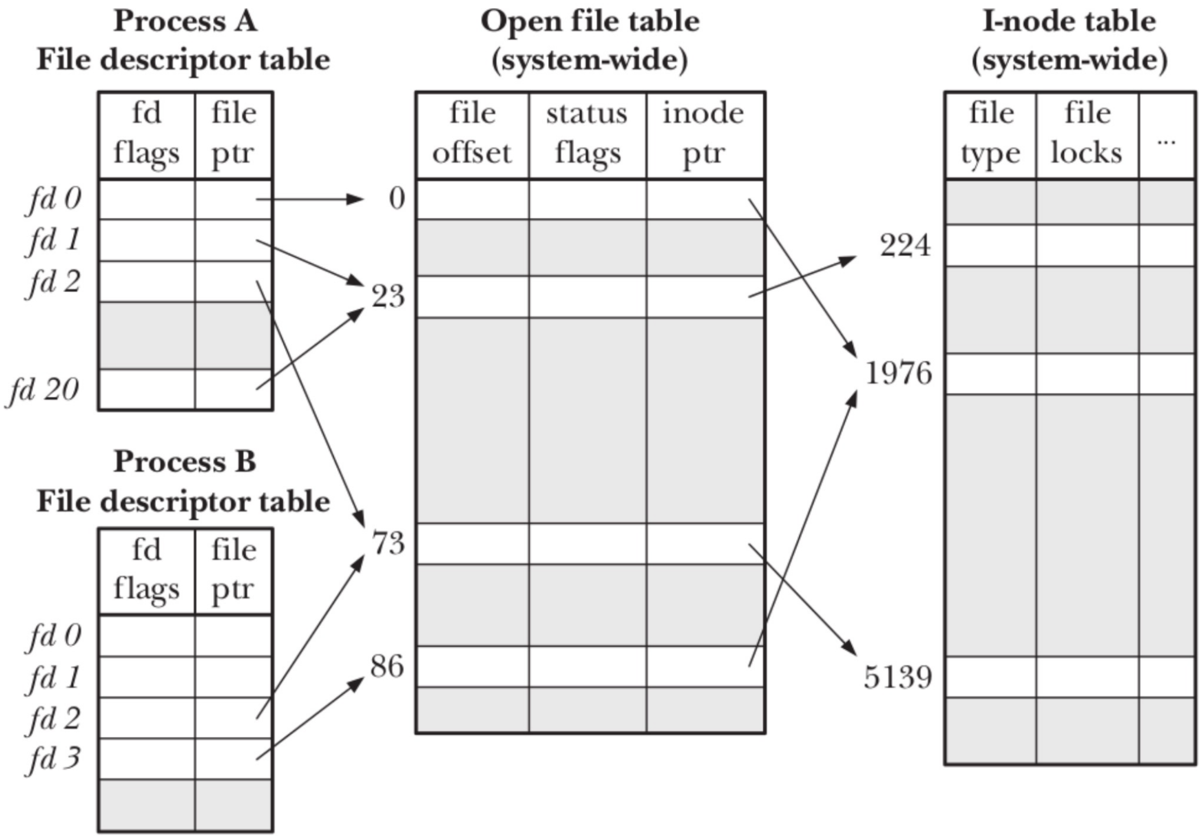


Some associated structures in kernel space



"sharing" file structure but have different file descriptors

Linux FDs



Process A
File descriptor table
fd flags
file ptr
fd 0
fd 1
fd 2
fd 20
Process B
File descriptor table
fd flags
file ptr
fd 0
fd 1
fd 2
fd 3
Open file table (system-wide)
file offset
status flags
inode ptr
0
23
73
86
I-node table (system-wide)
file type
file locks
...
224
1976
5139