CS110 Winter 2020 Midterm Review

Shrey Gupta + Andrew Benson

Slides contributed by Kristine Guo, Peter McEvoy, Armin Namavari, Ryan Eberhardt, Grace Hong, Hemanth Kini

Exam Date & Time

CS 110

Friday, February 14, 2020
NVIDIA Aud: A-S
Cubberley Aud: T-Z
1:30pm - 2:50pm

CS 110

Exam Deets

- Bring your laptop and charger
- Exam will be administered on Bluebook
  - Download software before exam.
- If you don’t have a working laptop, we need to know by midnight tonight
- Exam material emphasizes assignments, sections, lecture, readings (in order of decreasing emphasis)

Outline

Part 1: Filesystems

- Inodes, Directories, Links
- File descriptor table, open file table, vnode table
- System calls (open, close, read, write, dup, dup2, pipe)

Part 3: Signals

- Signal blocking and handlers
- Race conditions and sigsuspend

Part 2: Multiprocessing

- Processes and Virtual Memory
- fork, execvp, waitpid
- Pipes and multiprocessing

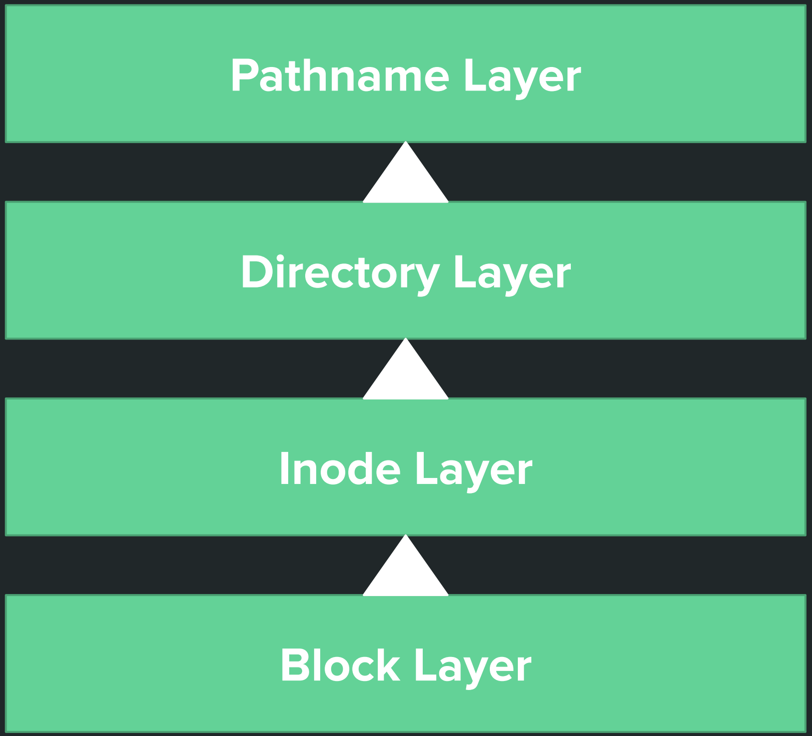
Part 4: Scheduling

Part 5: Threads

- Thread Syntax
- Race Conditions, Mutex

Part 1: Filesystems

Filesystem Layers

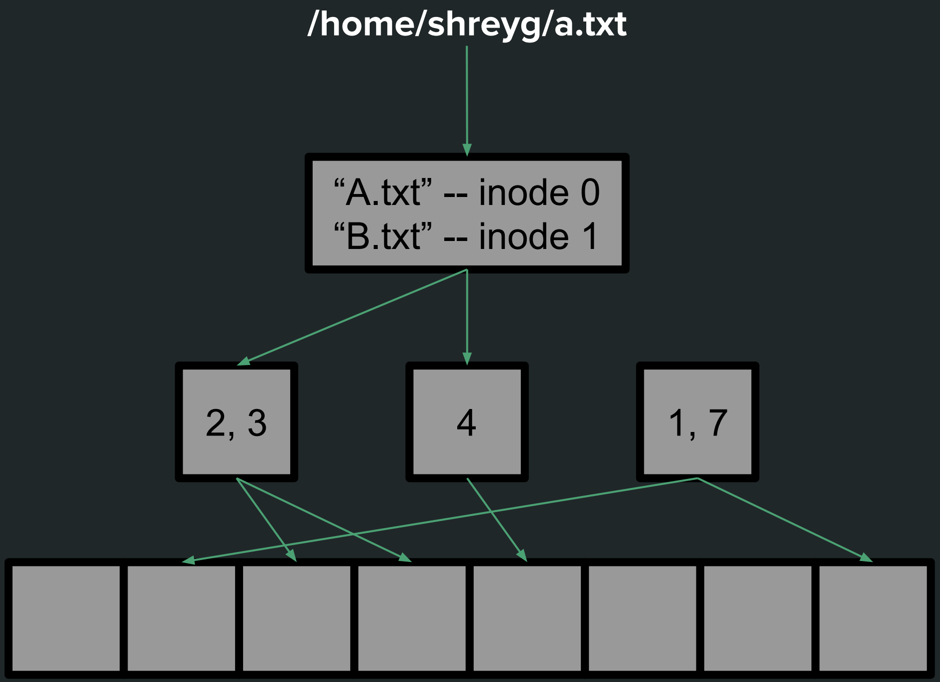


Pathname Layer

Directory Layer

Inode Layer

Block Layer



/home/shreyg/a.txt

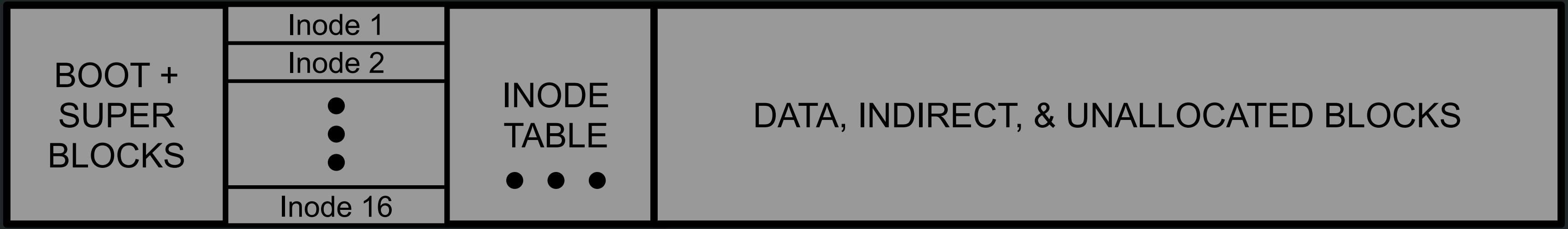
"A.txt" -- inode 0
"B.txt" -- inode 1

2, 3

4

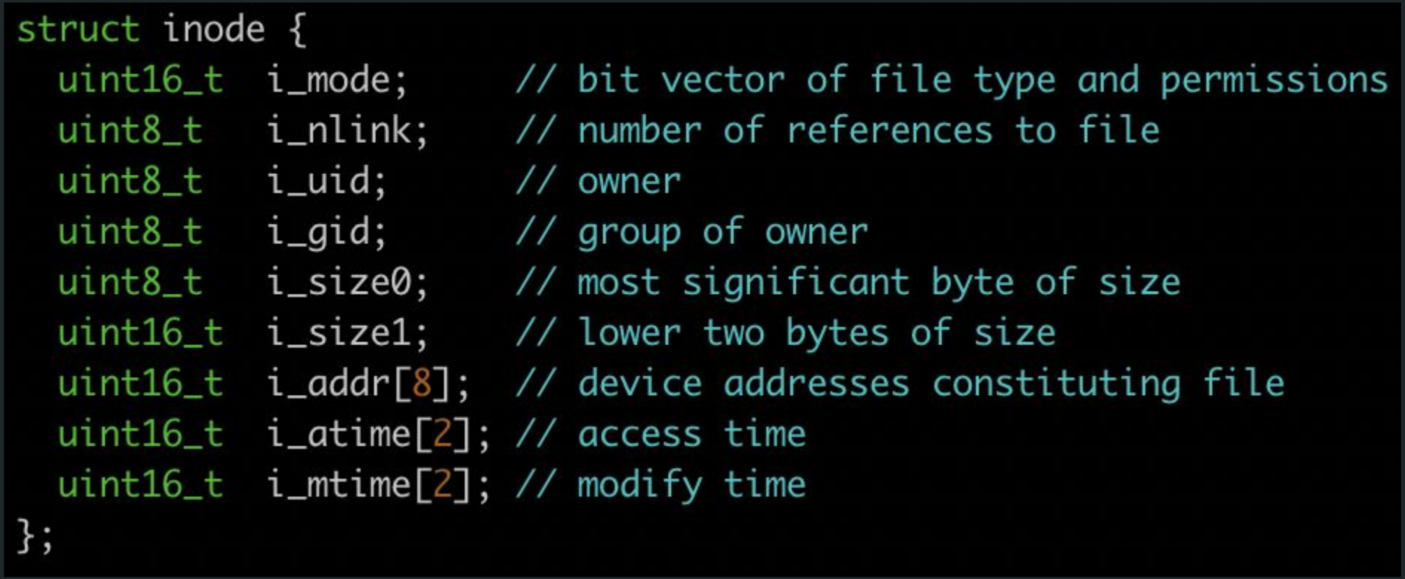
1, 7

Inodes & Files



- Super block contains info on the type and config of the filesystem
- Inodes contain all the metadata regarding a file/directory.
- Data blocks contain actual file data or directory entries
- Indirect blocks contain lists of block numbers to data/indirect blocks

What’s in an inode?

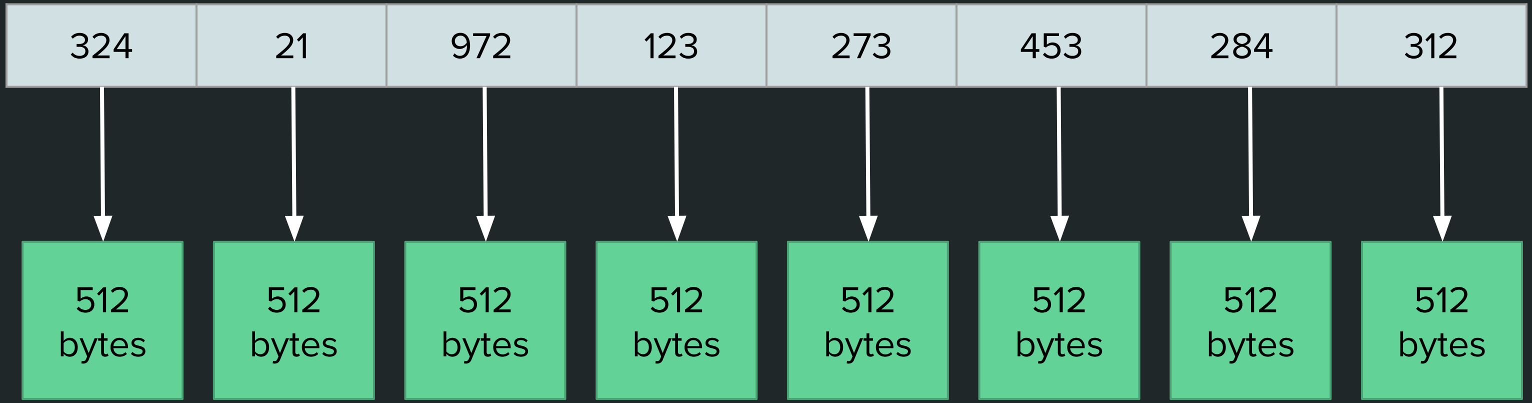


```
struct inode {
  uint16_t  i_mode;     // bit vector of file type and permissions
  uint8_t   i_nlink;    // number of references to file
  uint8_t   i_uid;      // owner
  uint8_t   i_gid;      // group of owner
  uint8_t   i_size0;    // most significant byte of size
  uint16_t  i_size1;    // lower two bytes of size
  uint16_t  i_addr[8];  // device addresses constituting file
  uint16_t  i_atime[2]; // access time
  uint16_t  i_mtime[2]; // modify time
};
```

(From ino.h, assign2)

Direct Addressing

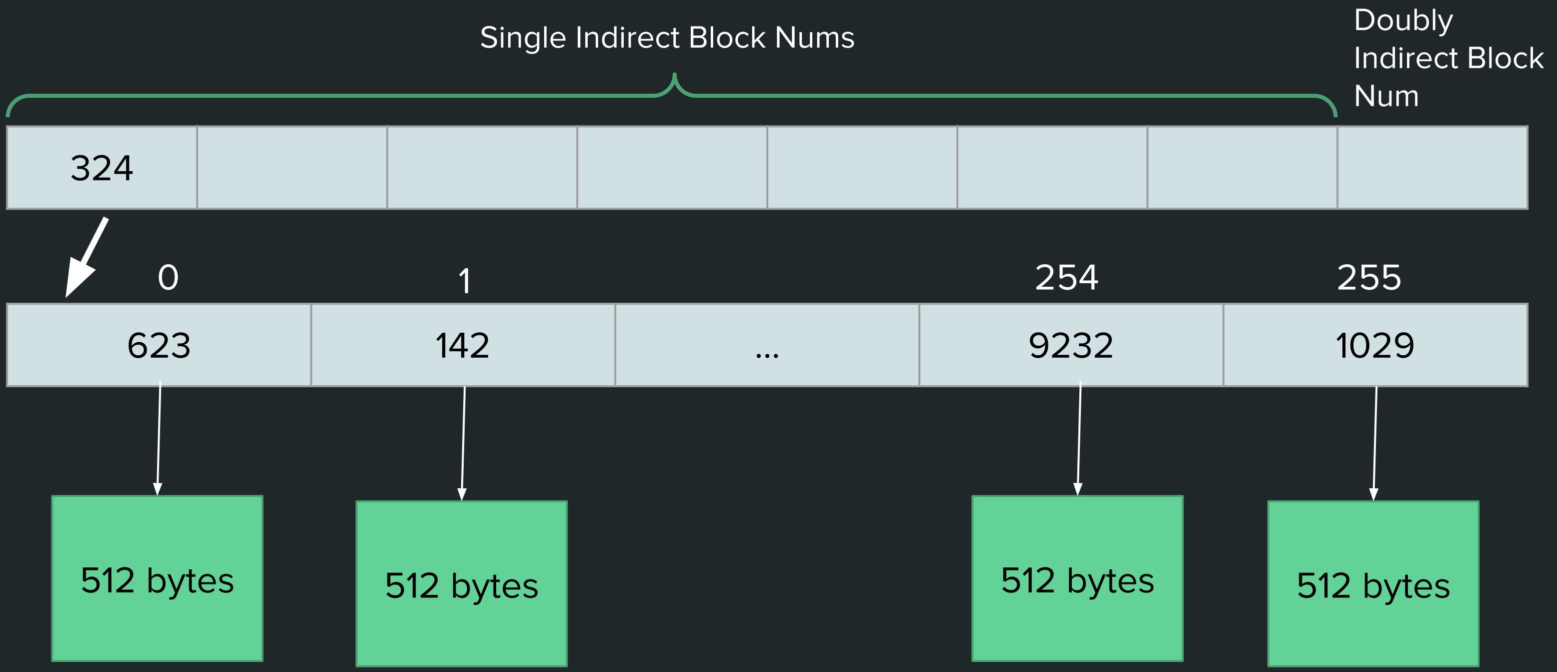
file size <= 8 * 512



324	21	972	123	273	453	284	312
512 bytes	512 bytes	512 bytes	512 bytes	512 bytes	512 bytes	512 bytes	512 bytes

Singly Indirect Addressing

$8 * 512 < \text{file size} <= 7 * 256 * 512$



Single Indirect Block Nums

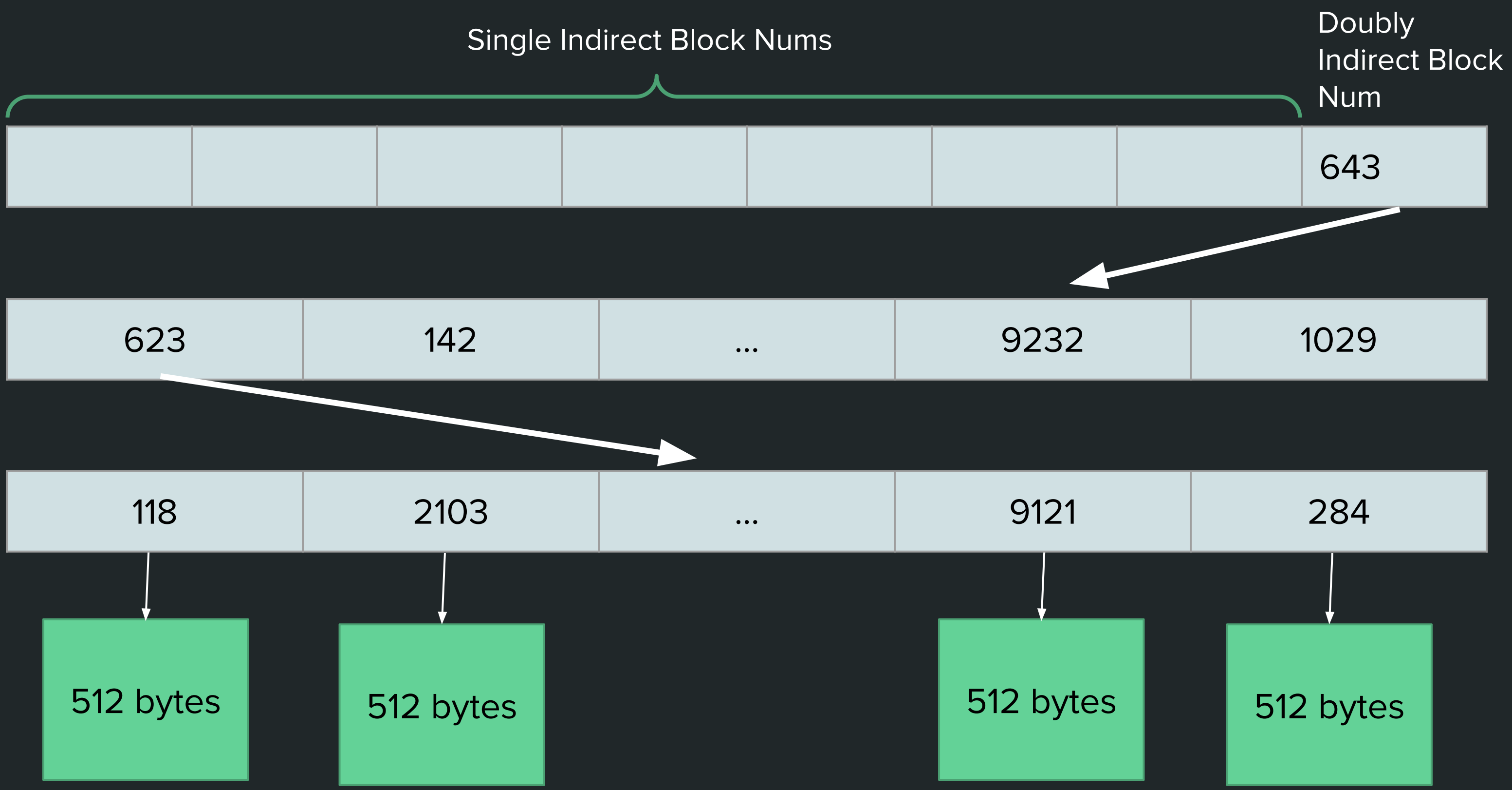
Doubly Indirect Block Num

324							

0	1		254	255
623	142	...	9232	1029

Doubly Indirect Addressing

$$7 * 256 * 512 < \text{file size} <=$$
$$7 * 256 * 512 + 256 * 256 * 512$$

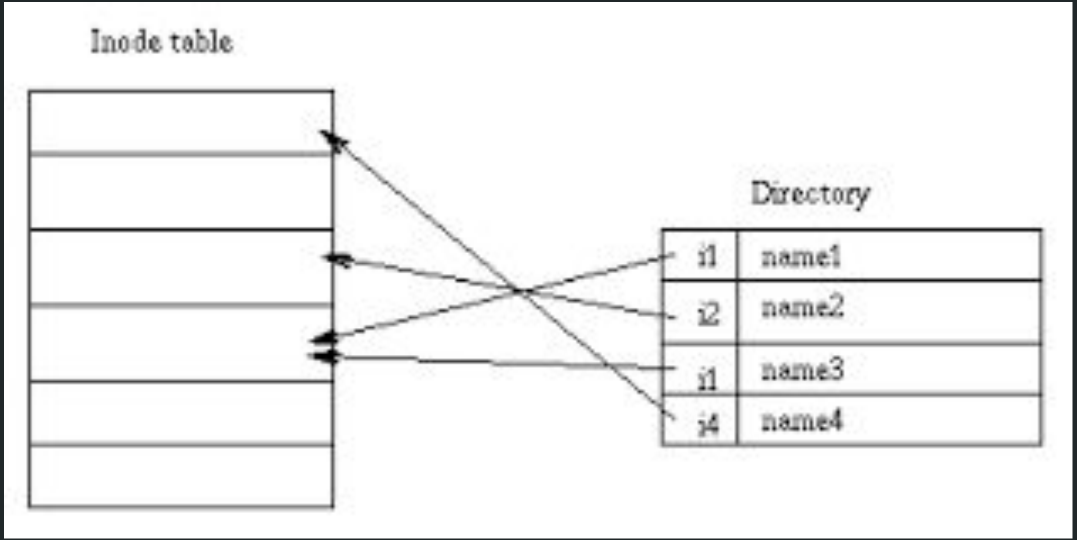


							643

623	142	...	9232	1029

118	2103	...	9121	284

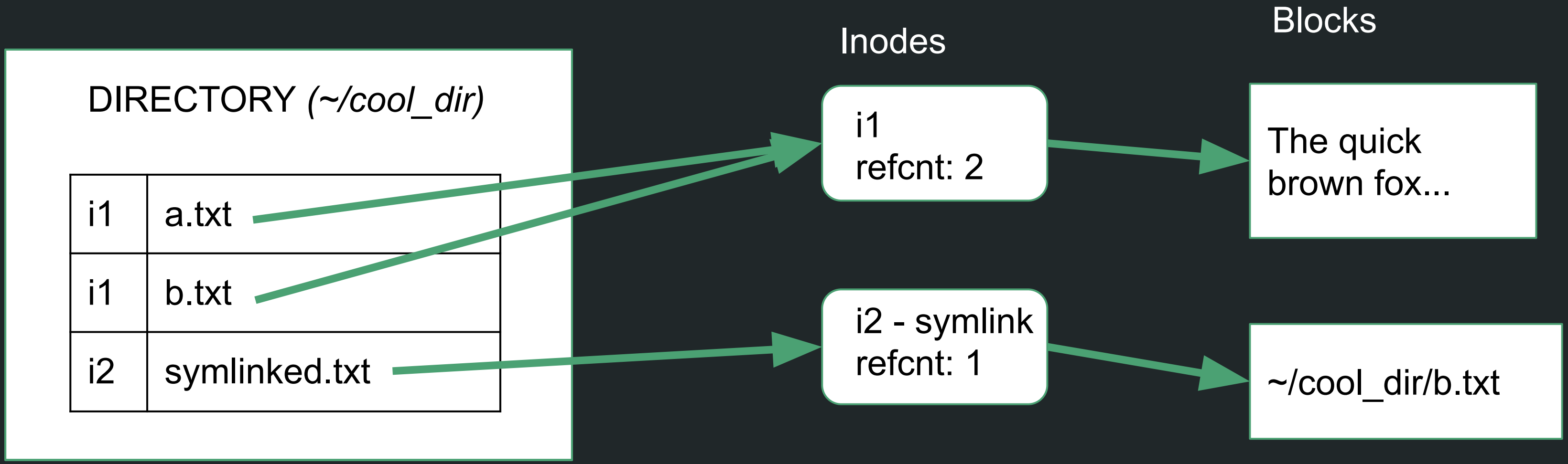
Directories



- Directories are just a special type of file
- Payload blocks consist of (inode block number, name) pairs

Links

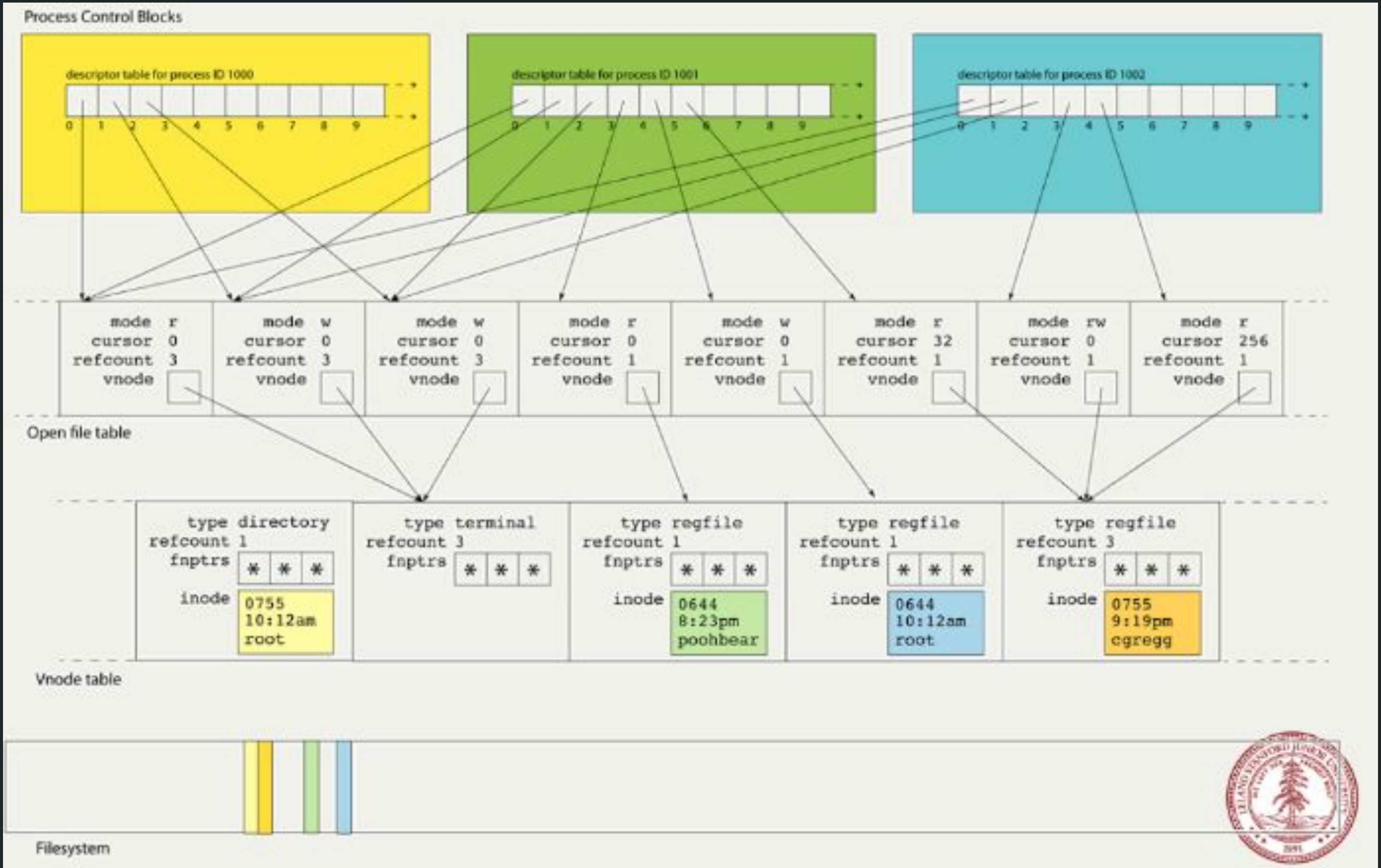
- Hard links vs. symbolic (soft) links
- All three of these links go to the same file!
- Can’t create hard link for directories (breaks pathnames, allows loops)



DIRECTORY *(~/cool_dir)*

i1	a.txt
i1	b.txt
i2	symlinked.txt

How the operating system manages files



Process Control Blocks

descriptor table for process ID 1000

descriptor table for process ID 1001

descriptor table for process ID 1002

Open file table

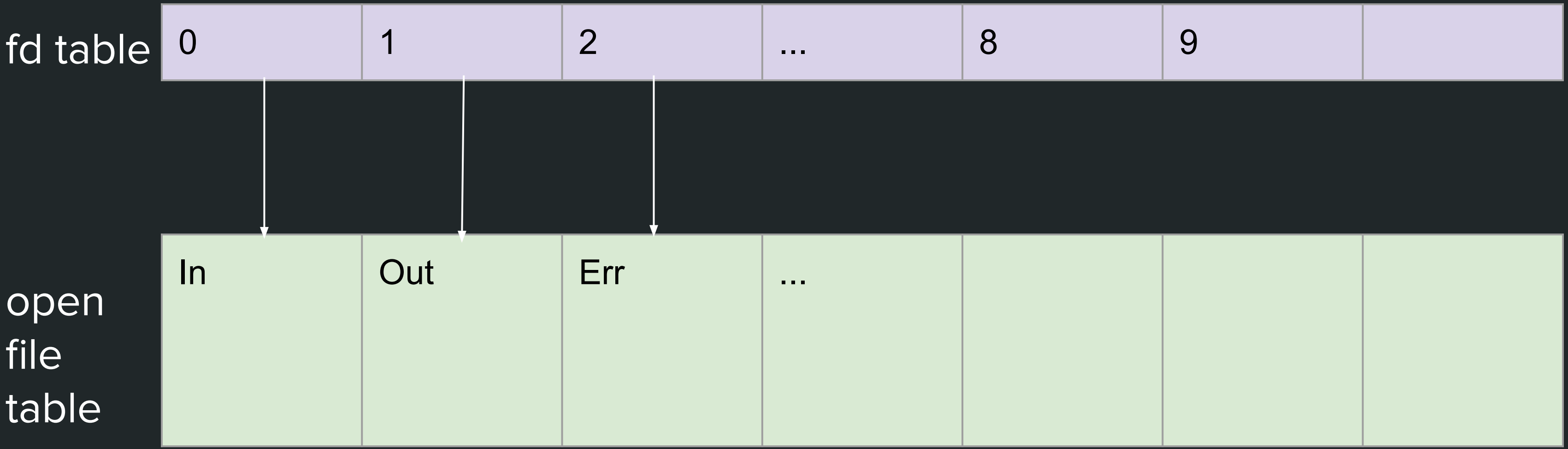
mode r	mode w	mode w	mode r	mode w	mode r	mode rw	mode r
cursor 0	cursor 0	cursor 0	cursor 0	cursor 0	cursor 32	cursor 0	cursor 256
refcount 3	refcount 3	refcount 3	refcount 1	refcount 1	refcount 1	refcount 1	refcount 1
vnode	vnode	vnode	vnode	vnode	vnode	vnode	vnode

Vnode table

type directory	type terminal	type regfile	type regfile	type regfile
refcount 1	refcount 3	refcount 1	refcount 1	refcount 3
fnptrs * * *	fnptrs * * *	fnptrs * * *	fnptrs * * *	fnptrs * * *
inode 0755 10:12am root		inode 0644 8:23pm poohbear	inode 0644 10:12am root	inode 0755 9:19pm cgregg

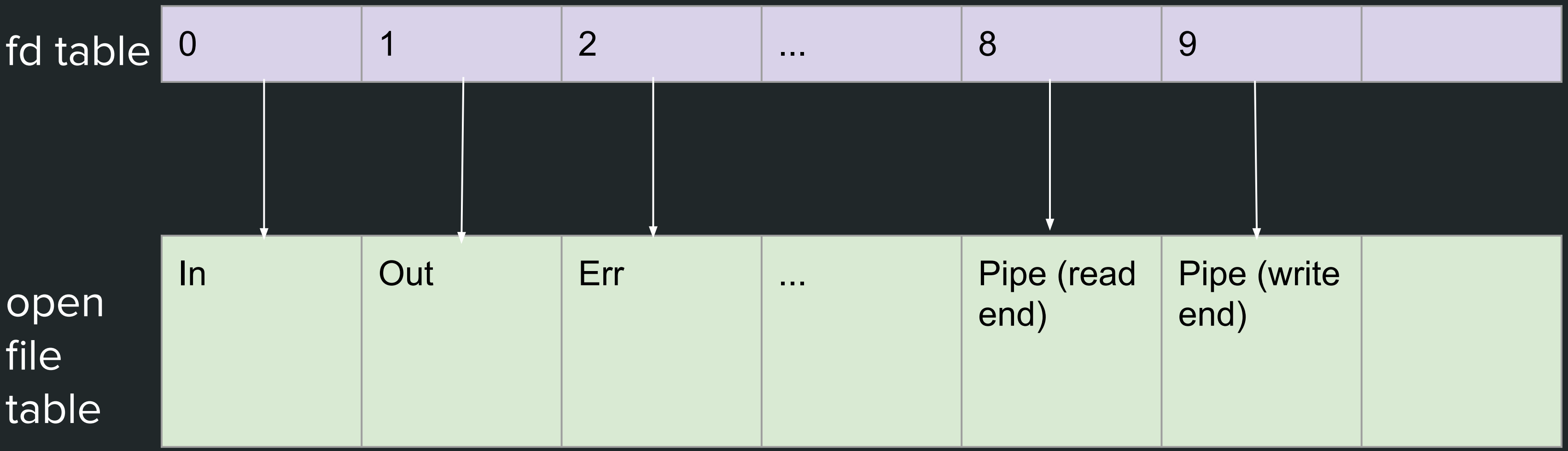
Filesystem

How System Calls Affect the File Tables



fd table	0	1	2	...	8	9	
open file table	In	Out	Err	...			

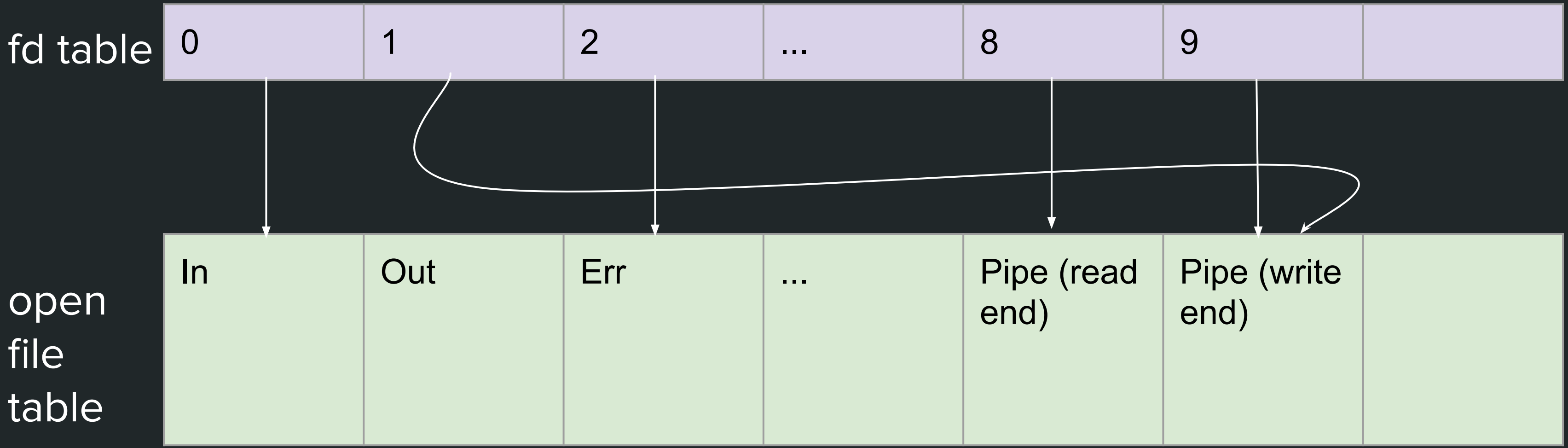
The Effect of `pipe` on the File Tables



fd table	0	1	2	...	8	9	
open file table	In	Out	Err	...	Pipe (read end)	Pipe (write end)	

```
int fds[2];
pipe(fds);
// fds[0] has 8, fds[1] has 9
// Q: How do we redirect STDOUT to the pipe?
```

The Effect of `dup2` on the File Tables

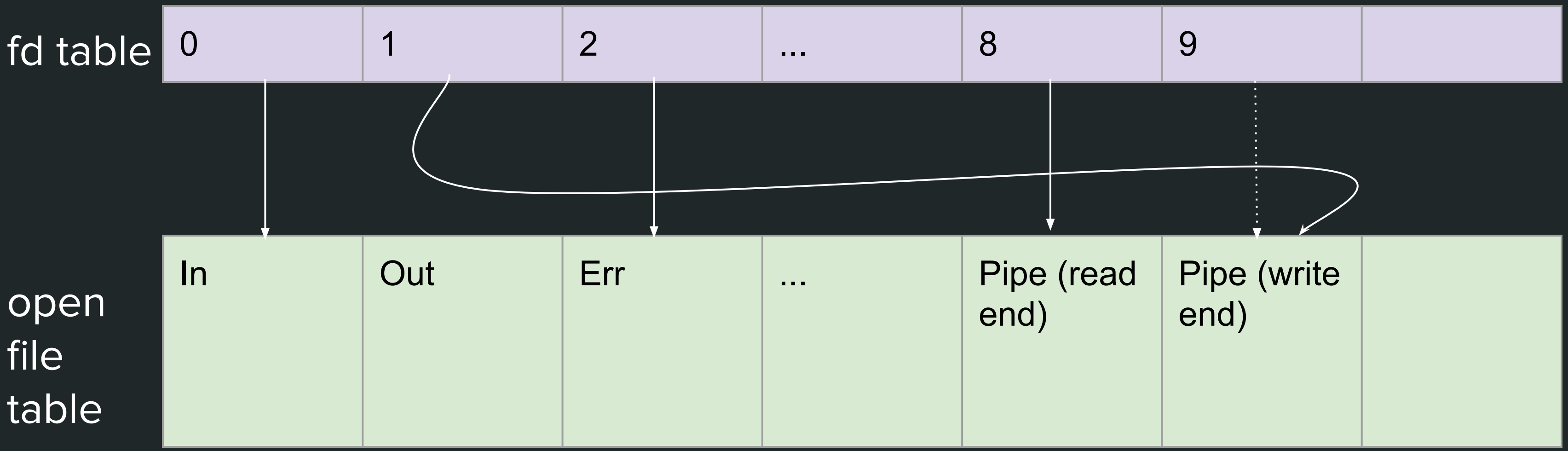


fd table	0	1	2	...	8	9	

open file table	In	Out	Err	...	Pipe (read end)	Pipe (write end)	

```
// int dup2(int oldfd, int newfd);
// have newfd point to what oldfd points to
dup2(fds[1], STDOUT_FILENO)
```

The Effect of `close` on the File Tables



fd table

0 1 2 ... 8 9

open file table

In

Out

Err

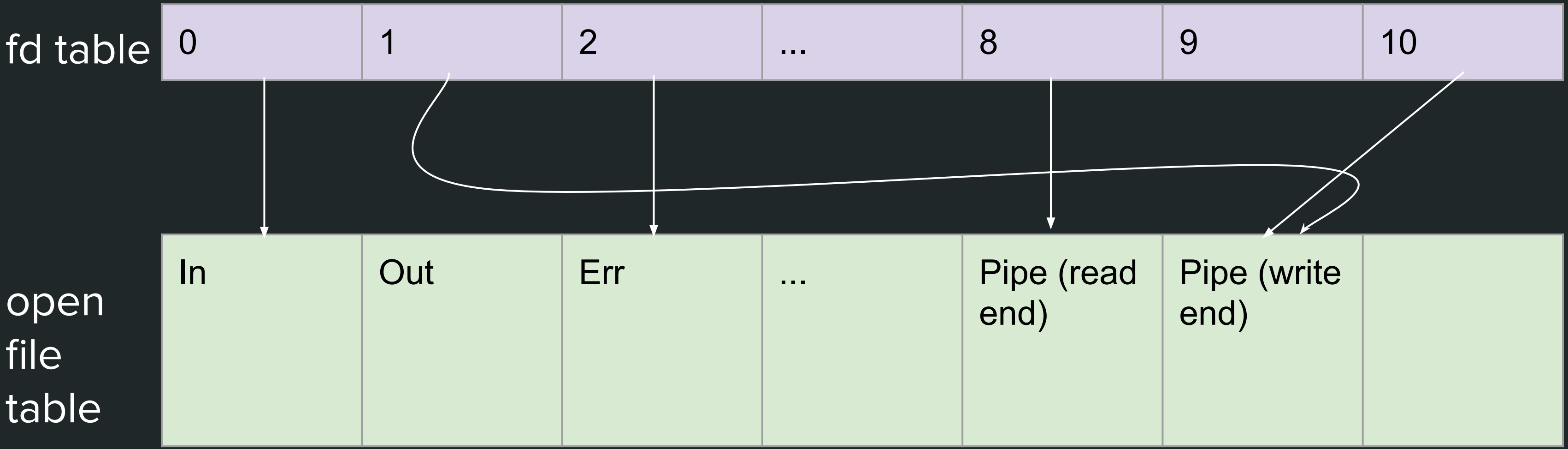
...

Pipe (read end)

Pipe (write end)

```
close(fds[1]);
// Q: What happens when we call
dup(STDOUT_FILENO)?
```

The Effect of `dup` on the File Tables



fd table

0 1 2 ... 8 9 10

open file table

In

Out

Err

...

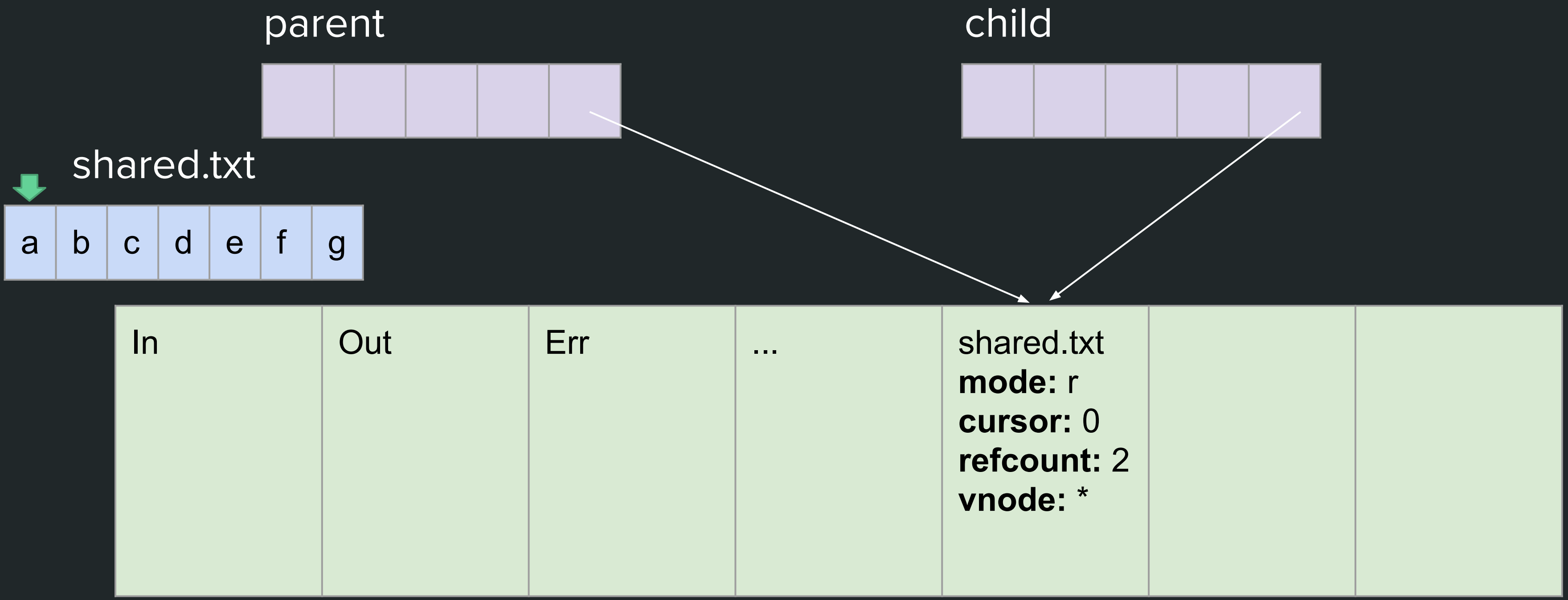
Pipe (read end)

Pipe (write end)

```
// Returns a new fd that points to what the
// fd passed in is pointing to.
dup(STDOUT_FILENO);
```

The Open File Table is Shared Across Processes

Q: Suppose the parent forks, then reads 4 bytes. What is the next byte the child would read from shared.txt?



parent

child

shared.txt

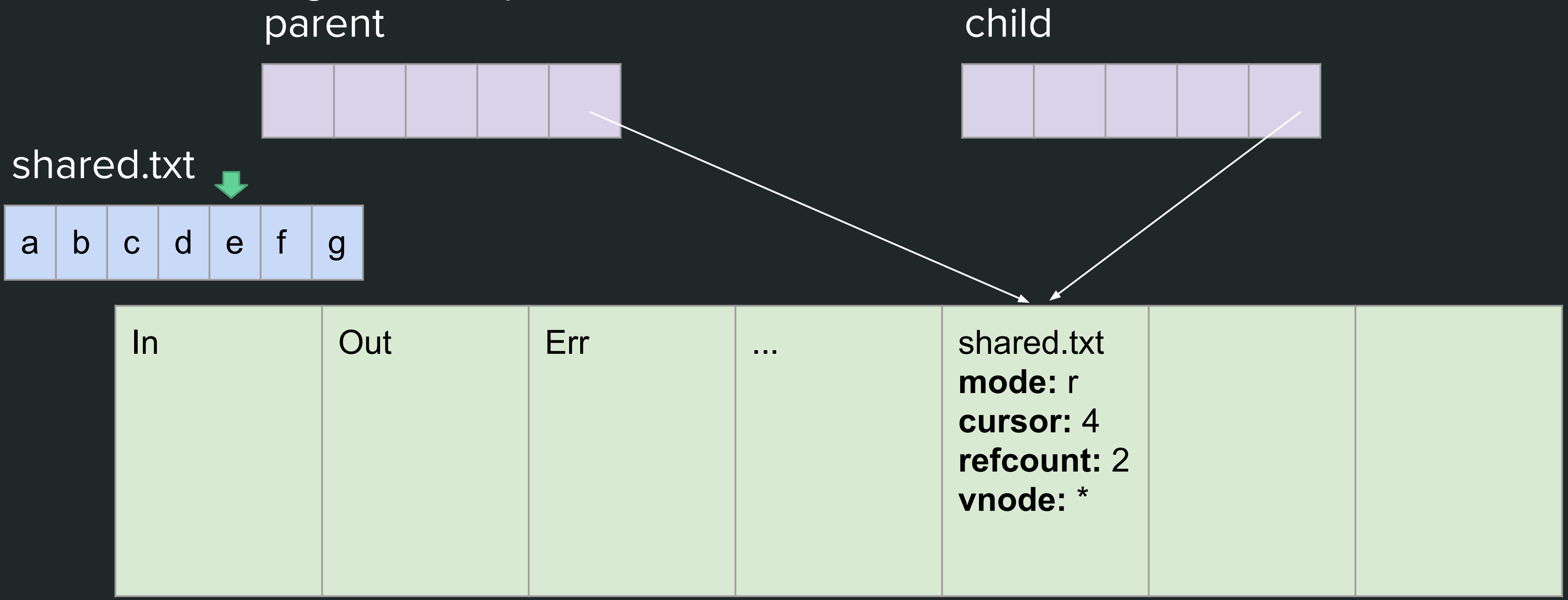
a	b	c	d	e	f	g

In	Out	Err	...	shared.txt **mode:** r **cursor:** 0 **refcount:** 2 **vnode:** *		

The Open File Table is Shared Across Processes

A: ‘e’

Q: What changes in this picture if the child closes shared.txt?



parent

child

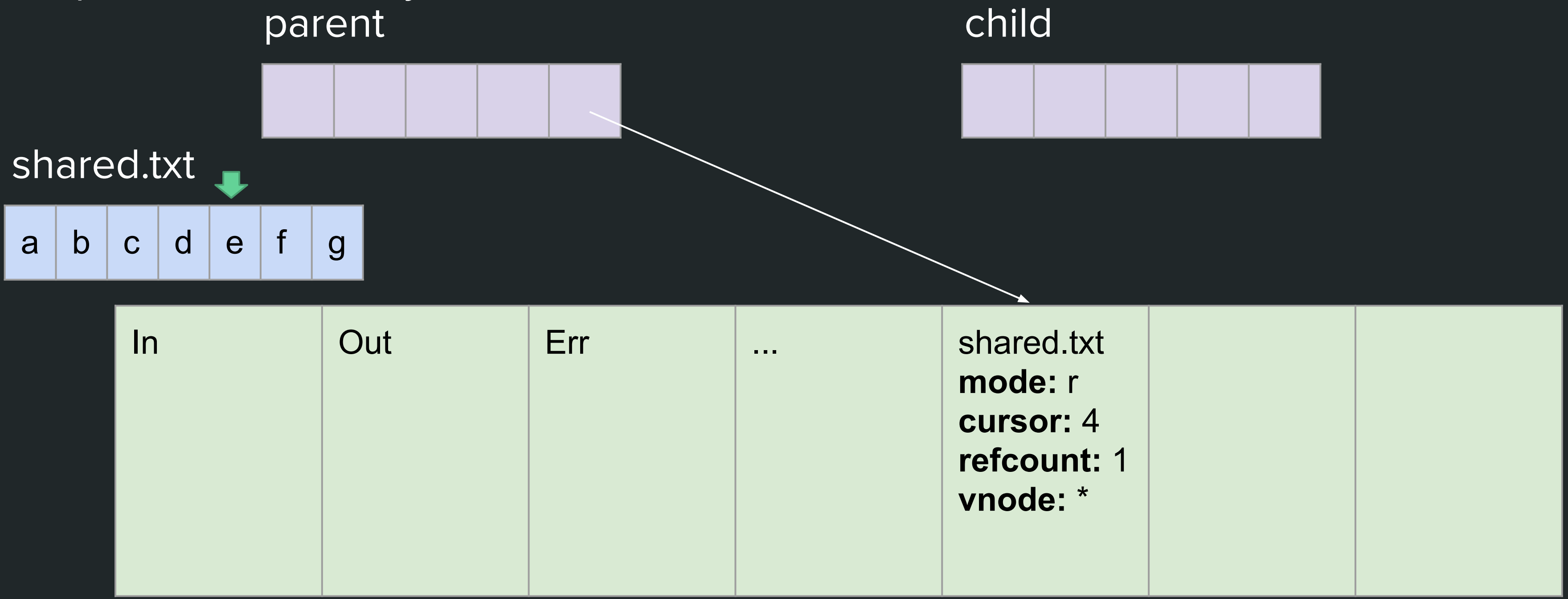
shared.txt

a	b	c	d	e	f	g

In	Out	Err	...	shared.txt **mode:** r **cursor:** 4 **refcount:** 2 **vnode:** *		

The Open File Table is Shared Across Processes

A: The refcount drops to 1. The child no longer has an fd that points to the open file table entry.

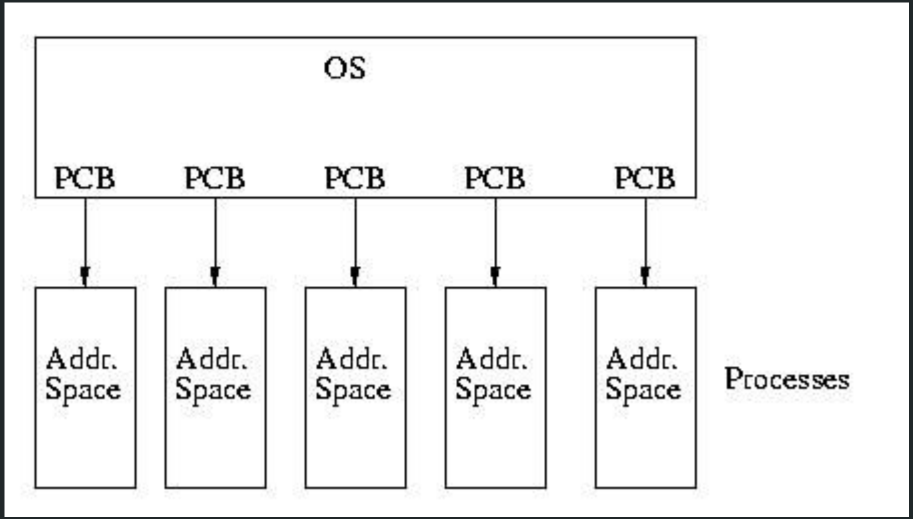


Part 2.1: Processes

🍴

Processes

- Unique PID
- Not-necessarily unique PGID
- At least one thread
- Its own file descriptor table
- Its own Virtual Memory space



Virtualization

- DREAM: Every process thinks it has its own address space

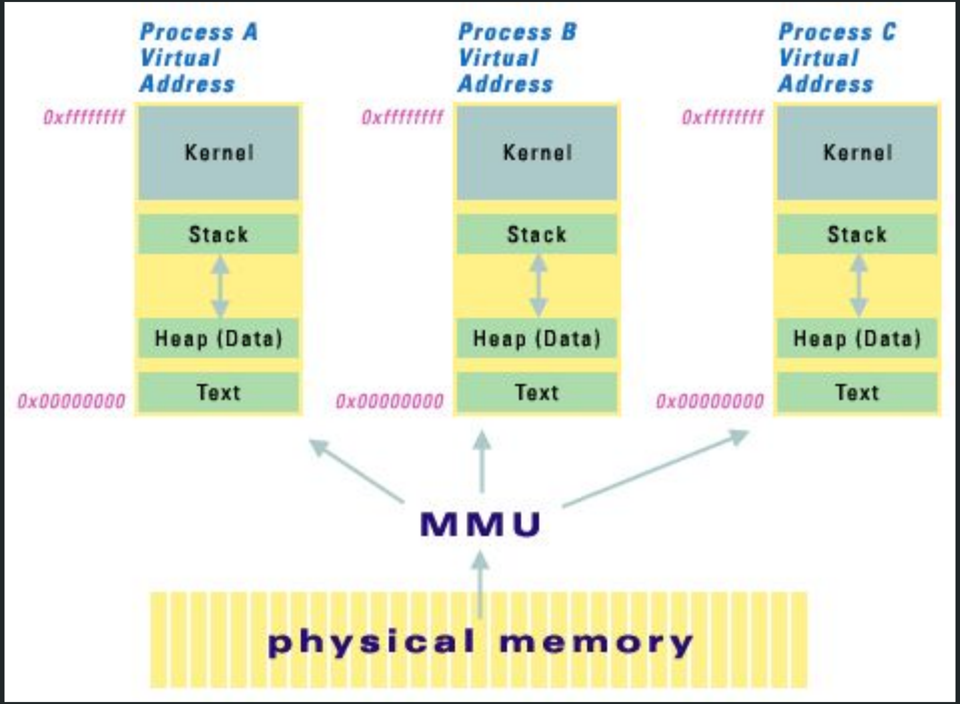
- i.e., each process thinks it has sole access to the addresses ranging from 0x00000000 to 0xffffffff

When you think you have your own private address space but the next process also has the same private address space

Virtual Memory

- REALITY: Process address space isn't really where data lives

- Translation facilitated by kernel on every "dereference" of an address.



Questions

Suppose P1 and P2 are separate process running /usr/bin/ls. Which of the following are possible?

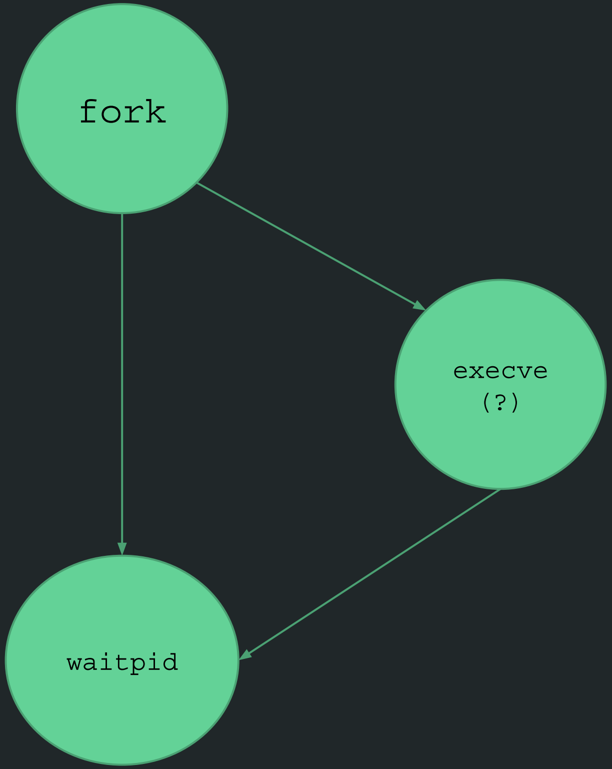
- Both P1 and P2 call open("foo.txt") and in both the returned fd was 5.
- Both P1 and P2 read 7 chars from the returned fd and read the same thing.
- Both P1 and P2 store variable foo at virtual address 0xdeadbeef.
- Both P1 and P2 store variable foo at physical address 0xdeadbeef.

Part 2.2: Multiprocessing

Basic Paradigm

```
int fork_child(char **argv) {
    pid_t pid = fork();
    if (pid == 0) {
        execvp(argv[0], argv);
        exit(0);
    }

    int status;
    waitpid(pid, &status, 0);
    return WEXITSTATUS(status);
}
```



fork()

- Duplicates almost everything: all of virtual memory, file descriptor table, signal handlers, signal mask, etc (not pending signals)
- Return value: child pid for parent, 0 for child
- I lied above: virtual memory is copied lazily i.e. Copy on Write (CoW)

execvp() (and friends)

- Replaces memory-related things: all of virtual memory, signal handlers, etc (not file descriptor table, pending signals, signal mask which are managed by the kernel)
- Return value: doesn’t return unless error (e.g. unknown program)

waitpid()

- Block until change in child processes’ running state (by default: only termination, but can detect stopped and signaled)
- Return value: -1 (error), 0 (WNOHANG specified and no children waiting), pid (of child who changed state)
- If you don’t reap, you’ll have zombie children!

waitpid()

	Terminated	Signaled	Stopped	Continued
Request Detection	<default>	<default>	WUNTRACED	WCONTINUED
Check State	WIFEXITED	WIFSIGNALED	WIFSTOPPED	WIFCONTINUED
Additional Info	WEXITSTATUS	WTERMSIG	WSTOPSIG	<none>

- First row are flags for waitpid’s third argument, last two rows are macros that take in the status.
- Also - WNOHANG, which makes waitpid return early if nothing has already changed.

Questions

- The following is from a commit Linus Torvalds made to Linux last Saturday.

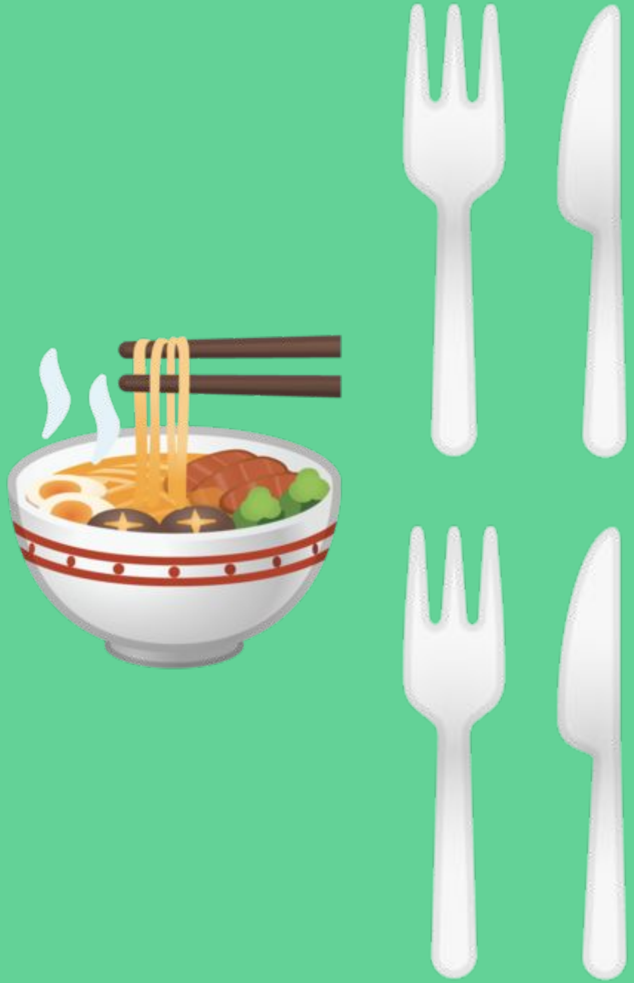
  ```
  fork(); fork(); fork(); fork(); fork(); fork();
  ```

  How many processes does it create?

- Review the following code. What possible issues could occur? (3-4 issues)

  ```
  int main(int argc, char *argv[]) {
      pid_t pid = fork();
      if (pid == 0) {
          execvp(argv[0], argv);
      }
      int* status;
      waitpid(pid, status, WNOHANG);
      assert(WIFEXITED(*status));
      return WEXITSTATUS(*status);
  }
  ```

Part 2.3: Multiprocess Communication



How do processes communicate?

What we’ve seen in CS 110:

- pipes
- mmap
- signals

Pipes

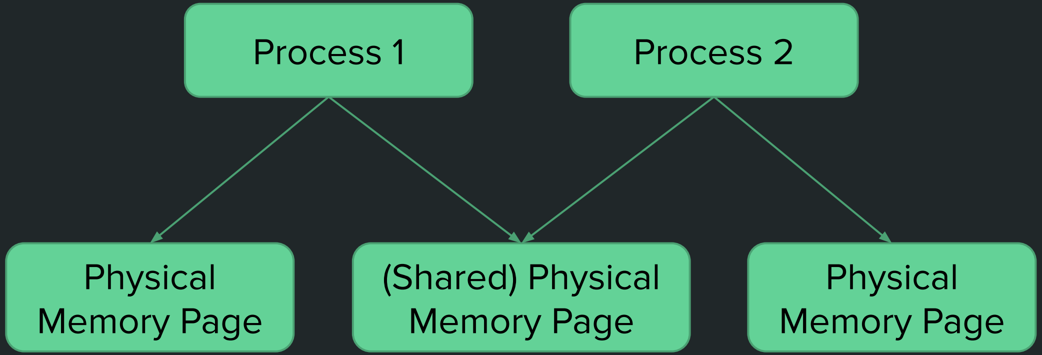
- What happens in the following code?

```
static void writeOutput(const char *array[]) {
   printf("Writing output to file named \"%s\".\n", array[0]);
   int outfile = open(array[0], O_WRONLY | O_CREAT | O_TRUNC, 0644);
   dup2(outfile, STDOUT_FILENO);
   close(outfile);
   if (fork() > 0) return;
   execvp(array[1], array+1);
   exit(0);
}

int main(int argc, char *argv[]) {
    char *array1[] = {“cal.txt”, “cal”, NULL};
    char *array2[] = {“date.txt”, “date”, NULL};
    writeOutput(array1); waitpid(-1, NULL, 0);
    writeOutput(array2); waitpid(-1, NULL, 0);
    return 0;
}
```

mmap

- Like malloc, but is able to allocate memory that is shared between processes (e.g. after fork()).



Part 3: Signals

What are signals?

Asynchronous notifications sent to a process by the kernel or another process.

- Created via kill() or raise() (what’s the difference?).
- Handled by signal handlers registered via signal().
- Can be blocked via sigprocmask().
- Can be awaited (e.g. sleep until signal) via sigsuspend().
- Examples: (why are these two lines separated?)
    - SIGINT, SIGTSTP, SIGCONT, SIGCHLD
    - SIGSTOP, SIGKILL

Signal Delivery

- If a process blocks a signal, delivery of the signal is delayed until it’s unblocked.
- If a process is not on CPU during signal delivery, it is delivered once it is.
- Signals aren’t queued!
    - The kernel tracks only what signals are delivered, not how many
- Signal handlers block the signal they are handling (e.g. can be interrupted by other signals).

Signal Handlers

- A function _you_ write that can be registered to run upon signal delivery.
    - `sighandler_t signal(int signum, sighandler_t handler);`
- Since signals are async, the signal handler might be run at any time! => beware of race conditions
- Avoid race conditions by blocking signals appropriately
    - `sigset_t set;`
    - `int sigemptyset(sigset_t *set);`
    - `int sigaddset(sigset_t *set, int signum);`
    - `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);`

Signal Blocking (find the bugs)

```
static int sum = 0;
static int children = 0;
int main(int argc, char *argv[]) {
    for (int i = 0; i < 5; i++) {
        pid_t pid = fork();
        if (pid == 0) {
            // do nonsense work
            exit(i);
        }
        children++;
    }
    signal(SIGCHLD, &handle);
    while (children > 0) {
        // busy wait :(
    }
    cout << “sum: “ << sum << endl;
    return 0;
}
```

```
static void handle(int signal) {
    int status;
    waitpid(-1, &status, 0);
    assert(WIFEXITED(status));
    sum += WEXITSTATUS(status);
    children--;
    cout << “One child exited!” << endl;
}
```

Signal Blocking (fixed!)

```
static int sum = 0;
static int children = 0;
int main(int argc, char *argv[]) {
    signal(SIGCHLD, &handle);
    for (int i = 0; i < 5; i++) {
        pid_t pid = fork();
        if (pid == 0) {
            // do nonsense work
            exit(i);
        }
        sigset_t set;
        sigemptyset(&set);
        sigaddset(&set, SIGCHLD);
        sigprocmask(SIG_BLOCK, &set, NULL);
        children++;
        sigprocmask(SIG_UNBLOCK, &set, NULL);
    }
    while (children > 0) {
        // busy wait :(
    }
    cout << "sum: " << sum << endl;
    return 0;
}
```

```
static void handle(int signal) {
    int status;
    while (true) {
        if (waitpid(-1, &status, WNOHANG) <= 0) {
            break;
        }
        assert(WIFEXITED(status));
        sum += WEXITSTATUS(status);
        children--;
        cout << "One child exited!" << endl;
    }
}
```

sigsuspend()

sigsuspend(&mask):

```
//ATOMICALLY:
   sigprocmask(SIG_SETMASK, &mask, &old);
   pause(); // wait for signal to wake us up
   sigprocmask(SIG_SETMASK, &old, NULL);
```

Another `kill-puzzle`!

```
static pid_t pid;
static int counter = 0;

static void parentHandler(int unused) {
  counter += 2;
  printf("counter = %d\n", counter);
}

static void childHandler(int unused) {
  counter += 1;
  printf("counter = %d\n", counter);
  kill(getppid(), SIGUSR1);
}
```

1. Can this program DEADLOCK?

BONUS: How many outputs are there?

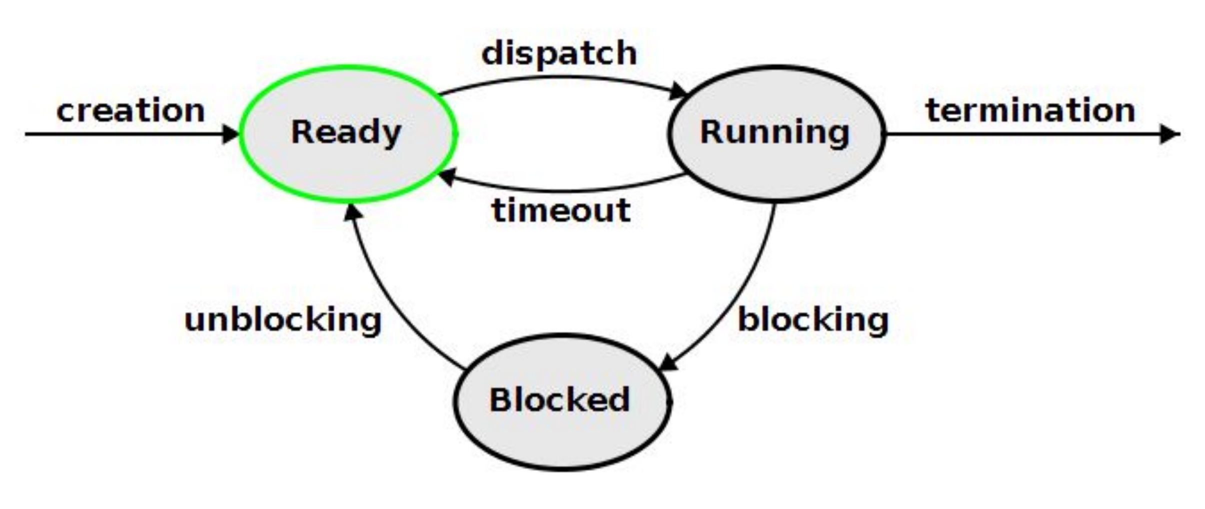
```
int main(int argc, char *argv[]) {
  signal(SIGUSR1, parentHandler);
  if ((pid = fork()) == 0) {
    signal(SIGUSR1, childHandler);
    sigset_t mask; sigemptyset(&mask);
    sigsuspend(&mask);
    return 0;
  }

  kill(pid, SIGUSR1);
  waitpid(pid, NULL, 0);
  counter += 3;
  printf("counter = %d\n", counter);
  return 0;
}
```

Part 4: Scheduling

Scheduling

(not emphasized on midterm)



Scheduling

- Process control block (PCB) - struct representing a process’ state
  - What code it last was executing, register values, PID, etc

- PCBs put into one of:
  - blocked queue, ready/runnable queue, running queue

What causes a process to move from one queue to another?

Part 5: Threads

What are threads?

- A thread is an independent execution sequence within a single process.
- Threads share global parts of a virtual address space (text, data, heap) but have their own stack + registers.
- Threads are multiplexed onto processors
- Threads are often called lightweight processes.

C++ Thread Syntax

```
#include <iostream>
#include <thread>

using namespace std;

int main()
{
    thread t([](a, b){
        cout << a + b << endl;
    }, 3, 3);
    t.join();
    return 0;
}
```

Thread constructor accepts a function and args. Processor schedules the thread to run the new function (i.e. adds thread to ready queue).

Parent thread waits for thread to finish executing.

Race Conditions with Threads

```
static float balance = 100.0;

void withdraw(float money) {
     if (money <= balance)
          balance -= money;
}

int main()
{
    thread t2(withdraw, 100);
    thread t2(withdraw, 100);
    t1.join();
    t2.join();
    return 0;
}
```

Can the balance ever be negative?

What are mutexes?

- A mutex allows you to control access to a critical section of code.
- It’s like a key: if you have it, you can enter, otherwise you must wait till someone else gives you the key.
- When a thread encounters a mutex:
  - If it’s unlocked, lock the mutex and continue.
  - If it’s locked, block until it’s unlocked.

Race Conditions Fixed with Mutex

```
static float balance = 100.0;

void withdraw(float money) {
     if (money <= balance)
          balance -= money;
}

int main()
{
    thread t1(withdraw, 100);
    thread t2(withdraw, 100);
    t1.join();
    t2.join();
    return 0;
}
```

Where should we put mutexes?

Race Conditions Fixed with Mutex

```
static float balance = 100.0;
static mutex lock;

void withdraw(float money) {
     lock.lock();
     if (money <= balance)
          balance -= money;
     lock.unlock();
}

int main()
{
    thread t1(withdraw, 100);
    thread t2(withdraw, 100);
    t1.join();
    t2.join();
    return 0;
}
```