CS 110 Summer 2018 Midterm Review Session

Matthew Katzman

Thanks to Ryan Eberhardt
Kristine Guo
Grace Hong
Hemanth Kini for some slide materials.

Exam Time

Monday, July 23

7PM-9PM

Hewlett 201

Study Resources

Be sure to look over:

- Assignments and Class Notes
- Labs and Handouts
- Practice Midterms

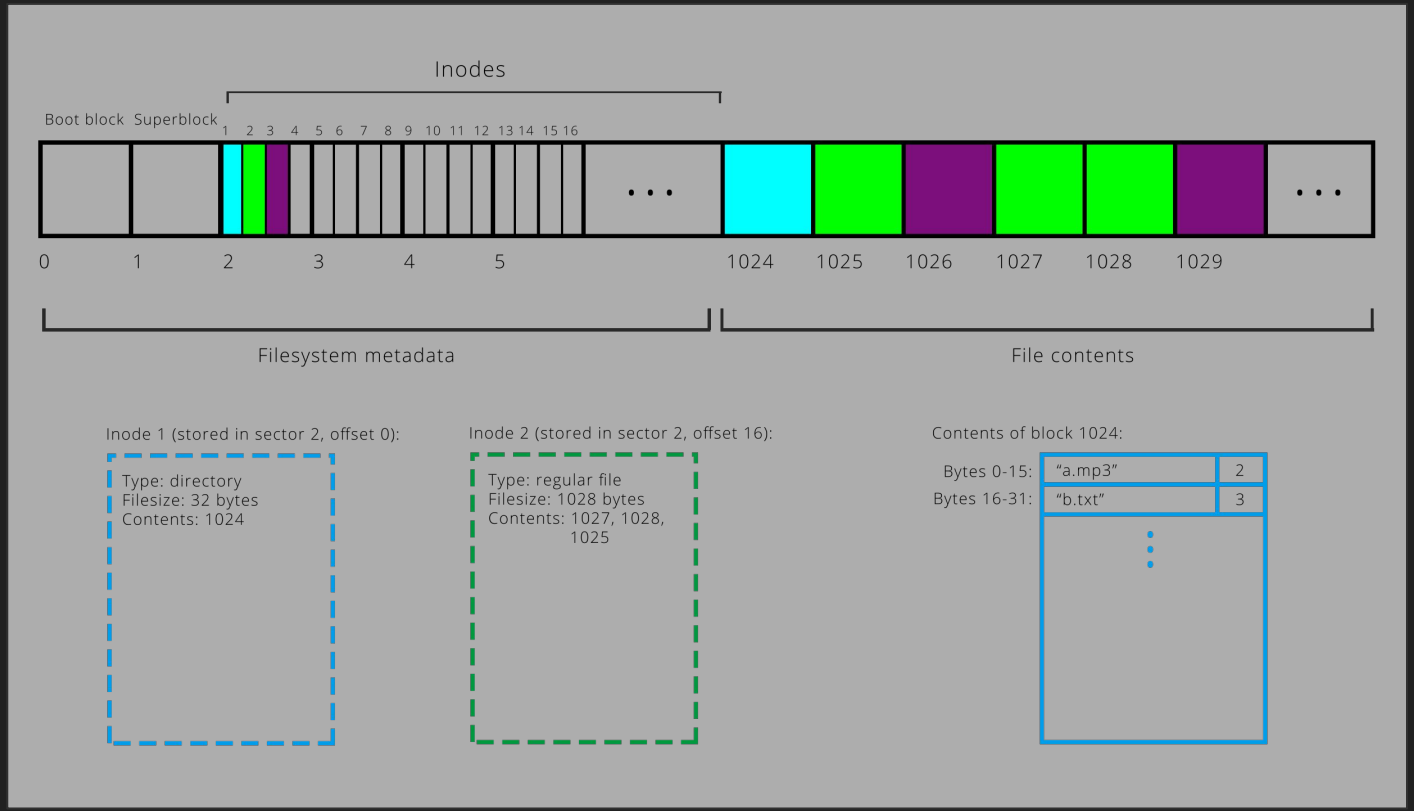
FILESYSTEMS

See assign1

The Important Types

- The Inode
- The File
- The Directory
- The Link

The Inode/FIle Layers



Directories

- A DIRECTORY IS A FILE
- Stores some important information, but mostly made up of dirEnt's (directory entries)
- Each of these consists of a filename and an inumber.

Links (two types)

HARD LINKS:

- The SAME as directory entries
- Map a filename to an inumber
- Increase the file's reference count

SYMBOLIC (soft) LINKS:

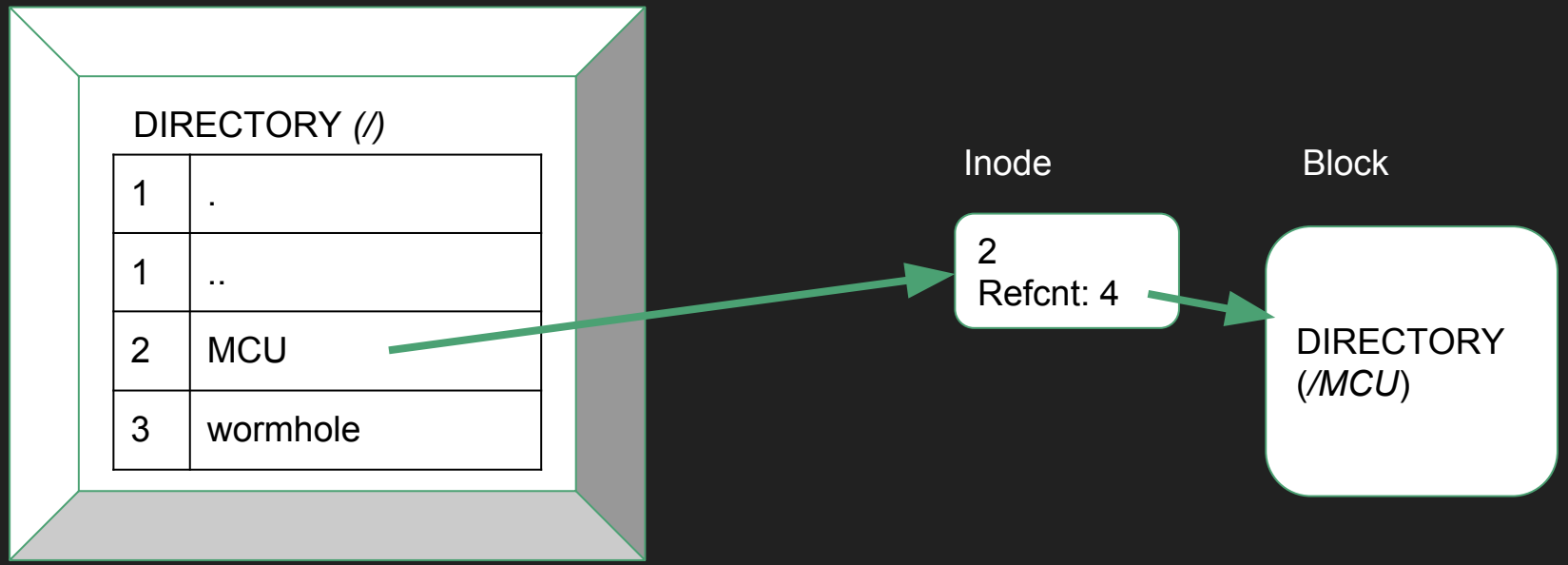
- Different type of file
- File contents include path to file
- Does not increase reference count

Where’s Hulk?

Hulk is located at /MCU/space/Sakaar/arena/Hulk.smash. How can we find him?

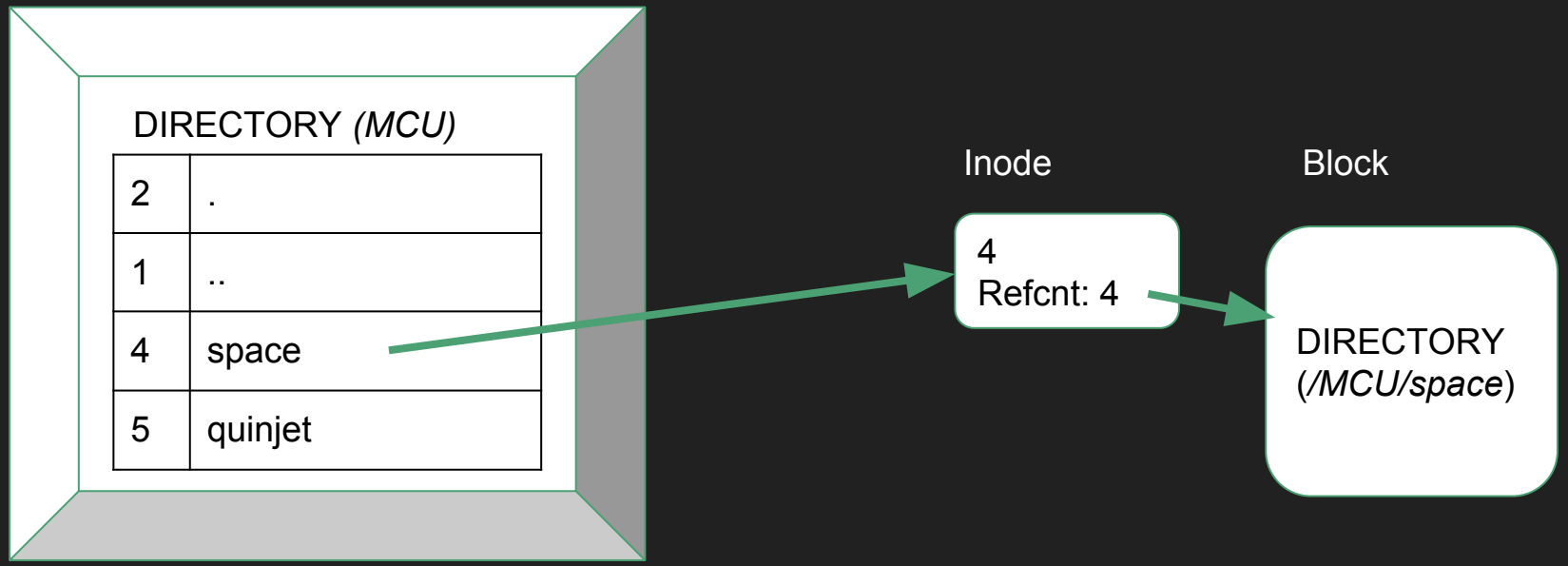
Where’s Hulk?

Hulk is located at /MCU/space/Sakaar/arena/Hulk.smash. How can we find him?



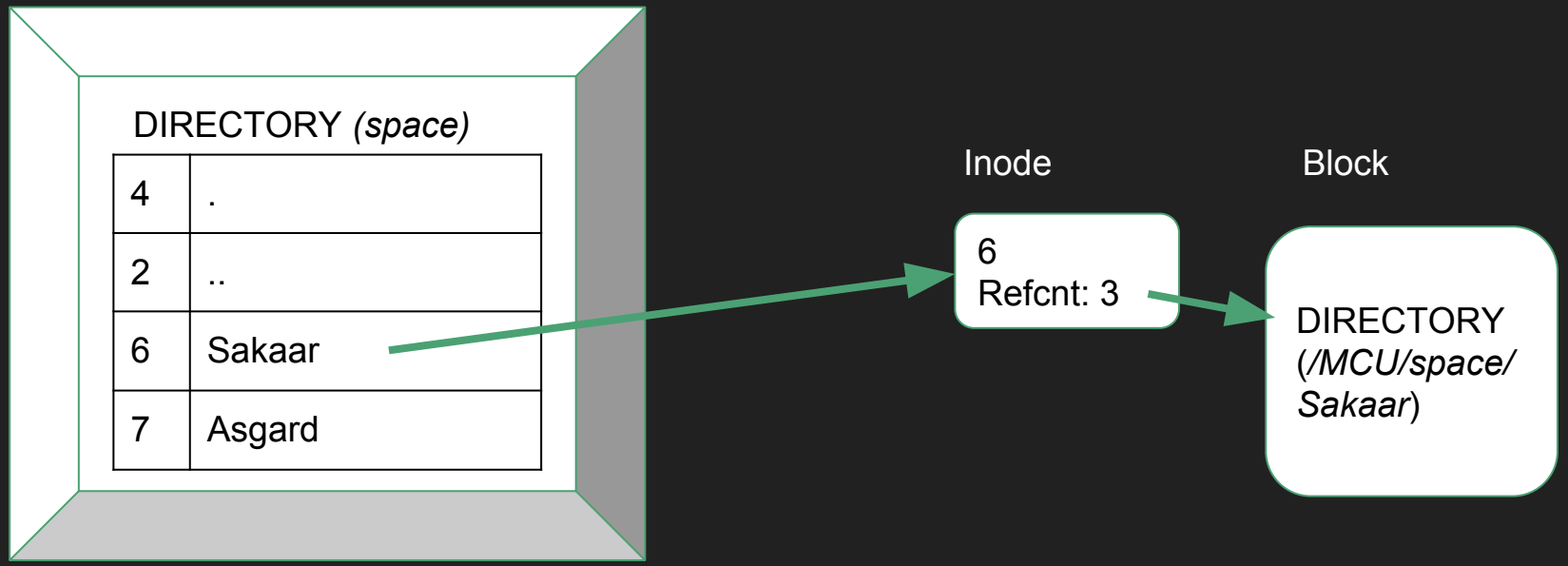
Where’s Hulk?

Hulk is located at /MCU/space/Sakaar/arena/Hulk.smash. How can we find him?



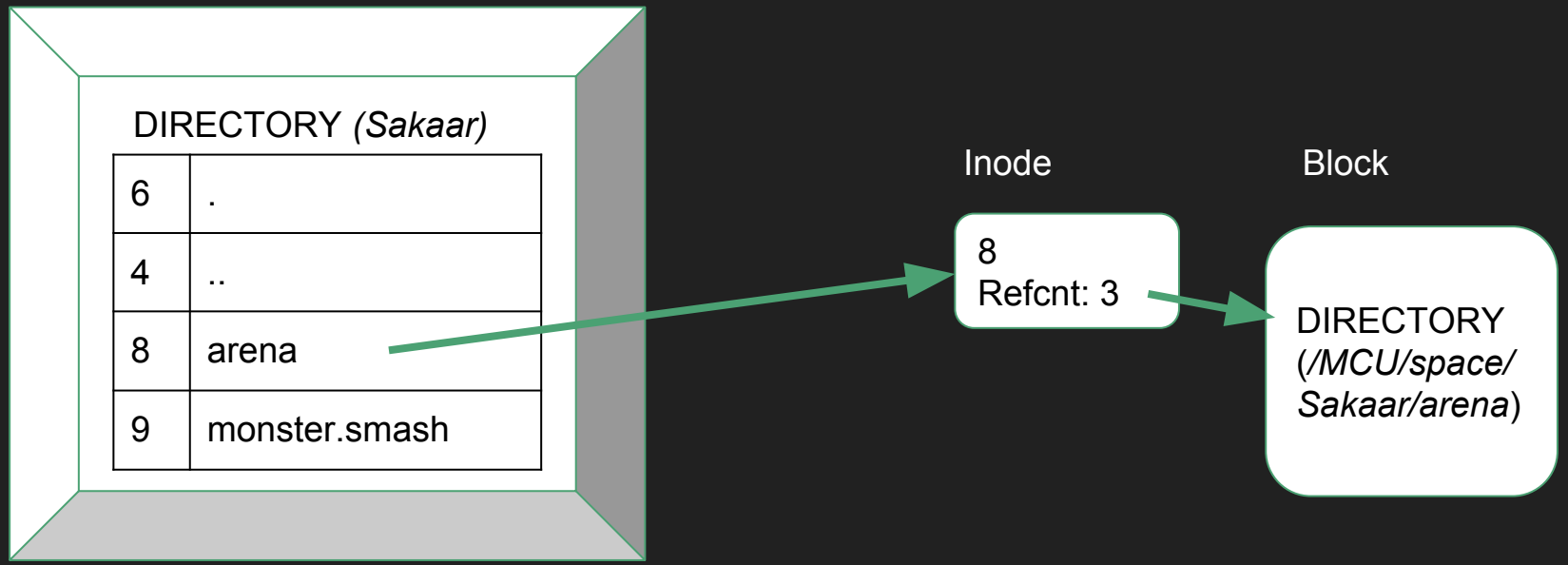
Where’s Hulk?

Hulk is located at /MCU/space/Sakaar/arena/Hulk.smash. How can we find him?



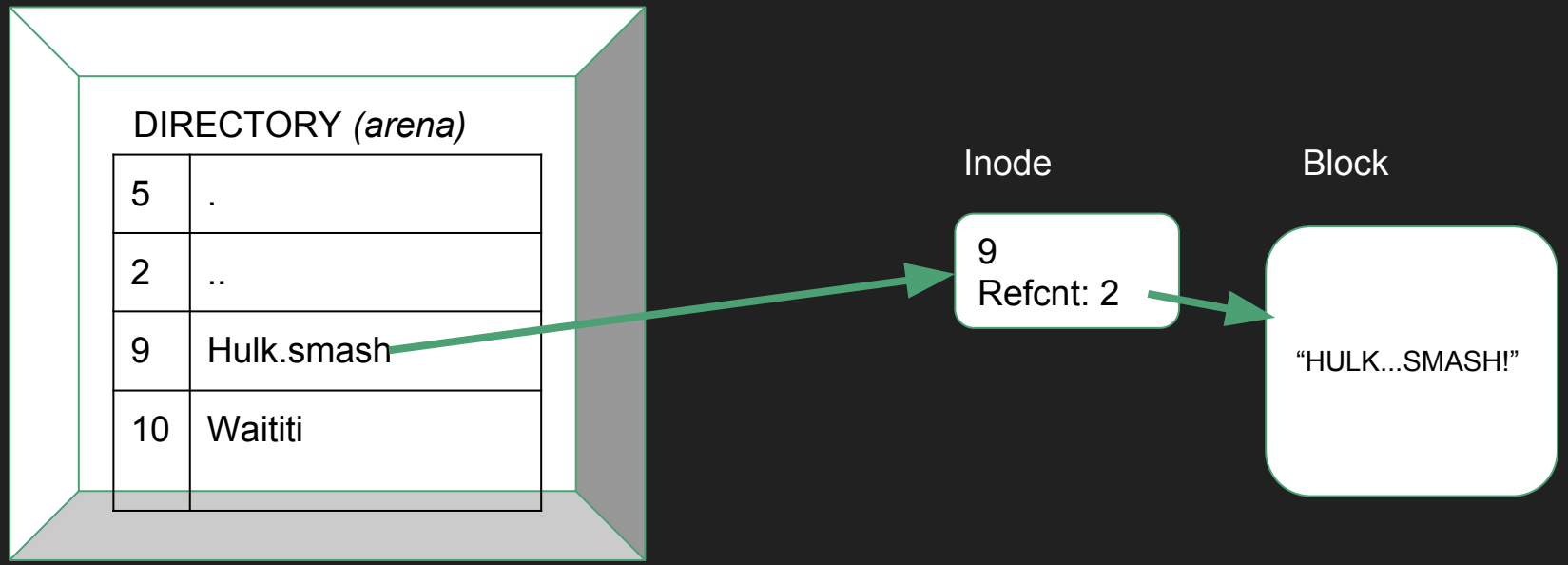
Where’s Hulk?

Hulk is located at /MCU/space/Sakaar/arena/Hulk.smash. How can we find him?



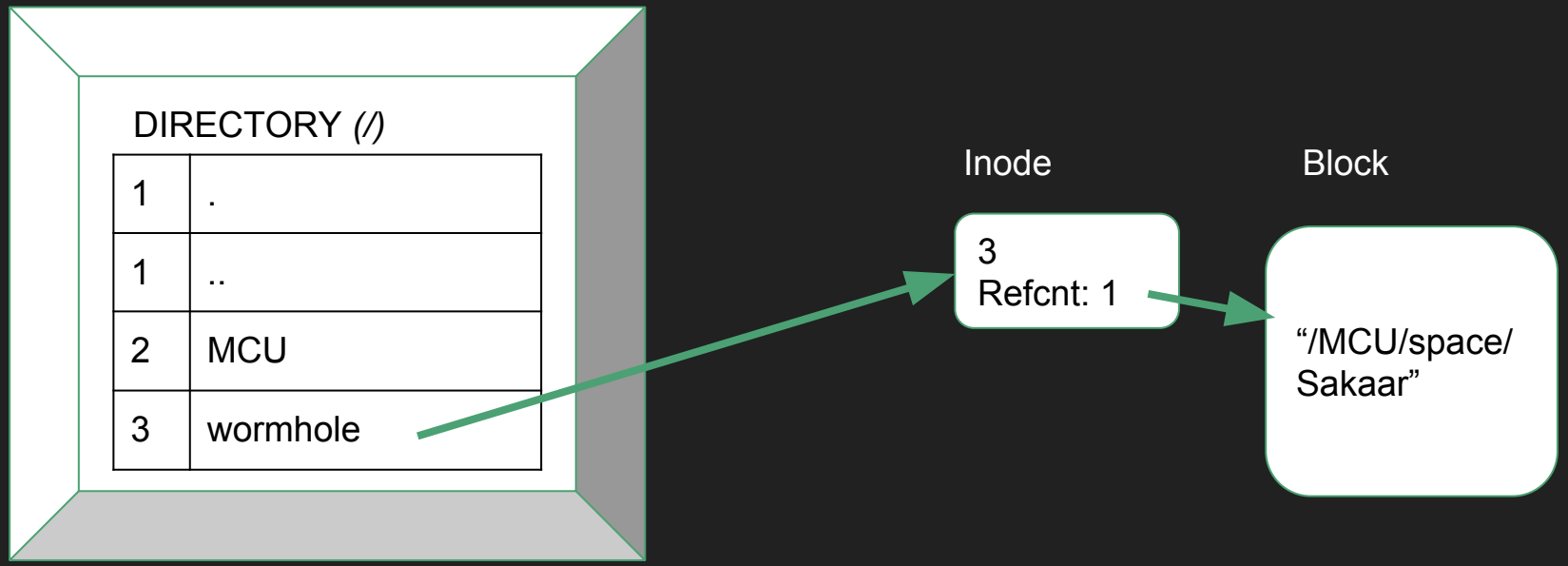
Where’s Hulk?

Hulk is located at /MCU/space/Sakaar/arena/Hulk.smash. How can we find him?



Where's Hulk?

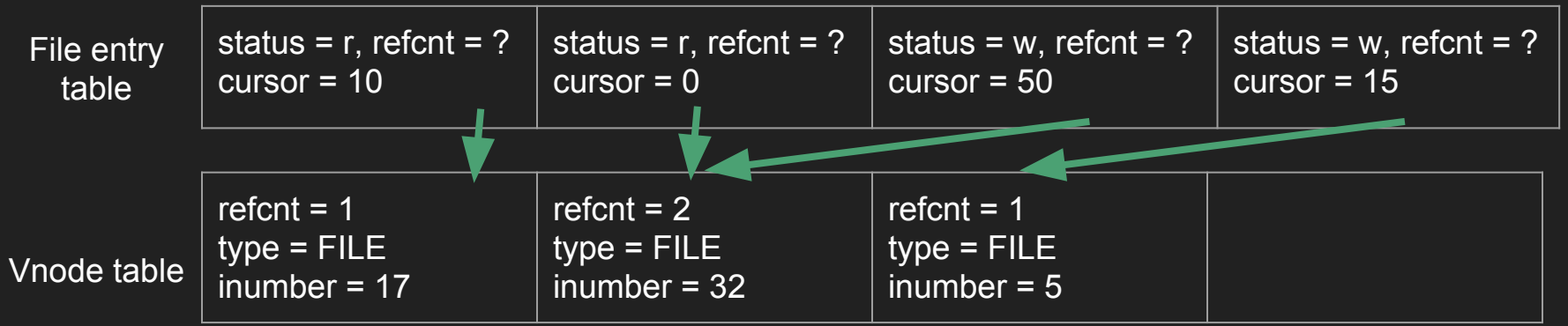
Hulk is located at /MCU/space/Sakaar/arena/Hulk.smash. How can we find him?



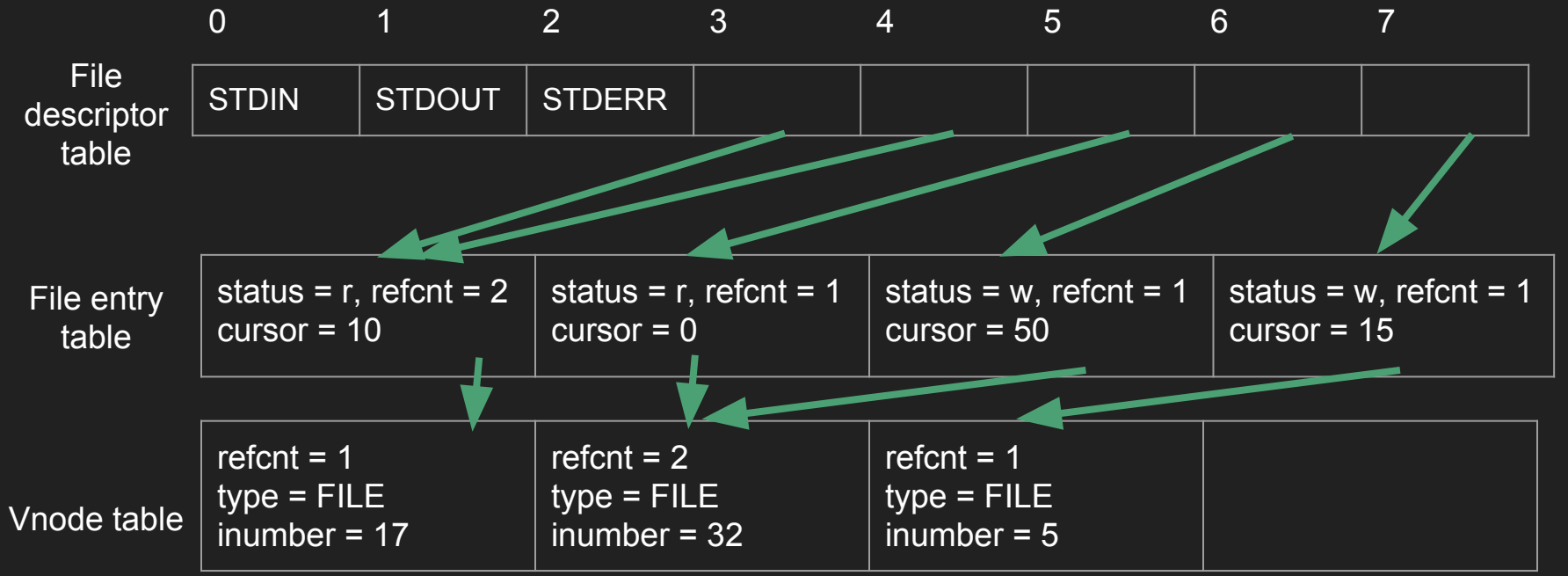
The Filesystem Tables (stored in Kernel Space)

Vnode table	a.txt	b.txt	c.txt	d.txt

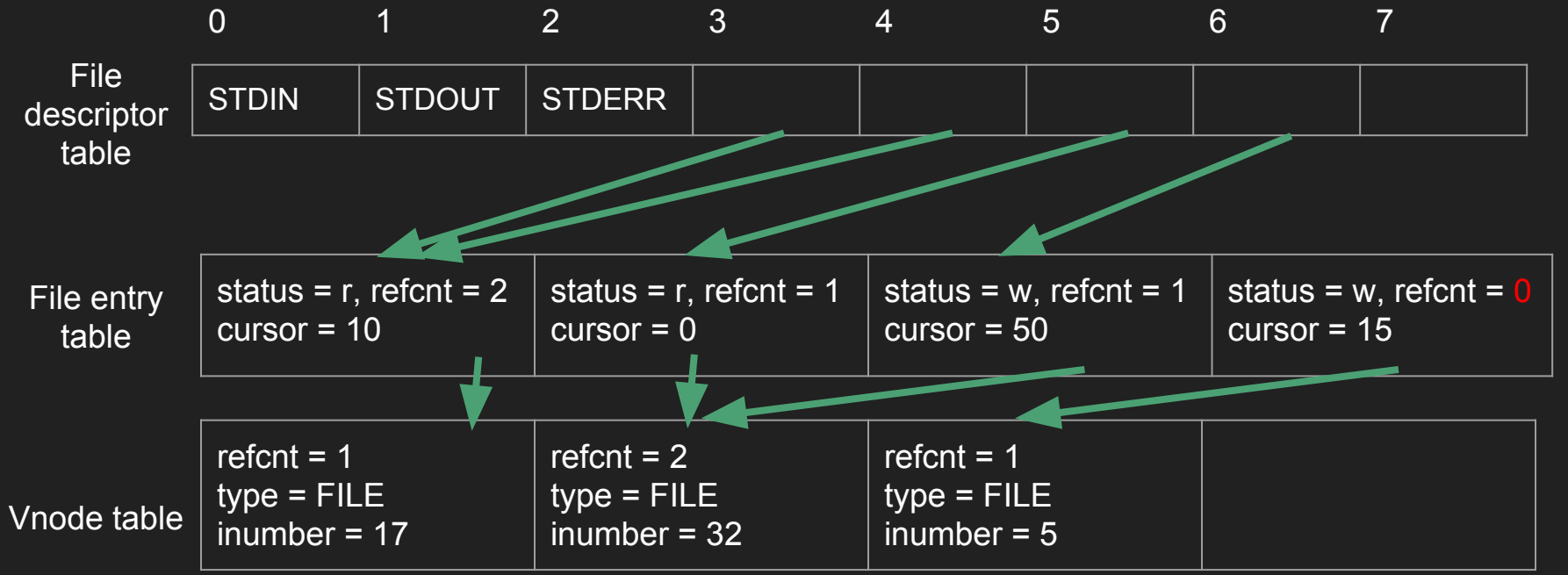
The Filesystem Tables



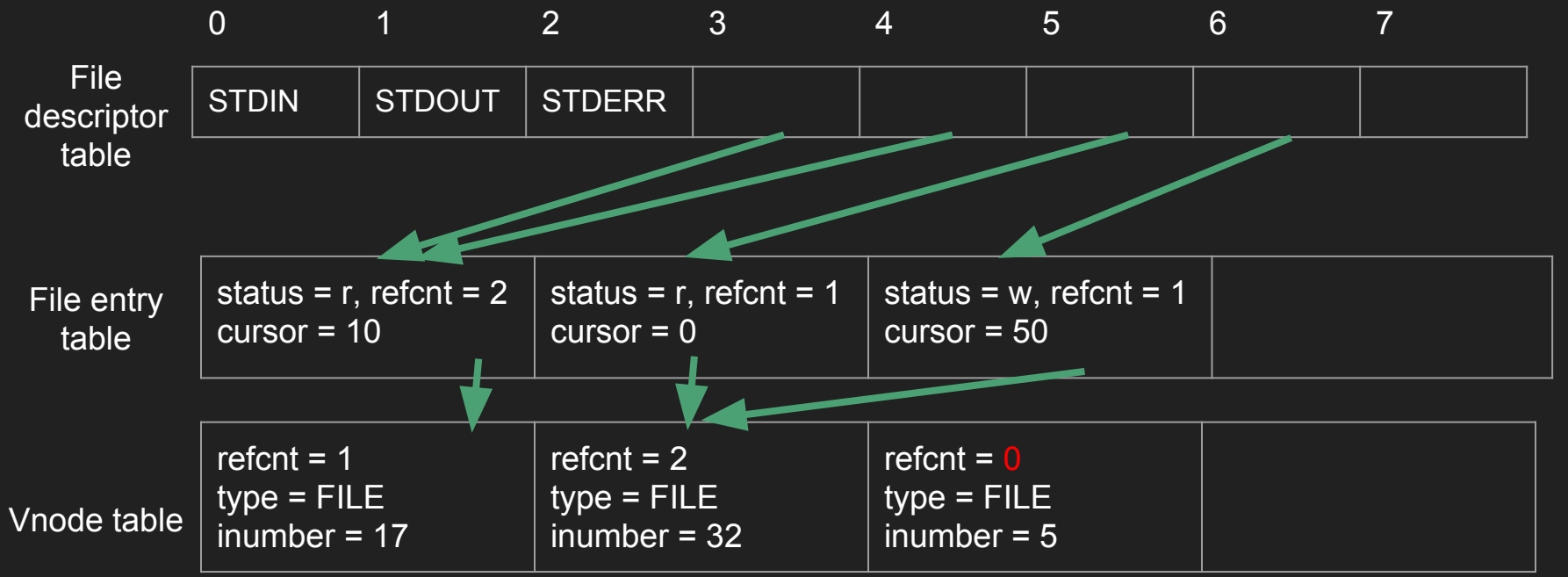
The Filesystem Tables



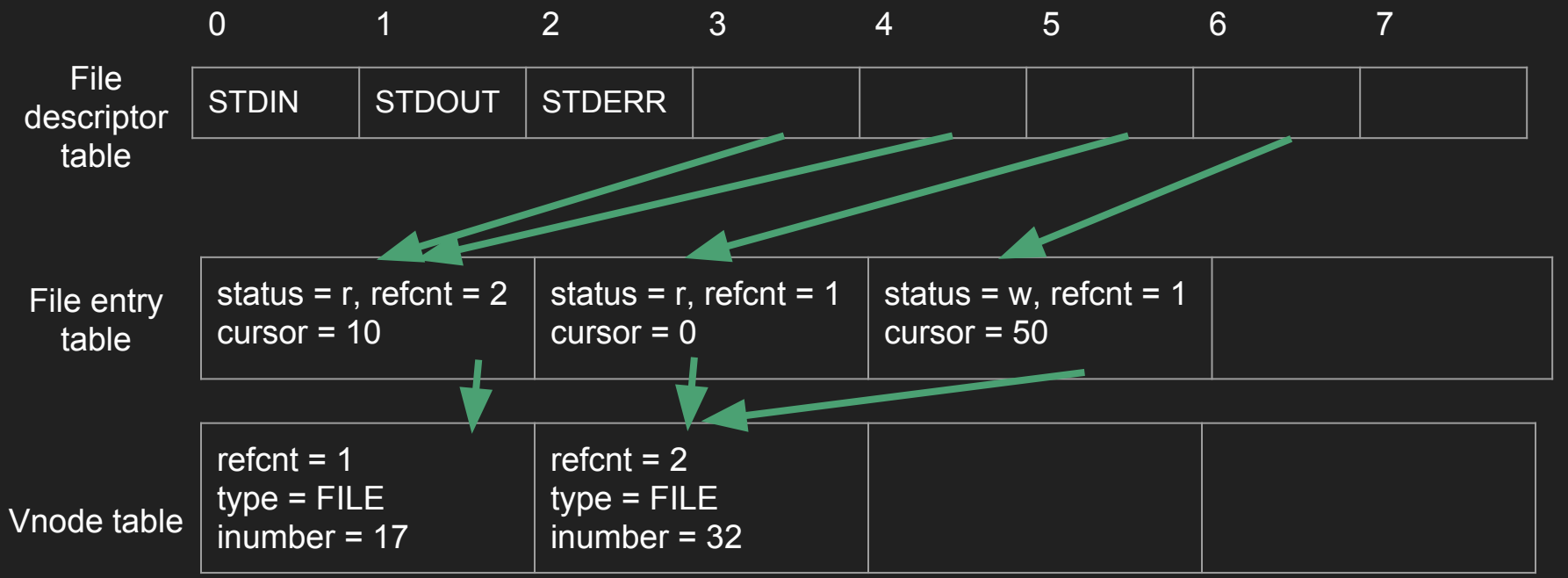
The Filesystem Tables



The Filesystem Tables



The Filesystem Tables



MULTIPROCESSING

See assign2, assign3

System Calls

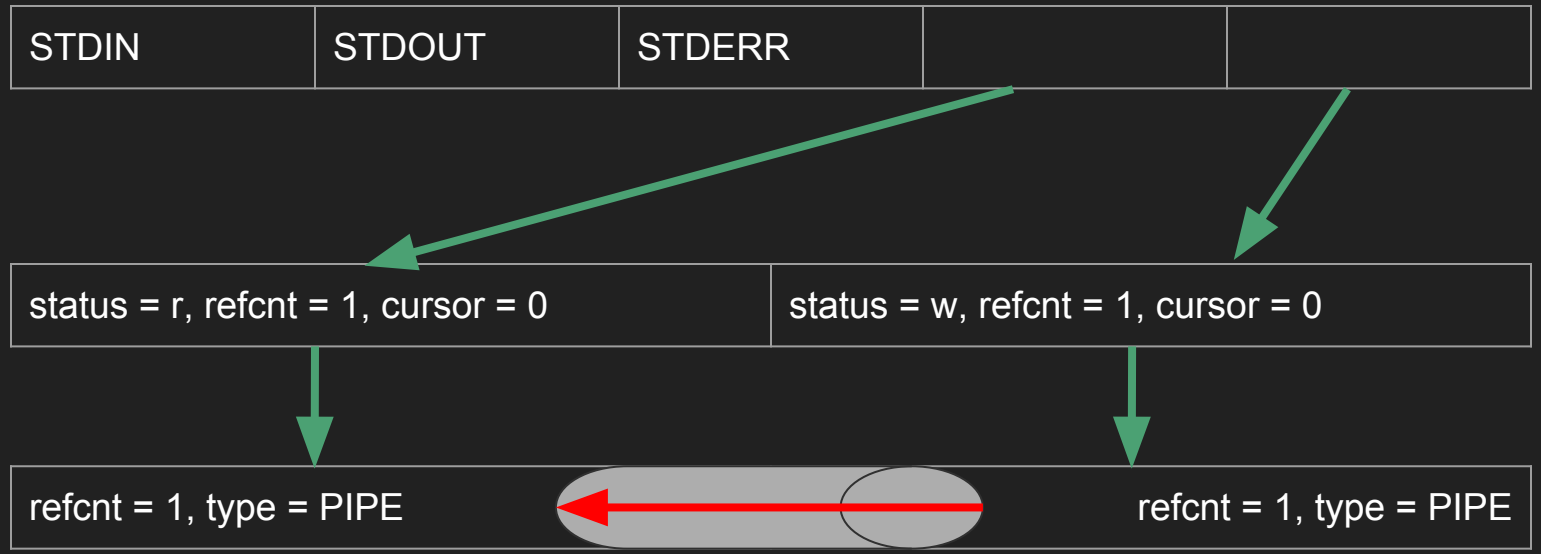
Interact with the raw blocks that users do not (and should not) have access to (privileged operations)

- Syscalls you should know:
  - `open()`
  - `read()`
  - `write()`
  - `close()`
  - `pipe()`
  - `dup2()`
  - `fork()`
  - `execvp()`
  - `kill()`
  - `waitpid()`
  - `kill()`
  - `signal()`
  - `sigprocmask()`
  - `sigsuspend()`

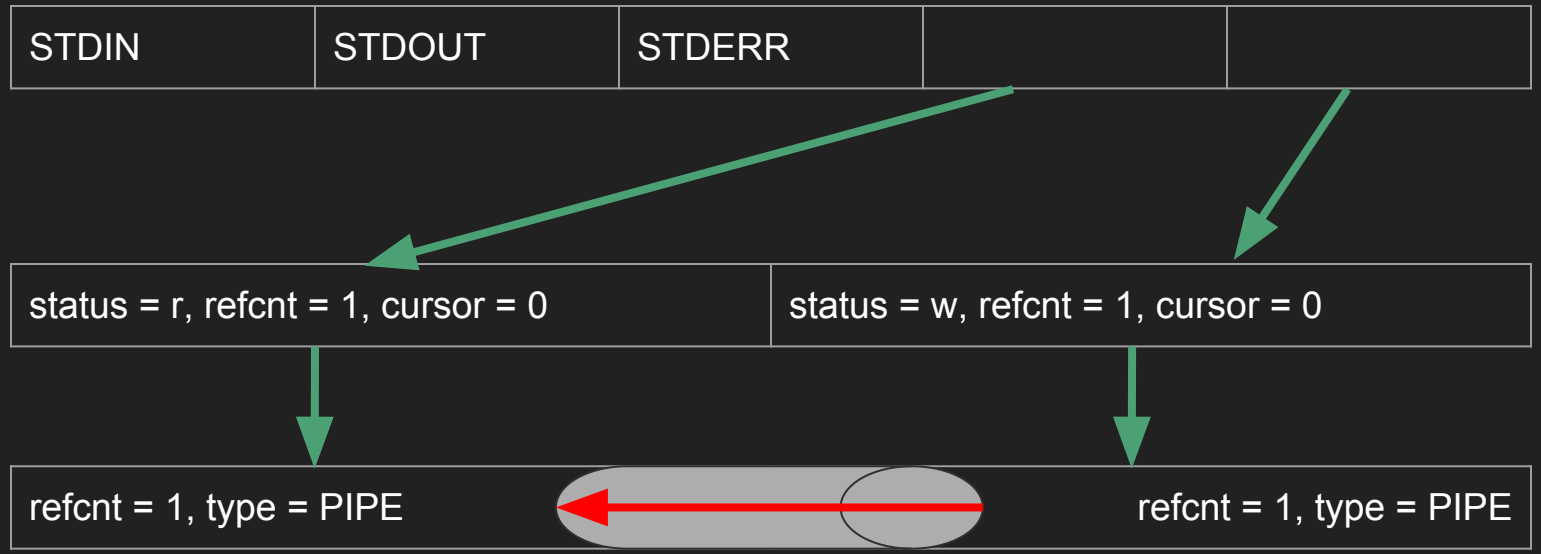
pipe(int[] fds)

STDIN	STDOUT	STDERR		

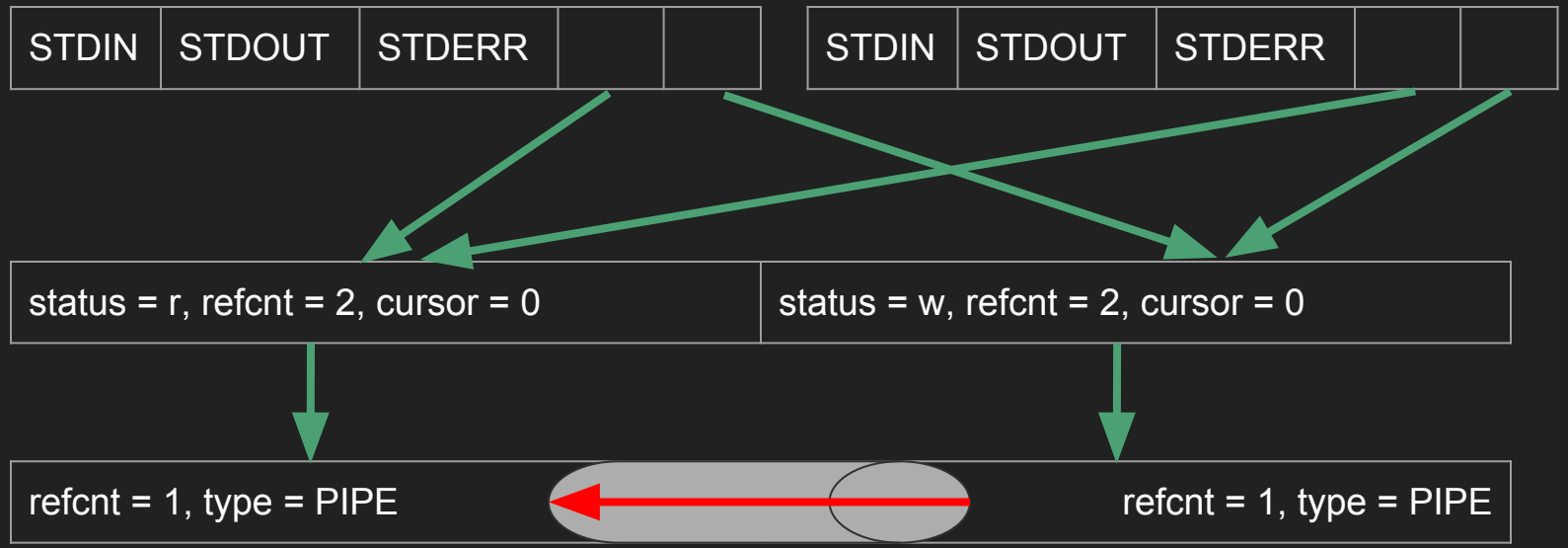
pipe(int[] fds)


STDIN
STDOUT
STDERR
status = r, refcnt = 1, cursor = 0
status = w, refcnt = 1, cursor = 0
refcnt = 1, type = PIPE
refcnt = 1, type = PIPE

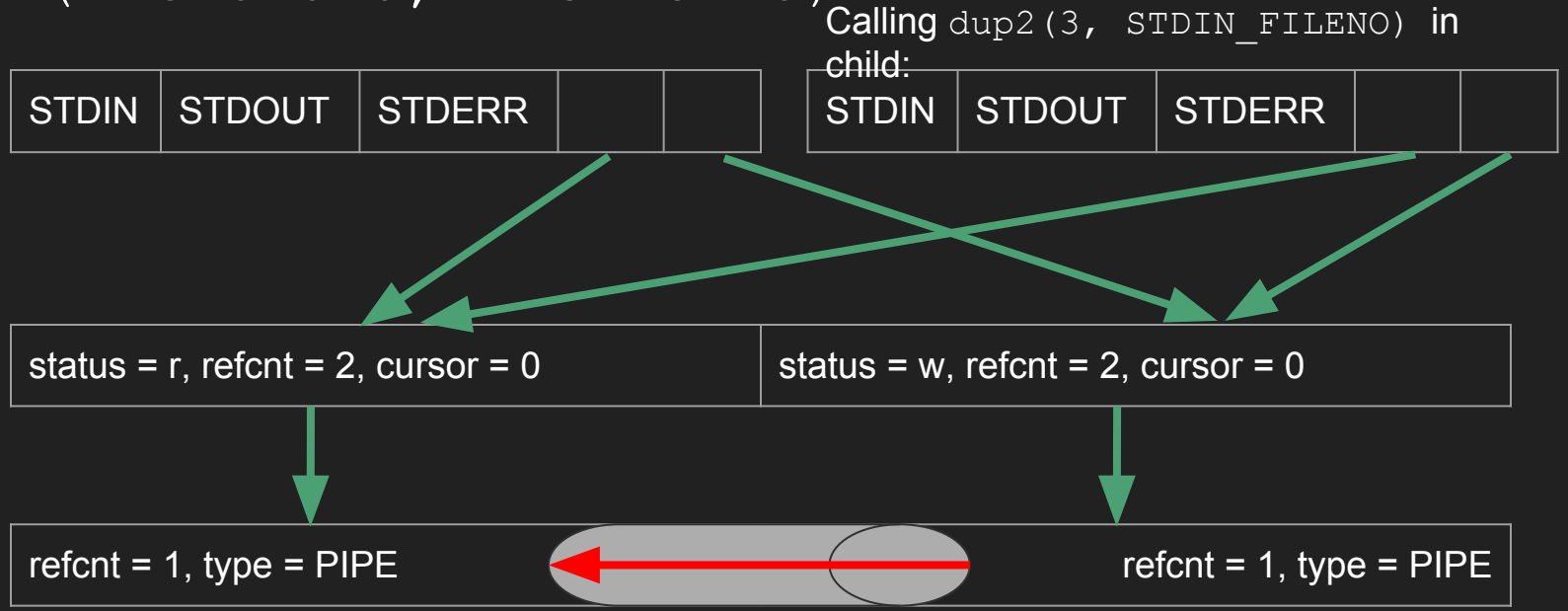
fork()


STDIN
STDOUT
STDERR
status = r, refcnt = 1, cursor = 0
status = w, refcnt = 1, cursor = 0
refcnt = 1, type = PIPE
refcnt = 1, type = PIPE

fork()

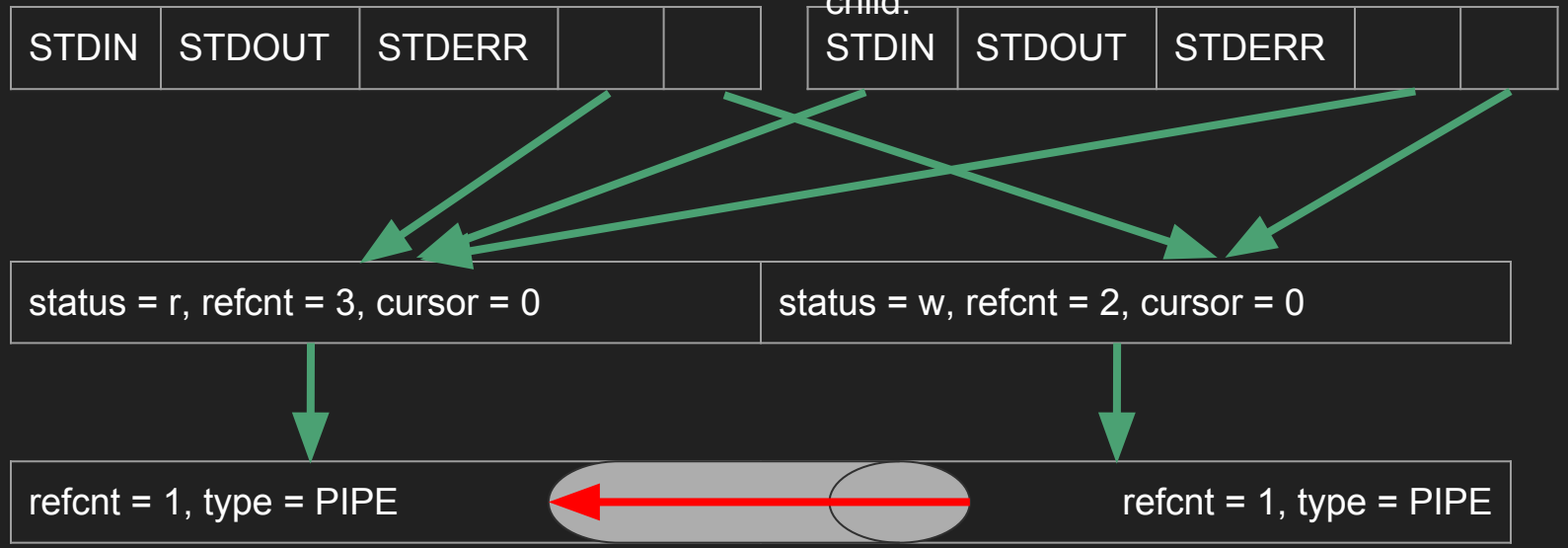

STDIN
STDOUT
STDERR
STDIN
STDOUT
STDERR
status = r, refcnt = 2, cursor = 0
status = w, refcnt = 2, cursor = 0
refcnt = 1, type = PIPE
refcnt = 1, type = PIPE

`dup2(int oldfd, int newfd)`



`dup2(int oldfd, int newfd)`

Calling `dup2(3, STDIN_FILENO)` in child:



What will print here?

```
void createUltron() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Ultron is here.");
    }
    printf("Jarvis is here.");
}
```

```
int main(int argc, char* argv[]) {
    createUltron();
    while(true) {
        pid_t pid = waitpid(-1, NULL, 0);
        if (pid == -1) break;
    }
    assert(errno == ECHILD);
    printf("The world is safe.");
}
```

What will print here?

```
void createUltron() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Ultron is here.");
        return;
    }
    printf("Jarvis is here.");
}
```

```
int main(int argc, char* argv[]) {
    createUltron();
    while(true) {
        pid_t pid = waitpid(-1, NULL, 0);
        if (pid == -1) break;
    }
    assert(errno == ECHILD);
    printf("The world is safe.");
}
```

What will print here?

```
void createUltron() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Ultron is here.");
        exit(0);
    }
    printf("Jarvis is here.");
}
```

```
int main(int argc, char* argv[]) {
    createUltron();
    while(true) {
        pid_t pid = waitpid(-1, NULL, 0);
        if (pid == -1) break;
    }
    assert(errno == ECHILD);
    printf("The world is safe.");
}
```

`waitpid()`

First argument (`pid_t pid`):

- <-1 (any child with pgid = |`pid`|
- -1 (any child)
- 0 (any child with pgid = getpgid())
- >0 the child with pid = `pid`.

Second argument (`int* status`):

- Create `int status` and pass in `&status`
  - `WIFEXITED(status)`
  - `WIFSTOPPED(status)`
  - `WIFCONTINUED(status)`

Third argument (`int options`)

- 0 (returns only after child terminates)
- `WUNTRACED` (also after child stops)
- `WNOHANG` (returns immediately)
- `WCONTINUED` (also after child continues)

Signals

The ones you should know:

- SIGINT (ctrl + C)
- SIGTSTP (ctrl + Z)

The two above can be caught and handled. The two below cannot:

- SIGKILL
- SIGSTOP

These have the same respective behavior, but cannot be caught.

- SIGCHLD
- SIGCONT

Wrong universe, but it worked too well...

`sigsuspend(const sigset_t* mask)`

Does the following ATOMICALLY:

- `sigprocmask(SIG_SETMASK, &mask, &old);`
- `sleep(); // wait for signal to wake us up`
- `sigprocmask(SIG_SETMASK, &old, NULL);`

What will print?

```
int counter = 0;

static void reapChild(int sig) {
     printf(“Wakanda Forever!”);
     counter++;
}

int main(int argc, char* argv[]) {
     pid_t pid = fork();
     if (pid == 0) {
          char[] argv = [“echo”, “Black Panther”, NULL];
          execvp(argv[0], argv);
     }
     sigset_t mask;
     sigemptyset(&mask);
     while (counter < 1) {
          sigsuspend(&mask);
     }
     printf(“T\’Challa has won.”);
}
```

What will print?

```
int counter = 0;

static void reapChild(int sig) {
      printf(“Wakanda Forever!”);
      counter++;
}

int main(int argc, char* argv[]) {
      pid_t pid = fork();
      if (pid == 0) {
            char[] argv = [“echo”, “Black Panther”, NULL];
            execvp(argv[0], argv);
      }
      sigset_t mask;
      sigemptyset(&mask);
      while (counter < 1) {
            sigsuspend(&mask);
      }
      printf(“T\’Challa has won.”);
}
```

RACE CONDITION!!!

What will print?

```
int counter = 0;

static void reapChild(int sig) {
     printf("Wakanda Forever!");
     counter++;
}

int main(int argc, char* argv[]) {
     pid_t pid = fork();
     if (pid == 0) {
          char[] argv = ["echo", "Black Panther", NULL];
          execvp(argv[0], argv);
     }
     sigset_t mask;
     sigemptyset(&mask);
     sigset_t existing = blockSIGCHLD();
     while (counter < 1) {
          sigsuspend(&mask);
     }
     unblockSIGCHLD(existing);
     printf("T\'Challa has won.");
}
```

sigprocmask(int how, const sigset_t* set, sigset_t* oldset)

```
sigset_t blockSIGCHLD() {
    sigset_t mask;
    sigset_t existing;
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);
    sigprocmask(SIG_BLOCK, &mask, &existing);
    return existing;
}

void unblockSIGCHLD(sigset_t existing) {
    sigset_t mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);
    sigprocmask(SIG_UNBLOCK, &mask, &existing);
}
```

Virtual Memory

Virtualization

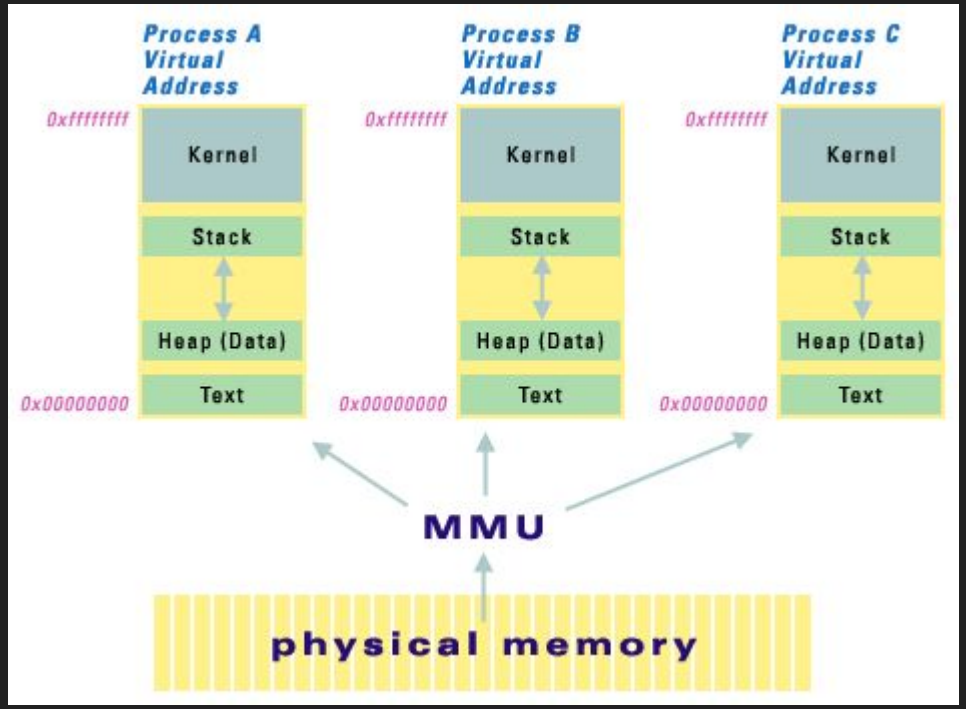
Every process thinks it has exclusive access to addresses 0x00000000 to 0xffffffff.

Virtualization

Every process thinks it has exclusive access to addresses 0x00000000 to 0xffffffff.

Virtual Memory

Instead, the kernel keeps the MEMORY MANAGEMENT UNIT which helps map the virtual address spaces of every process to actual locations in physical memory.



Lazy Loading

If a process needs to use a large library, the library will only be loaded into working memory as needed.

That way we can run many more processes at once.

Memory management is LAZY like these guys.

Copy-on-write

While every process has its own virtual memory space, physical memory is only duplicated when necessary.

In fact, this is not even done when a process reads from memory, only when it writes.

Memory management is LAZY like these guys.

Question

What are some situations in which the same virtual address in multiple processes map to the same physical address in main memory?

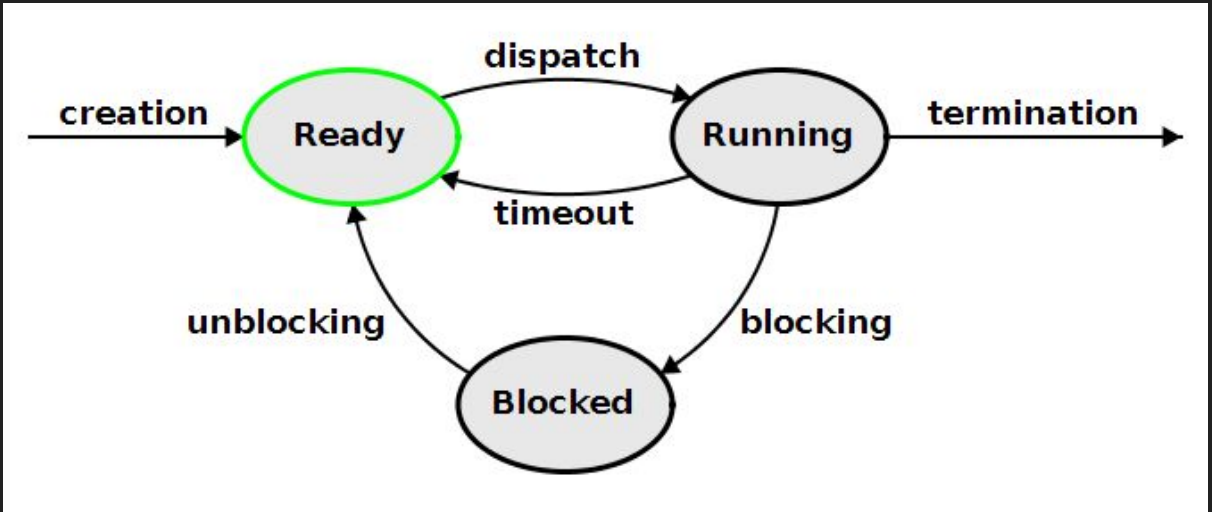
Iron Man is stumped. Are you?

Question

What are some situations in which the same virtual address in multiple processes map to the same physical address in main memory?

- Right after a fork if the OS is using copy-on-write
- Forked processes explicitly want to share data among themselves (mmap)
- Shared code segments

Scheduling


dispatch
creation
Ready
Running
termination
timeout
unblocking
blocking
Blocked

Scheduling

Each process has an associated PCB (process control block) representing its state.

This includes (among other things):

- Register values
  - Including %rip so it knows what code was being executed
- PID

Scheduling

Each process has an associated PCB (process control block) representing its state.

This includes (among other things):

- Register values
  - Including %rip so it knows what code was being executed
- PID

From Lab 2:

- What commands may or may not move a running process to the blocked set?
- What commands are 100% guaranteed to move a running process to the blocked set?
- What needs to happen for a blocked process to be moved back into the ready queue?

MULTITHREADING

No assignments yet (assign4, assign5, assign6)

What is a thread?

- Threads are sometimes referred to as “lightweight processes”.
- They still have their own stacks, but unlike processes they share the same heap, global variables, file descriptor table, etc.
- We create them in C++ by passing in the method we want to run in the newly spawned thread:

```
thread newThread =
thread(sayHello());
```

- The above line will spawn a thread called newThread that will run concurrently until sayHello() is completed.

Joining threads

If you forget to call waitpid() on a process, you have a memory leak.

If you forget to call .join() on a thread, your code will crash!

Passing arguments to threads

Suppose we want a new thread to run the method `foobar(int n, semaphore& sem)`, where `foobar()` is a method in `MyClass`. There are a few ways to do so:

- `thread t([this, n, &sem] {foobar(n, sem)});`
  - This captures all necessary variables and defines a lambda function that takes no parameters
- ```
  thread t([this](int number, semaphore& s) {
      foobar(number, s);
  }, n, ref(sem));
  ```
  - This instead defines a lambda function that takes two parameters
- `thread t(&MyClass::foobar, this, n, ref(sem));`
  - This is more like what we saw in Lab 3.
  ```

Synchronization

This picture from last quarter's review session sums this up better than any Marvel picture.

Mutex

```
int main(int argc, const char *argv[]) {
    int counter = 0;

    thread thread1 = thread([&] () {
        counter++;
    });
    thread thread2 = thread([&] () {
        counter++;
    });

    thread1.join();
    thread2.join();

    cout << "counter = " << counter << endl;
    return 0;
}
```

As we saw, the code on the left is not thread-safe. Because ++ is not an atomic operation (the value is copied into a register, increased, and copied back), it is possible that we will end up printing

“counter = 1”

Mutex

```
static mutex counterLock;

int main(int argc, const char *argv[]) {
    int counter = 0;

    thread thread1 = thread([&] () {
        counterLock.lock();
        counter++;
        counterLock.unlock();
    });
    thread thread2 = thread([&] () {
        counterLock.lock();
        counter++;
        counterLock.unlock();
    });

    thread1.join();
    thread2.join();

    cout << "counter = " << counter << endl;
    return 0;
}
```

This fixes the issue because whenever a thread is accessing or modifying counter's value, no other thread can be doing so.

When `counterLock.lock()` is reached in one thread, it will block any other thread that reads that line until the lock is released.

Be sure to release the lock before going out of scope.

Or you could use a...

Lock Guard

```
static mutex counterLock;

int main(int argc, const char *argv[]) {
    int counter = 0;

    thread thread1 = thread([&] () {
        lock_guard<mutex> lg(counterLock);
        counter++;
    });
    thread thread2 = thread([&] () {
        lock_guard<mutex> lg(counterLock);
        counter++;
    });

    thread1.join();
    thread2.join();

    cout << "counter = " << counter << endl;
    return 0;
}
```

This is the same thing as using a mutex, except will automatically unlock when `lg` goes out of scope.

Condition Variable

Unlike mutexes and lock guards, these provide notifications when a state has changed. In this sense, they are similar to signals.

Similar to `sigsuspend()`, we get race conditions if we repeatedly check a condition and then wait until that condition may no longer hold. Just as `sigsuspend()` atomically unblocks signals and waits, condition variables atomically check the condition and wait.

Condition Variable commands:

- `condition_variable_any cv;`
- `cv.wait(mutex m, Predicate p);`
- `cv.notify_all();`

Also `cv.notify_one()`, but more on that later.

Condition Variable

```
while (numQueued == 0) {
    //WHAT IF numQueued becomes 0 right here?  RACE CONDITION!!!
    numQueuedLock.unlock();
    queueCv.wait();
    numQueuedLock.lock();
}

vs.

queueCv.wait(numQueuedLock, [&](){return numQueued > 0;}); //Since this is done atomically, no risk.
```

Semaphore

Very often, we will want to limit the number of threads that can be doing something, but not restrict it to a single thread.

Semaphore

Very often, we will want to limit the number of threads that can be doing something, but not restrict it to a single thread.

ENTER SEMAPHORE!!!

Semaphore

Very often, we will want to limit the number of threads that can be doing something, but not restrict it to a single thread.

A semaphore can be thought of as a set of permission slips. The initial value is the number of permission slips, `signal()` adds a permission slip, and `wait()` (once it is unblocked) takes a permission slip. Again, this is all done atomically.

Semaphore commands:

- `semaphore sem(int value);`
- `sem.signal();`
- `sem.wait();`

Just like with mutexes (mutices? this seems to be a point of contention on stack overflow...) be sure to `signal()` before going out of scope, because signaling is not the default behavior.

Semaphore

Very often, we will want to limit the number of threads that can be doing something, but not restrict it to a single thread.

A semaphore can be thought of as a set of permission slips. The initial value is the number of permission slips, `signal()` adds a permission slip, and `wait()` (once it is unblocked) takes a permission slip. Again, this is all done atomically.

Keep in mind that semphores are implemented using condition variables, so anything a semaphore does can also be done using only condition variables (but shouldn't be!).

Semaphore commands:

- `semaphore sem(int value);`
- `sem.signal();`
- `sem.wait();`

Just like with mutexes (mutices? this seems to be a point of contention on stack overflow...) be sure to `signal()` before going out of scope, because signaling is not the default behavior.

Example

The Avengers need to fight Thanos! They need to take away all 5 infinity stones to win, but only 3 of them can attack Thanos at any given time (this entails sleeping and with some probability removing an infinity stone)! How can we handle this?

Example

```
numInfinityStones = 5;

int main(int argc, char* argv[]) {
      thread avengers[kNumAvengers];
      for (int i = 0; i < kNumAvengers; i++) {
            thread(fightThanos, i);
      }
      for (thread avenger : avengers) {
            avenger.join();
      }
      if (numInfinityStones == 0) {
            cout << oslock << “Hooray, the Avengers have defeated Thanos!” << endl << osunlock;
      } else {
            cout << oslock << “Yep, the Universe has been destroyed..." << endl << osunlock;
      }
}
```

Example

```
void fightThanos(int i) {
     //SOMEHOW WE CAN ONLY HAVE 3 MOVE ON AT A TIME:
     sleep_for(rand() % i * 100);
     if (rand() % i == 0) numInfinityStones--;
}
```

Example

```
static semaphore attackPermission(3);

void fightThanos(int i) {
     attackPermission.wait();
     sleep_for(rand() % i * 100);
     if (rand() % i == 0) numInfinityStones--;
     attackPermission.signal();
}
```

Example

```
static semaphore attackPermission(3);
static mutex infinityStoneLock;

void fightThanos(int i) {
     attackPermission.wait();
     sleep_for(rand() % i * 100);
     if (rand() % i == 0) {
          lock_guard<mutex> lg(infinityStoneLock);
          numInfinityStones--;
     } //lg goes out of scope here, infinityStoneLock is released.
     attackPermission.signal();
}
```

Threads

Processes

Threads

- `Thread.join()` `//wait for a thread to finish`

Processes

- ?

Threads

- `Thread.join()`

Processes

- `waitpid()`

Threads

- `Thread.join()`
- `cv.wait(pred)` `//block until some notification that the state may have changed`

Processes

- `waitpid()`
- `?`

Threads

- `Thread.join()`
- `cv.wait(pred)`

Processes

- `waitpid()`
- `while (pred) {sigsuspend()}`

Threads

- `Thread.join();`
- `cv.wait(pred);`
- `m.lock(); //prevent other threads from interrupting`

Processes

- `waitpid();`
- `while (pred) {sigsuspend();}`
- `?`

Threads

- `Thread.join();`
- `cv.wait(pred);`
- `m.lock();`

Processes

- `waitpid();`
- `while (pred) {sigsuspend();}`
- `sigprocmask(SIG_BLOCK,...);`

GOOD LUCK!

Any questions?