



# Lab 2 Intro

CSE451 22wi - 1/13/22



# Admin

- Lab 2 design due 4/15 (next Friday)
- Lab 2 due 4/22 (two weeks from now)
- Lab 2 has a design doc. The better you fill it out, the more helpful we can be in commenting on it, and the more prepared you will be for writing the code!
  - Put design doc in your repo, similar to lab1design.
  - Grading expects *how* details, not just *what* -- more than just copying from spec



# Design Document

Do it BEFORE you write code

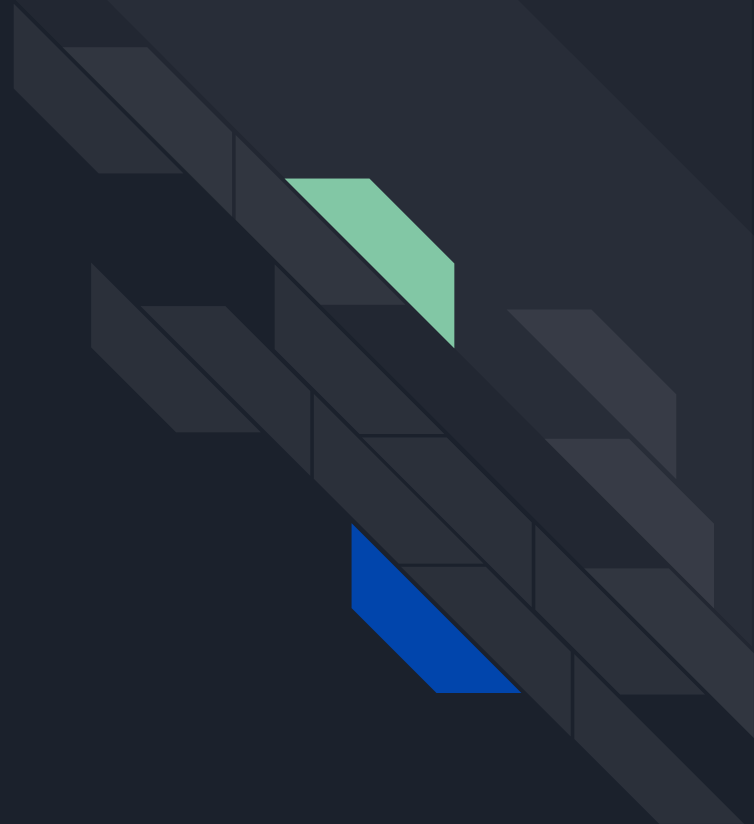
This is mainly for you to think carefully before implementing them  
Include whatever design choice that will help you succeed

Knowing what to include is difficult (you probably haven't done this before!)  
You'll learn as the quarter goes  
Use [lab/designdoc.md](#) & [lab1design](#) as a reference of what should be included!  
Edge cases, unanswered questions

Office hours are a good time to talk about design

# Locks

Question: Why do we need them?





# Spinlocks

- Disable interrupts and spin until resource is acquired
  - Prevent concurrency issues by not yielding to the scheduler until we are done
- Relevant files
  - `inc/spinlock.h`
  - `kernel/spinlock.c`
- Pros/Cons of spin locks?
  - Fast to acquire resource once it's freed up
  - Hangs entire core while waiting
  - Xk is single-core, so nothing will ever actually spin on one of these (can't be interrupted while holding)



# Synchronization Functions

- Main API for process control: `wakeup/sleep`
  - Helper: `wakeup1` (use it if you already hold the ptable lock)
  - `sleep(void* chan, struct spinlock* lk)`
    - Sets process state to **SLEEPING**
      - i.e. won't be scheduled to run by the scheduler
    - Sets the process's "channel" variable (`chan`)
    - Yields to the scheduler, switching to another process
  - `wakeup(void* chan)`
    - Acquire ptable lock
    - Looks for all **SLEEPING** processes with the given channel (`chan`), wakes them up
      - i.e. sets processes state to **RUNNABLE**
- Condition variables
  - Go to sleep using the variable's address as channel. Signal variable change by `wakeup()`. When woken up from sleep, check if condition is now true
    - If not, go back to sleep
    - `while (!condition) {sleep(&var, &mylock);}`
- Relevant files, `inc/proc.h`, `kernel/proc.c`



# Sleeplocks

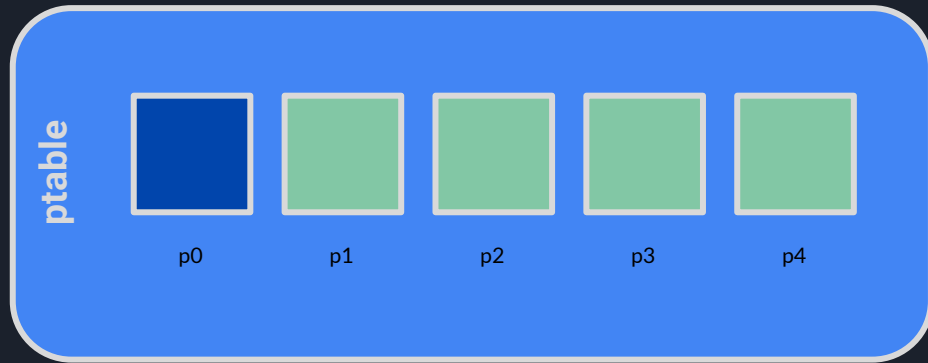
- Uses the `sleep/wakeup` interface from the previous slide, with `&lock` as channel
  - On “`acquiresleep`”, if the resource is unavailable, it will sleep on `&lock`
    - Sets state of process to **SLEEPING**: it will not be scheduled until awoken
  - On “`releasesleep`”, the code with the sleeplock will wakeup all process waiting on `&lock`.
    - Sets all processes sleeping on `&lock` to **RUNNABLE**: they can be scheduled
    - All waiting processes will wake up. One of them will get the lock. The rest will sleep again.
- Relevant files:
  - `inc/sleeplock.h`
  - `kernel/sleeplock.c`
- Pros/Cons?
  - Doesn't waste CPU time waiting for slow operations (e.g. IO)
  - Process gets **descheduled**; **more overhead**



- **WARNING:**
  - CPU scheduling relies on timer interrupts, but spinlocks disable interrupts while active.
  - Trying to schedule while holding a spinlock will cause a “**sched locks**” panic
    - Otherwise, control would never go back to the scheduler
  - Sleeplocks invoke the scheduler if a resource is unavailable
  - Question: how to avoid the panic?



# Sleeplocks - Brief Example



Key



Running



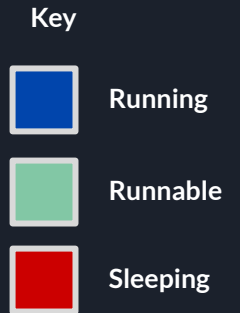
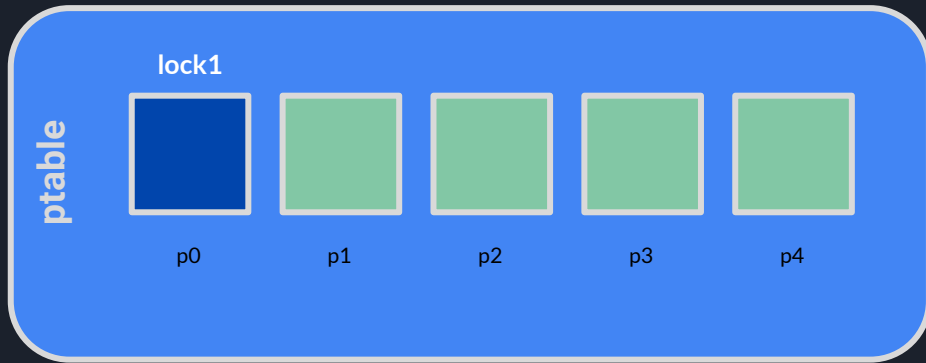
Runnable



Sleeping

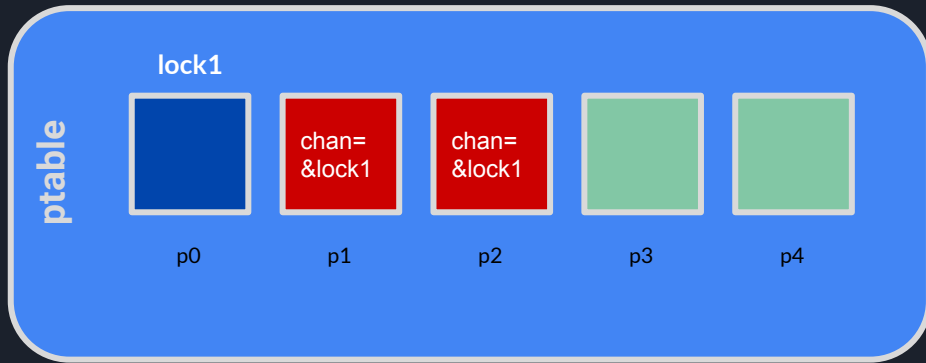
# Sleeplocks - Brief Example

p0 acquires lock1






# Sleeplocks - Brief Example

- p1 and p2 try to acquire lock1
- since p0 is holding, both go to sleep
- `sleep(&lock1, lock1->lk);`



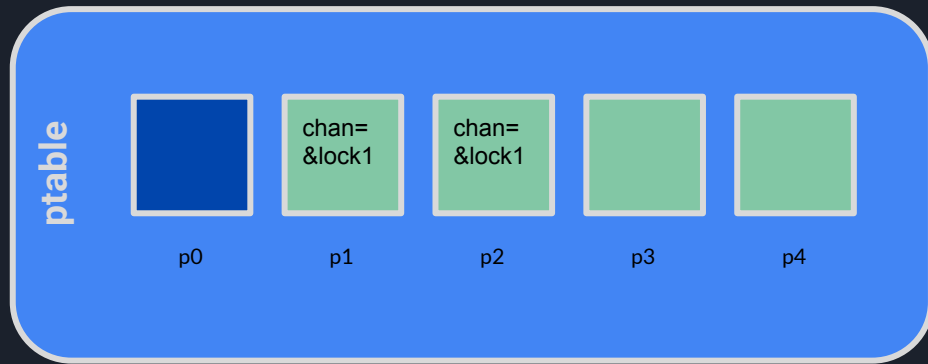
Key

-  Running
-  Runnable
-  Sleeping

# Sleeplocks - Brief Example

p0 releases lock

- calls `wakeup(&lock1)`, waking up p1 and p2



Key



Running



Runnable

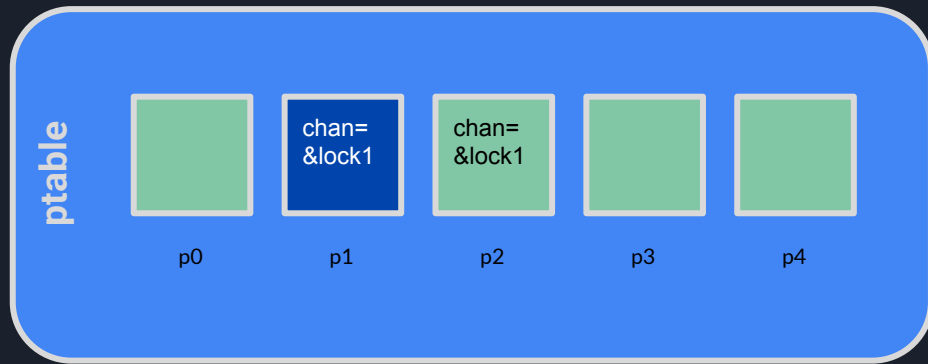


Sleeping

# Sleeplocks - Brief Example

p1 scheduled to run

- acquires lock1 since no other process is holding



Key



Running



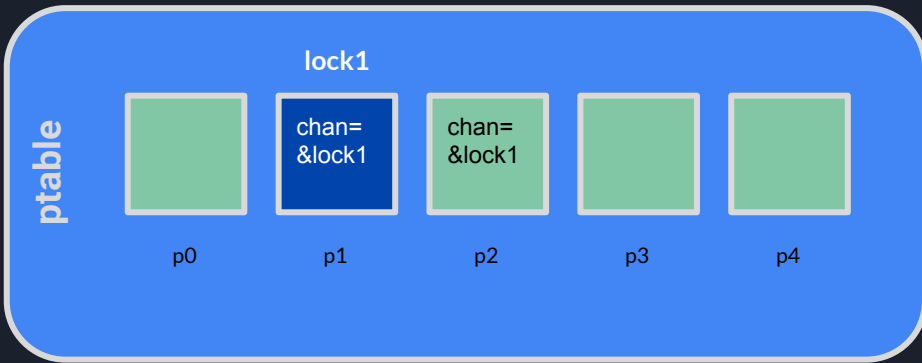
Runnable






Sleeping

# Sleeplocks - Brief Example

p2 scheduled to run  
- sees that the lock is still being held, goes back to sleep

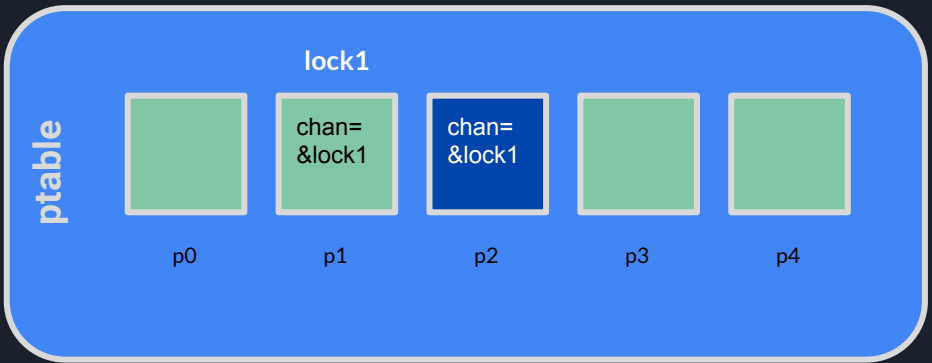


Key




-  Running
-  Runnable
-  Sleeping

# Sleeplocks - Brief Example

p2 scheduled to run  
- sees that the lock is still being held, goes back to sleep



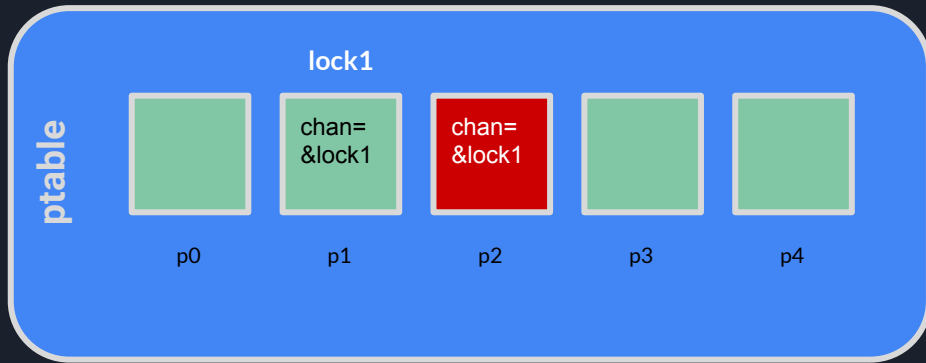
Key

-  Running
-  Runnable
-  Sleeping

# Sleeplocks - Brief Example

p2 scheduled to run

- sees that the lock is still being held, goes back to sleep



Key



Running



Runnable

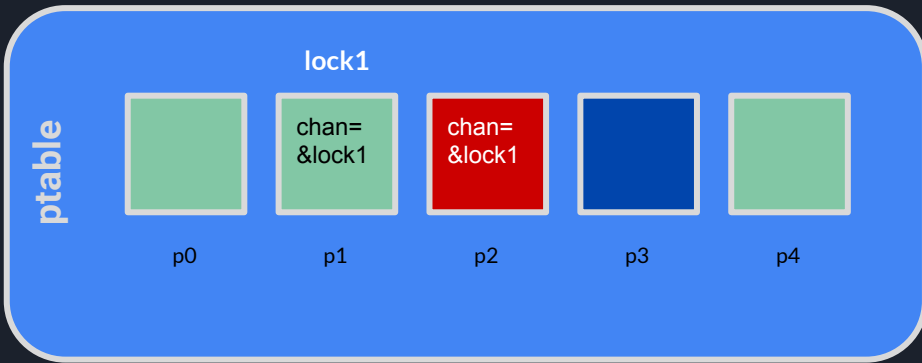


Sleeping



# Sleeplocks - Brief Example

p2 scheduled to run  
- sees that the lock is still being held, goes back to sleep



Key

- Running (blue square)
- Runnable (light green square)
- Sleeping (red square)

# Sleeplocks - Brief Example

- Our condition variable in this case is whether the lock is being held
  - See the pattern: `while (!condition) {sleep(&var, &mylock);}`
- Condition Variables can be used for general purposes, which we'll see later
  - Key functions, sleep and wakeup in `proc.c`

```
// a sleeping lock relinquishes the process
// note mesa semantics: process can wakeup
void acquiresleep(struct sleeplock *lk) {
    acquire(&lk->lk);
    while (lk->locked) {
        sleep(lk, &lk->lk);
    }
    lk->locked = 1;
    lk->pid = myproc()->pid;
    release(&lk->lk);
}
```

```
// a sleeping lock wakes up a waiting process
void releasesleep(struct sleeplock *lk) {
    .....
    acquire(&lk->lk);
    lk->locked = 0;
    lk->pid = 0;
    wakeup(lk);
    release(&lk->lk);
}
```

*excerpts from proc/sleeplock.c*



# Spinlocks vs. Sleeplocks, Summary

- Which should I use?
- Spinlocks
  - Fast to acquire if no contention
  - But, disables interrupts (this is specific to xk. Other systems might not need this)
  - Also wastes CPU cycles if the wait time is very long
- Sleeplocks
  - More overhead, process getting descheduled means context switch
  - But, other processes can run while this process is waiting
- Rule of thumb...
  - Fast critical sections - spinlocks
  - Long critical sections - sleeplocks
- Question: Which should you use for file table?

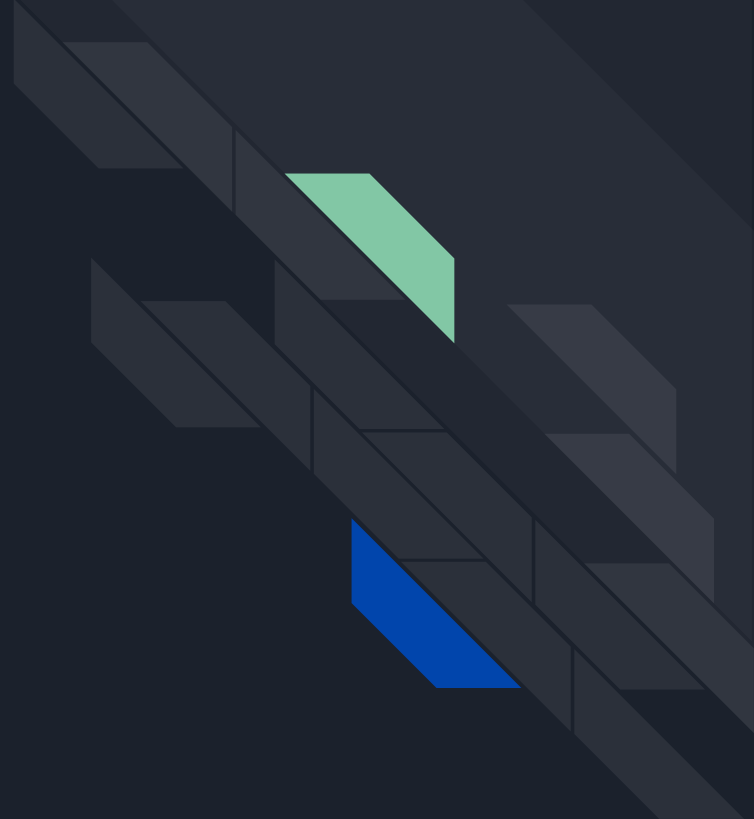


# Curious about locks, still?

- See chapter 5 in the textbook
- Wait 'til later in the quarter (fancier locks)

# Lab 2 - Processes

IF IN DOUBT: DO WHAT LINUX DOES





# fork()

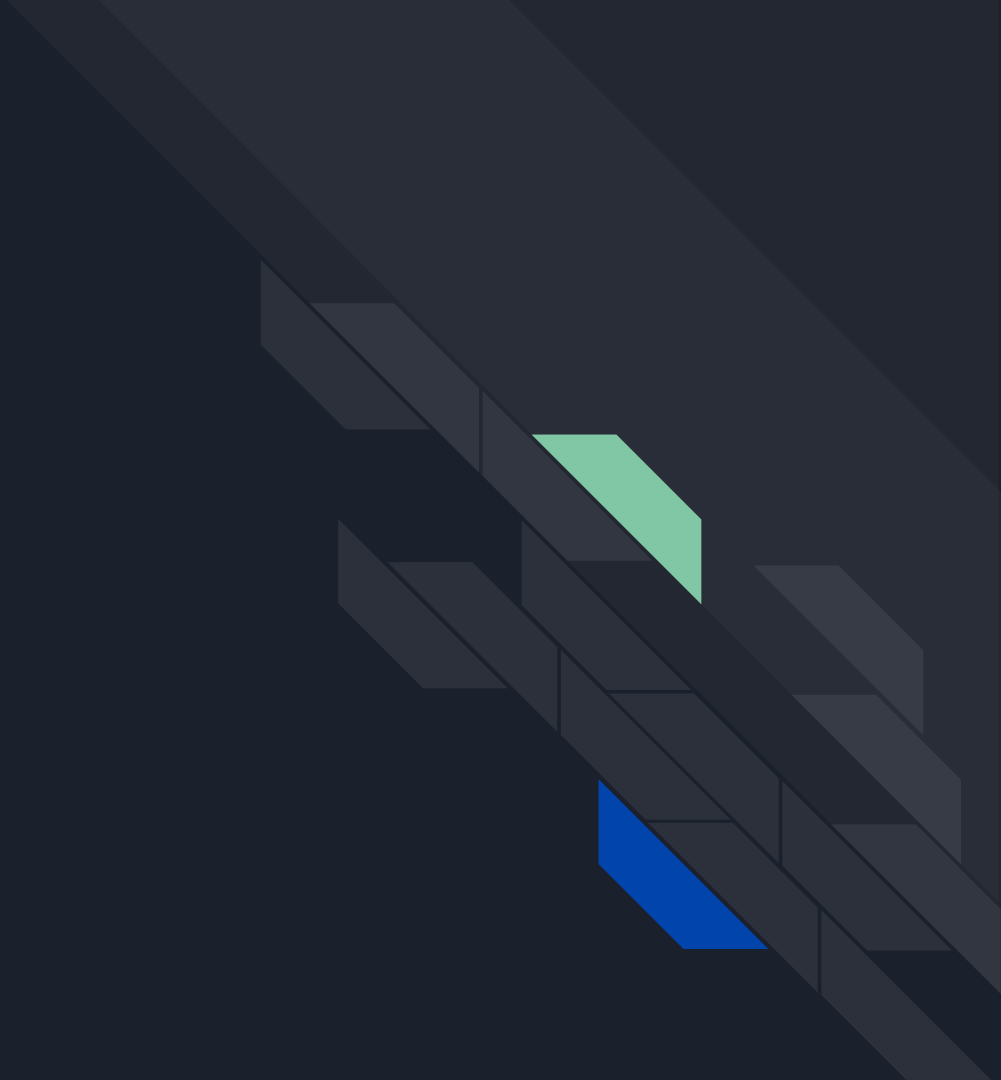
- Create a new process by duplicating the calling process.
- Returns twice!
  - 0 in the child (newly created) process
  - Child's PID in the parent
  - Question: how to do this? (We aren't telling you here!)
- What does this entail? What needs to be created, and how do we copy parent state?
  - Need to clone all open resources
    - Files (make sure to increase reference count)
    - All memory (look into `vspaceinit` and `vspacecopy` to copy virtual memory space)
    - Data to return to the correct place (trap frame)
    - Anything else?



# wait()/exit()

- **wait():** Sleep until a child process terminates, then return that child's PID.
  - Need to keep track of some data
    - Need to know parent/child relationships between processes
  - Process shouldn't return from here until a child has exited...
    - Should put to sleep and invoke the scheduler
- **exit():** Halts program and sets state to have its resources reclaimed
- What are some edge cases to consider?
  - Wait should return child's process ID EVEN IF the child exited before **wait()** was called
  - The kernel and the user program share the same page table
    - Can't clean up all data in **exit()**...
  - Parent should go to sleep until a child exits
  - What if the parent exits before children terminates? Or never calls **wait()**?
    - Need to clean up, somehow.
- For xk, looping through the process table is reasonable

# Lab 2 - Pipe







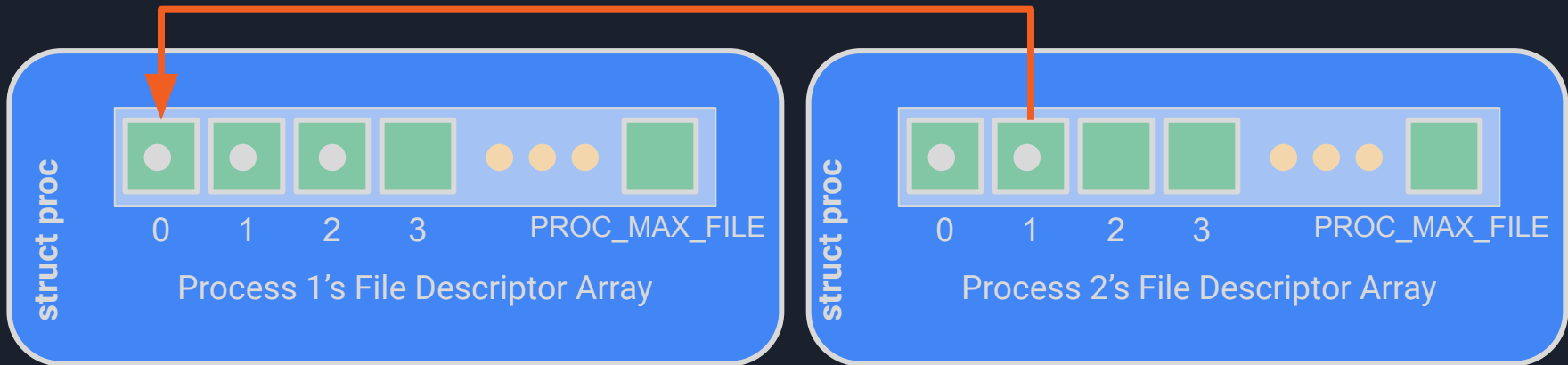
# pipe(pipefds)

- Creates a pipe (internal buffer) for reading from/writing to
- From the user perspective: two new files
  - One (“read end”) is not writable
  - Other (“write end”) is not readable
- In practice, allows parent/child or child/child to communicate with each other.
- You’ll want to somehow make this compatible with the read/write(fd) interface

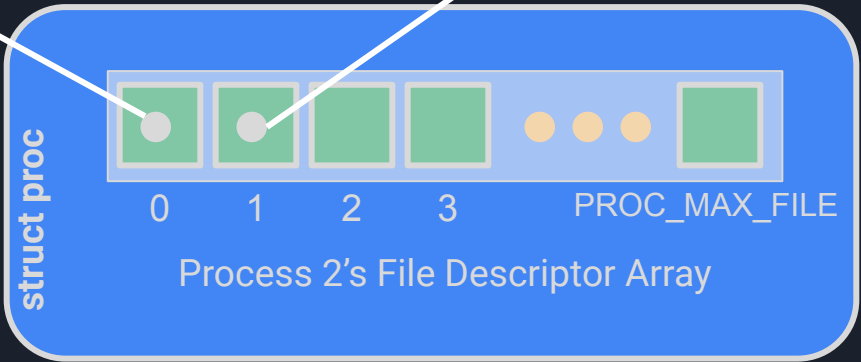
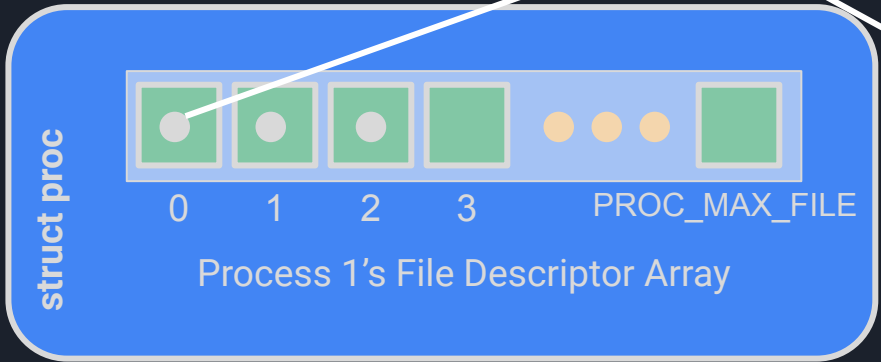
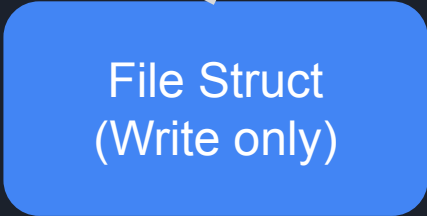
# Pipes

- A mechanism for inter-process communication (“IPC”)
- By calling `sys_pipe`, a process sets up a writing and reading end to a “holding area” where data can be passed between processes

Abstraction of a pipe



# Implementation of a pipe





# Pipes

- What should happen if the write end or read end are closed (by potentially multiple readers/writers)? **When** can you free the buffer?
  - What happens if the buffer is full and we try to write? Empty and try to read?
    - Wait for data to be removed/added
      - Spin or sleep?
    - What if all of the other endpoint type are closed already?
- Pipes should be allocated at runtime, as requested
  - What mechanisms does xk have for dynamic memory allocation to the kernel?
- Each pipe should behave like a file so we can reuse the same **read()** and **write()**
  - Need a way to determine if a struct file is an inode or a pipe



# exec(progname, args)

Replaces the process' state by executing the given program with the given arguments.

This will require you to (carefully!) set up the process' stack memory and register state.

- This will be tricky! You'll be using a number of `vspace__` functions
  - `vspaceinit` for initialization
  - `vspaceloadcode` to load code
  - `vspaceinitstack` to init stack
    - `vspacewritetova` to write initial arguments into the stack
  - `vspaceinstall` to swap in the new vspace
  - `vspacefree` to release the old vspace
- The swapover to the new vspace can be tricky to get right!
  - Look at what `vspacefree` does
  - Remember the difference between copying a struct and copying a pointer



## More on `exec`

- This fully replaces the current process; it does not create a new one
  - Often used with `fork`. Fork off a child as a new process; that child immediately `exec()`s a new program.
    - It's a bit wasteful to copy the entire memory space in `fork()` if it'll be immediately discarded...
    - For now: don't worry about that. Naive `fork` is ok; lab3 will improve upon it
- Many uses
  - The shell uses `fork/exec` to run commands
  - Linux uses `fork/exec` to load new programs
    - Windows has a "launch a new process running *that*" function
    - Linux does not.
    - Whenever you run a new process, forks off of the root process and `execs`.
- Question: how do we pass arguments into the new program?



# X86\_64 Calling Conventions

- `%rdi`: Holds the first argument
  - `%rsi`: Holds the second argument
    - (`%rdx`, `%rcx`, `%r8`, `%r9` come next)
    - Overflow onto stack
  - `%rsp`: Points to the top of the stack (lowest address)
- 
- Local variables are stored on the stack
  - If an array is an argument, the array contents are stored on the stack and the register contains a pointer to the array's beginning



# Main

```
int main(int argc, char** argv)
```

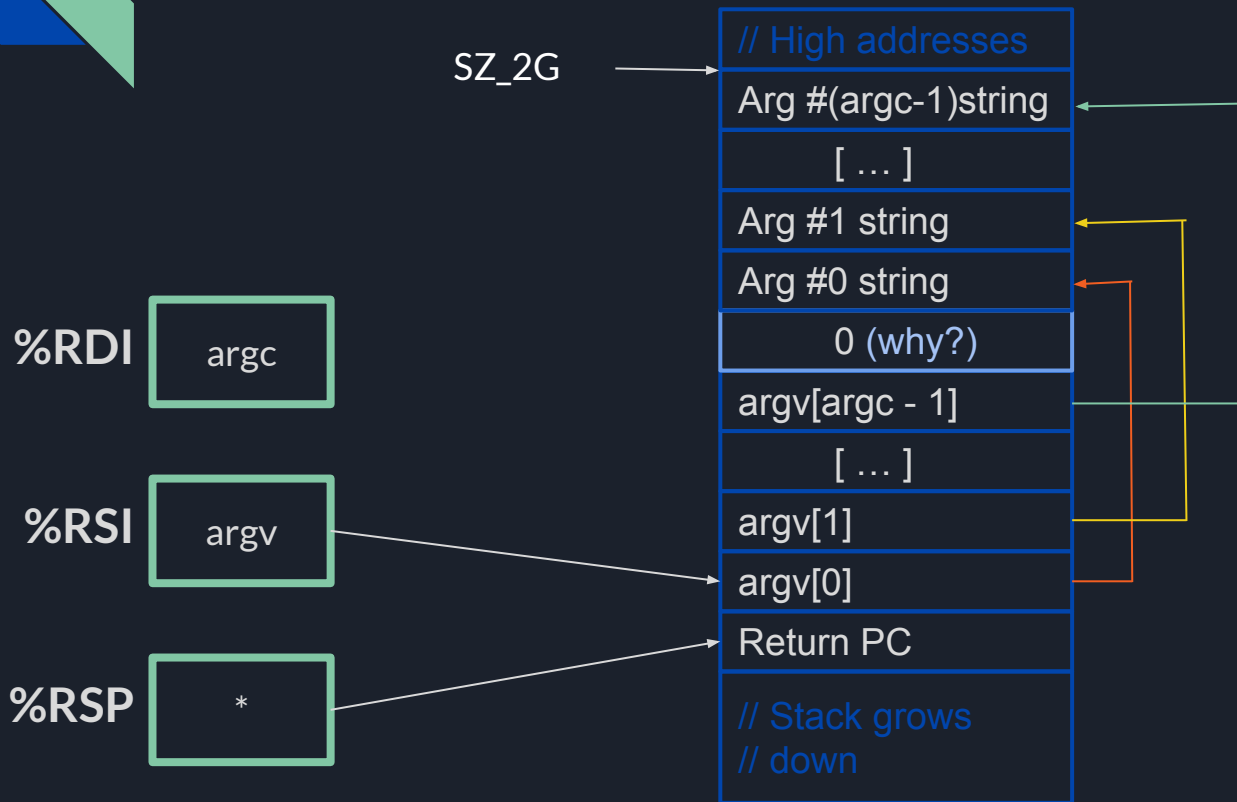
Argc: The number of elements in argv

Argv: An array of strings representing program arguments

- First is always the name of the program
- Argv[argc] = 0



# Main's Stack



- Since `argv` is an array of pointers, `%RSI` points to an array on the stack
- Since each element of `argv` is a `char*`, each element points to a string elsewhere on the stack
- Alignment

# Practice Exercise 1

**%RDI**

???

**%RSI**

???

**%RSP**

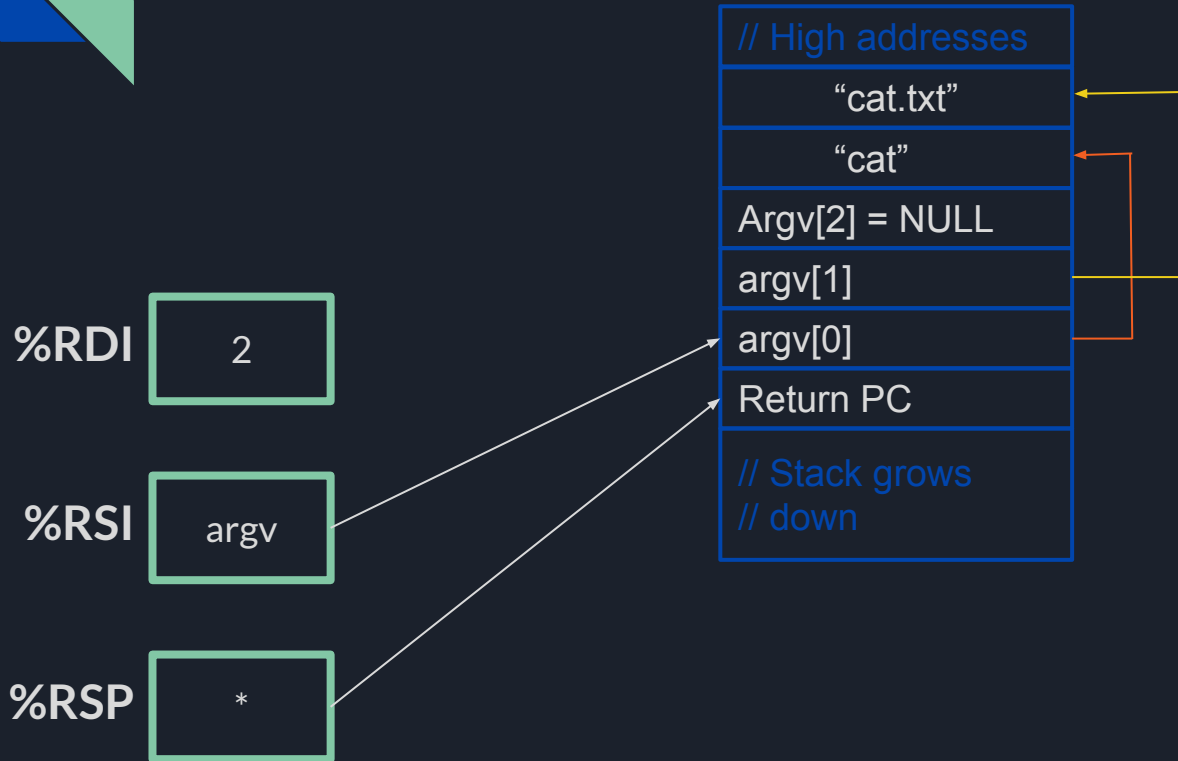
???

// High addresses

// Stack grows  
// down

TODO:  
Draw stack layout and  
determine register values  
for exec called with  
"cat cat.txt"

# Practice Exercise 1: soln



- RDI holds `argc`, which is 2
- RSI holds `argv`: the beginning of the `argv` array
- RSP is properly set to the bottom of the stack.
- The specific value of the return PC doesn't matter (program exits from `main` without returning)

# Practice Exercise 2

**%RDI**

???

**%RSI**

???

**%RSP**

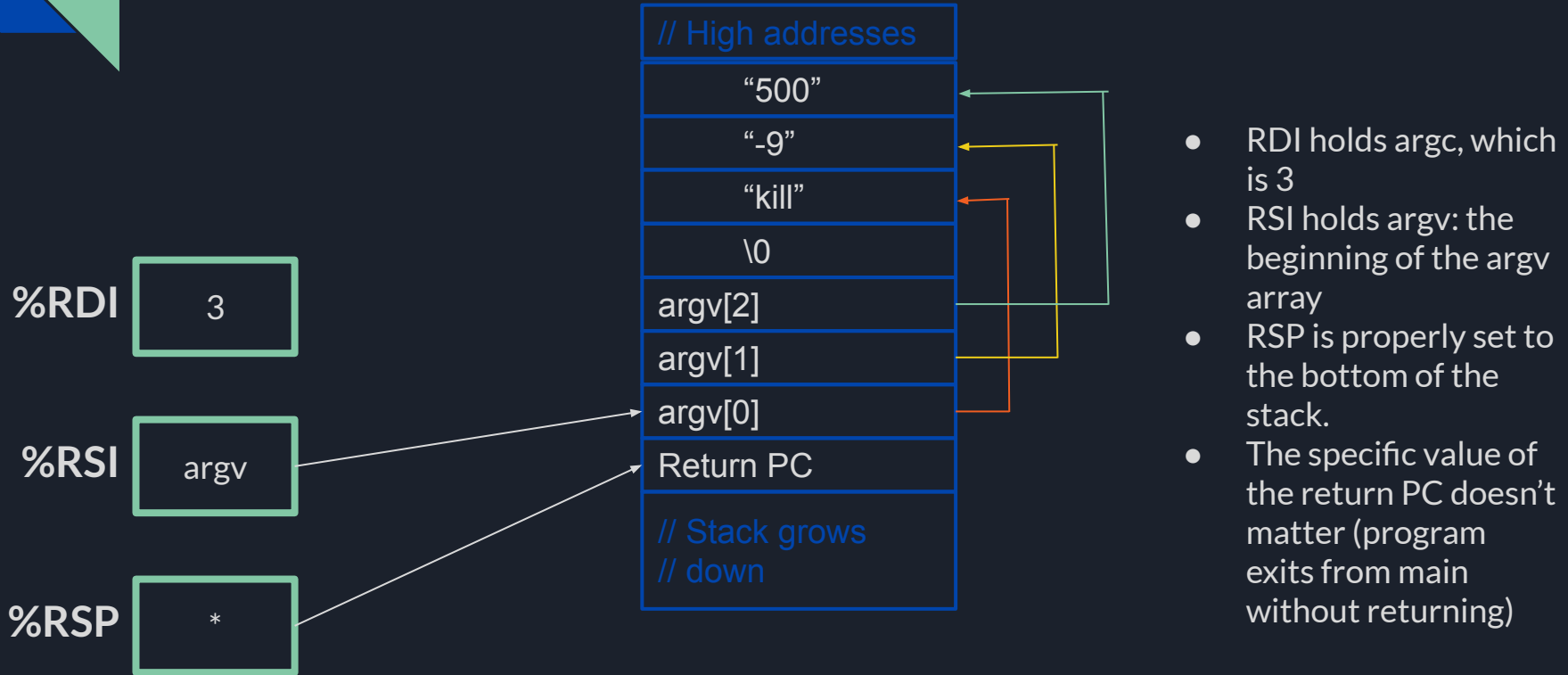
???

// High addresses

// Stack grows  
// down

TODO:  
Draw stack layout and  
determine register values  
for exec called with  
"kill -9 500"

# Practice Exercise 2: soln



- RDI holds argc, which is 3
- RSI holds argv: the beginning of the argv array
- RSP is properly set to the bottom of the stack.
- The specific value of the return PC doesn't matter (program exits from main without returning)