



Lab 2 Intro

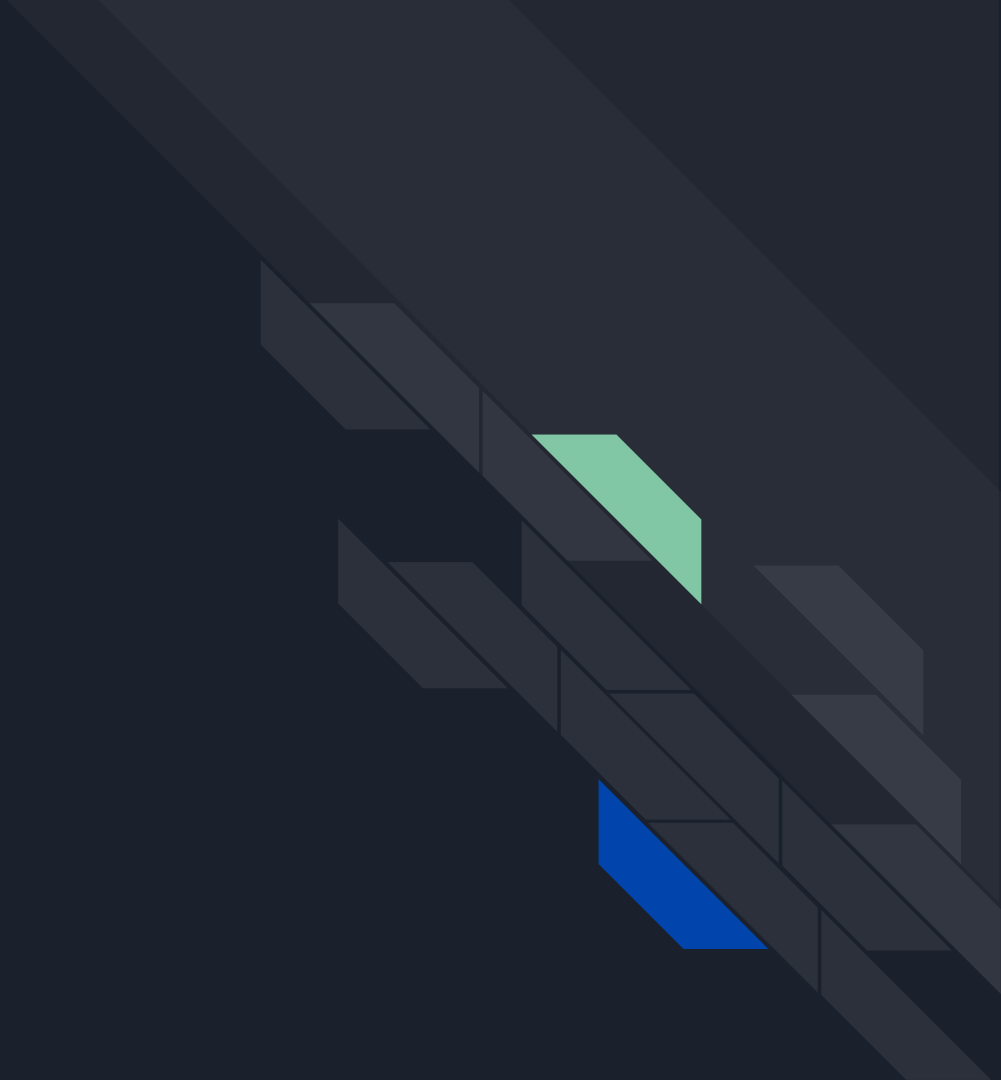
CSE451 20Wi - 4/15/20



Admin

- Lab 1 due tomorrow
- 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!

Locks





Spinlocks

- Disable interrupts and spin until resource is acquired
 - Prevent concurrency issues by not yielding the scheduler until 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
- **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



Synchronization Functions

- Main API for process control: wakeup/sleep
 - Helper: wakeup1 (find all processes sleeping on channel and wake them up)
- Relevant files
 - inc/proc.h
 - kernel/proc.c
- Note that other code may run after wakeup but before process is scheduled
 - If you want to guarantee some condition before proceeding, need something like this:
while (!condition)
 sleep(channel, &mylock);
- Channel is an arbitrary number



Sleeplocks

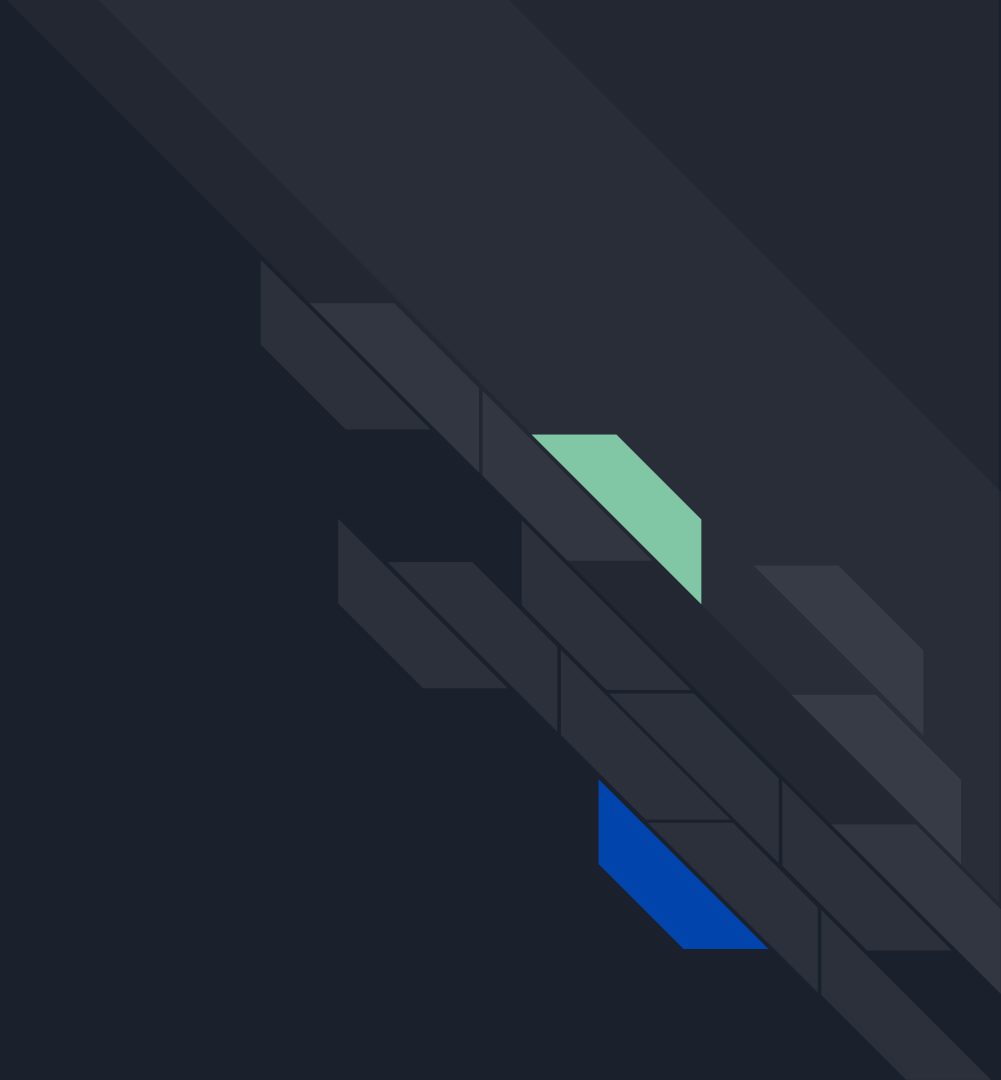
- Uses the sleep/wakeup interface from the previous slide, with `&lock` as channel
- On “acquiresleep”, if the resource is unavailable it will sleep on `&lock`
 - Sets state of process to ASLEEP: it will not be scheduled until awoken
- On “releasesleep”, the code with the sleeplock will wakeup all process waiting on `&lock`.
 - Sets all processes sleeping on `&lock` to RUNNABLE: they can be scheduled
 - All will wake up. One will acquire the lock, the rest will go back to sleep.
- 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



Curious about locks, still?

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


Lab 2





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
- 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 dupe virtual memory space)
 - Data to return to the correct place (trap frame)
 - Anything else?
- Which register holds the return value?
 - Modify this to return in different ways



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 reclaims resources consumed by it
- What are some edge cases to consider?
 - Wait should return child data EVEN IF the child exited before wait() was called
 - 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 not unreasonable



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



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

- loadcode and initstack may be helpful for initializing a new memory space
- use vspacewritetova to export data to a page table that isn't currently installed
- once the memory space is ready, use vspaceinstall(myproc()); to engage



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.



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

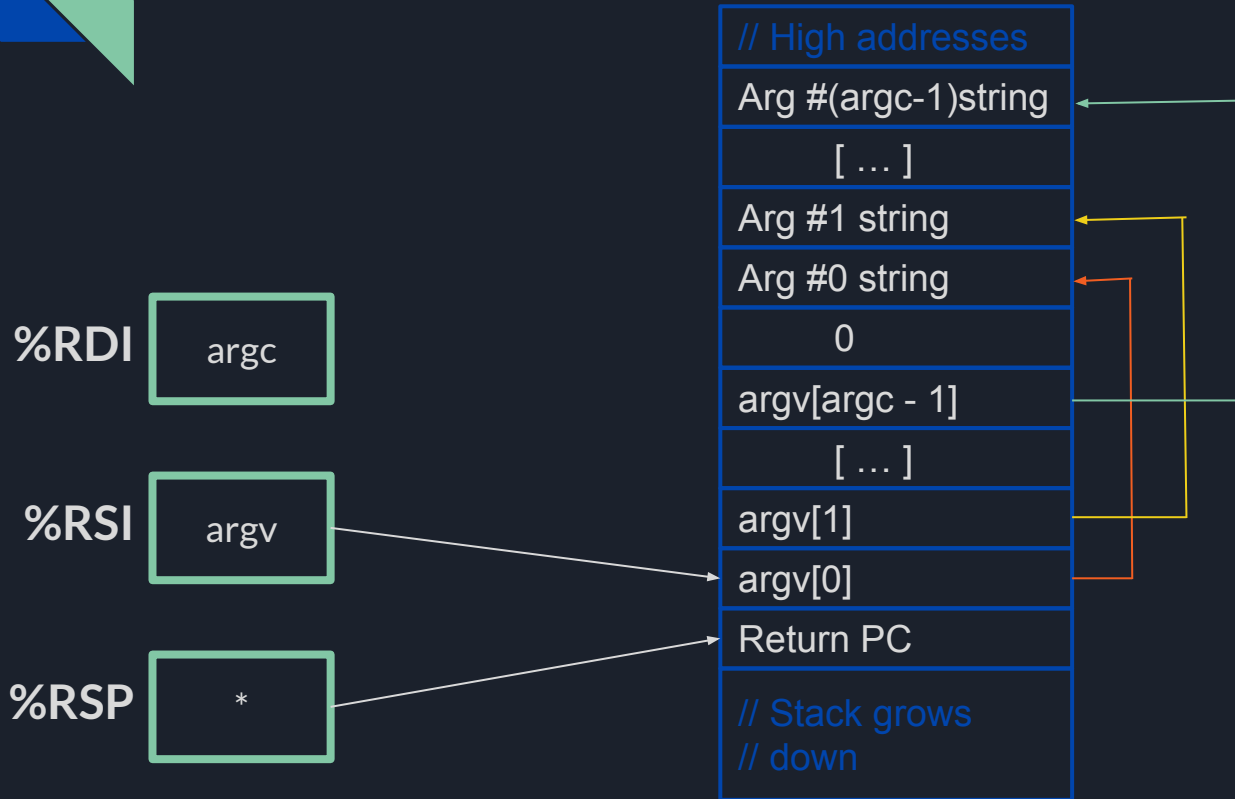
```
int main(int argc, char** argv)
```

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

Practice Exercise 1

%RDI

???

%RSI

???

%RSP

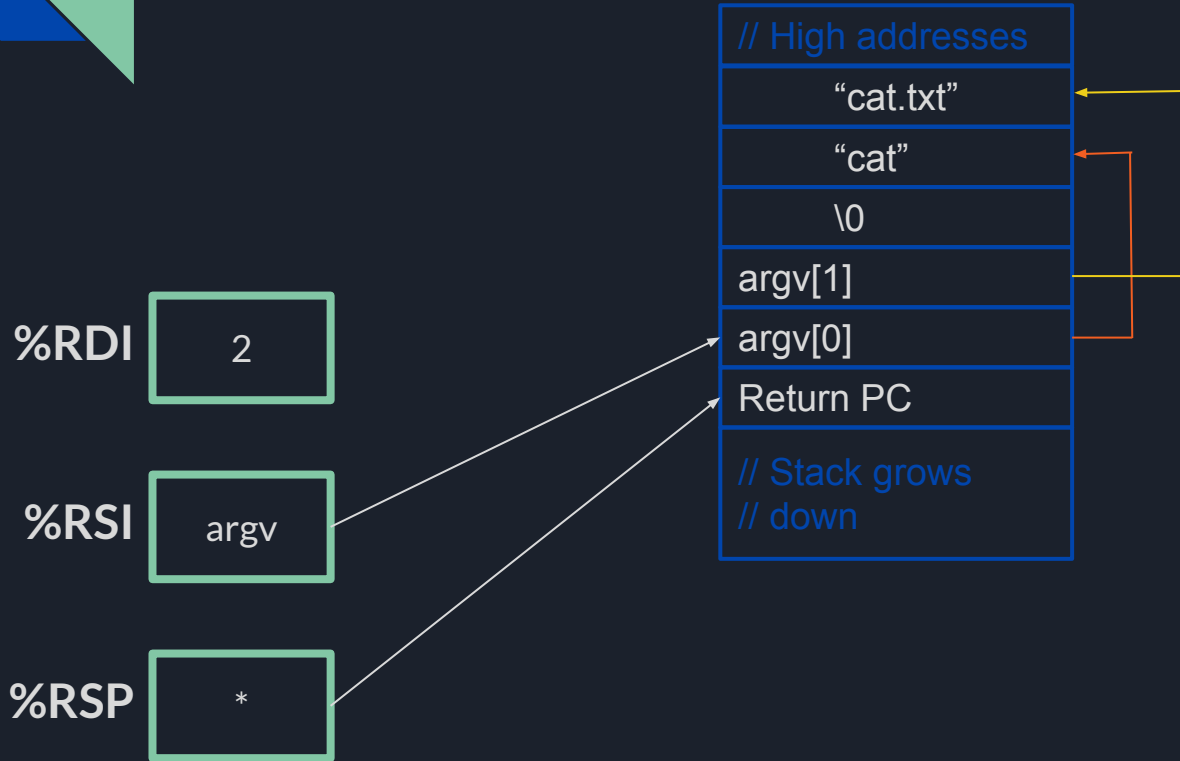
???

// 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

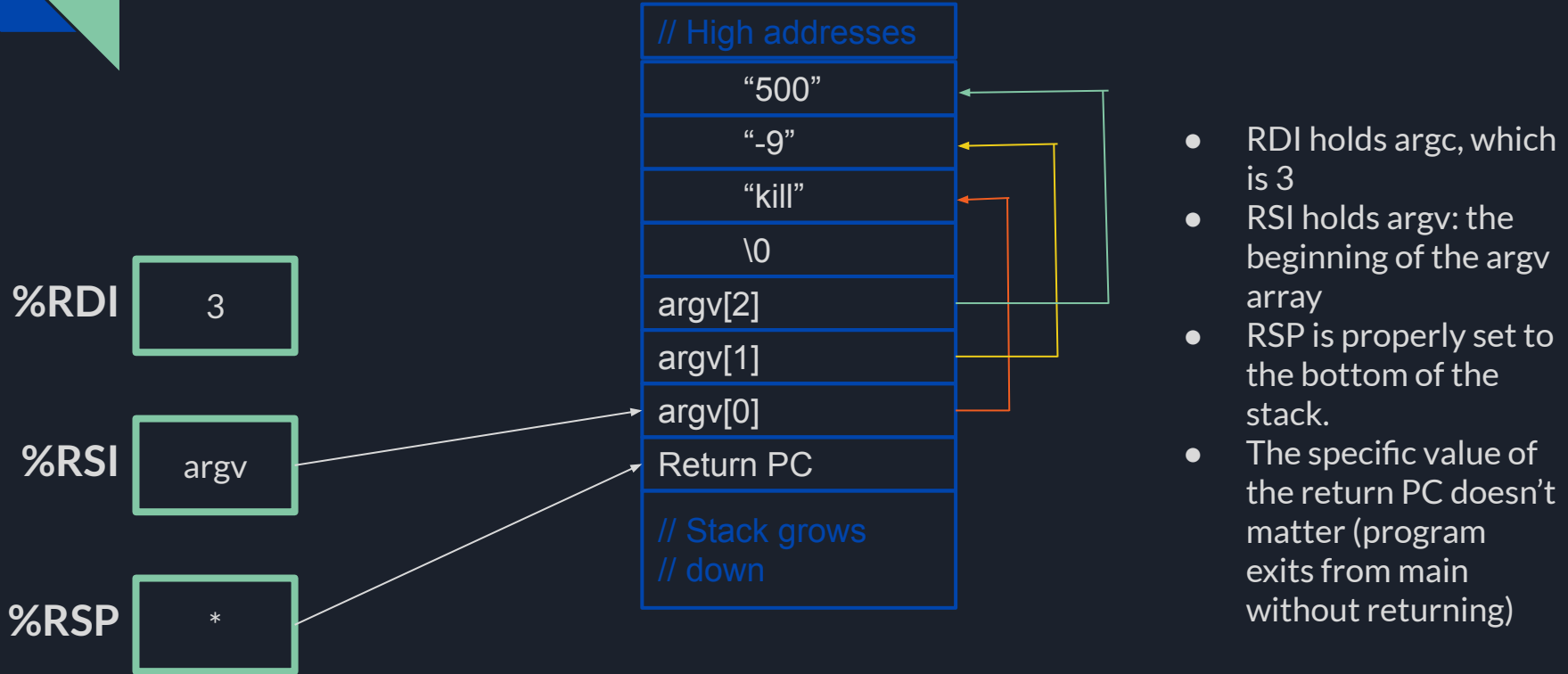
???

// 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)



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
- 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



Design Document

It will be due [Date TBD]

Do it BEFORE you write code

This is mainly for you to plan how you want to implement things before implementing them
Whatever design will help you succeed on implementation should be here

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!

Office hours are a good time to talk about design