Lab 2

Multiprocessing

Admin

- Lab 2 out!
  - Design Doc due 10/21/24 @11:59pm
  - Lab Code + Questions due 10/28/24 @11:59pm
- You will write design docs starting with lab 2
  - The better you fill it out, the more helpful we can be with our feedback!
  - Graded on effort basis, NOT correctness (although correctness would be nice)
- Pset 2 Out
  - Due 10/18/24 @11:59pm

Agenda

What we'll cover today:

1. Design Docs & Debugging Tips
2. Locks
3. Lab 2 - Filesystem Synchronization
4. Lab 2 - Processes

Note: this section does not cover all of Lab 2 content. Please come to Dara's lecture on Monday to get familiar with the second half of Lab 2!

Design Document

- Do it BEFORE you write code
  - This is mainly for you to think carefully before implementing the spec
  - Include whatever design choices that will help you succeed
  - The questions should guide your design decisions
- Knowing what to include is difficult
  - You'll learn as the quarter goes
  - Edge cases, unanswered questions

Office hours are a good time to talk about design!

Quick Note: Debugging

Throughout the quarter, you will most likely run into many errors and bugs…

How do we debug Trap errors and Panics?

- GDB
- Debugging.md is a good place to look!
  - Please look through the doc and debug as much as you can before OH

Debugging Tips: Trap Errors

- Trap Errors
  - `unexpected trap 14 from cpu 0 rip ffffffff80102f27 (cr2=0x0)`
  - `trap 14`: page fault, invalid memory access (most of the time)
  - `rip ffffffff80102f27`: line of code caused the page fault
  - `cr2=0x0`: the memory address that caused the page fault

```
(gdb) info line *0xffffffff80102f27
Line 41 of "kernel/sysfile.c"
   starts at address 0xffffffff80102f23 <sys_write+85>
   and ends at 0xffffffff80102f2d <sys_write+95>.
```

```
40    int *a = NULL;
41    *a = 4;
```

Locks

Why do we need locks?

Why locks?

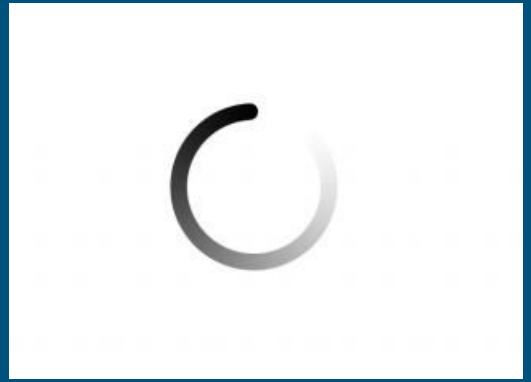
Race conditions threaten data integrity!!

- Recall 333

And in lab 2 part 1: you're supporting multiprocessing => concurrent execution in the kernel!

Spinlocks

- Busy waits until lock can be acquired
  - acquire(): while (can't acquire lock) { ; }
  - release(): marks lock as free by setting lock_status to 0
- Relevant files
  - inc/spinlock.h, kernel/spinlock.c
- Pros:
  - Fast to acquire resource once it’s freed up
- Cons:
  - Wastes CPU while waiting, worse with more waiting threads
  - Interrupts are disabled while holding a spinlock



Sleeplocks

- Sleeps until lock can be acquired
  - acquire(): while (lock is busy) { sleep() }
  - release(): set lock as free, wakeup() ALL sleeping processes for this lock
    - It's a competition! Only 1 process gets the lock, the rest goes 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, incurring overhead

When to use which?

Lab Advice: Spinlocks vs. Sleeplocks

Use **spinlocks** for:

- Protecting scheduler data structures (can’t go to sleep if scheduler is busy)
- Very short, deterministic critical sections (be careful to avoid deadlock)
- Any shared data structure used by interrupt handler (real time responsiveness)

Use **sleeplocks** for:

- I/O operations (readi, writei, file system calls)
- long critical sections (allocating and copying memory)

Lab 2- Synchronization

Synchronization

- Lab1: single-process
  - Initial kernel thread + user init process
  - ⇒ Didn’t need synchronization!
- Lab2: multi-process
  - Support fork() ⇒ multiprocessing
  - Need to revisit previous syscalls and protect global variables accessed
  - **What might those be?**

Lab2: Synchronization

- How are you protecting access to the global open file table?
- How are you making sure two readers using the same file will update its offset correctly?

Note that you don't want your locking scheme to only allow for one process to use the file system at a time. Processes operating (read/stat) on different files should be able to make progress concurrently.

Lab 2 - Processes

What’s lab 2 for?

You’re adding multiprocessing!

How will I enable multiprocessing?

By *implementing* the following syscalls:

- fork
- wait
- exit

Using these *provided* functions in xk:

- sleep
- wakeup

fork()

- Create a new process by duplicating the calling process
  - returns 0 in the child (newly created) process
  - returns child PID in the parent
- What does this entail? What needs to be created, and how do we copy parent state?

fork()

- Create a new process by duplicating the calling process
  - returns 0 in the child (newly created) process
  - returns child PID in the parent
- What does this entail? What needs to be created, and how do we copy parent state?
  - Create an entry in the process table (allocproc)
  - Clone all open resources
    - Files (make sure to increase reference count)
    - All memory (look into vspaceinit and vspacecopy to copy virtual memory space)
    - Copy over the trap frame
    - Anything else? How do we return 0 from the child?

exit()

- exit(): Halts program and sets state to have its resources reclaimed
  - Should clean up resources as much as possible (e.g. close all open files)
  - Let your parent know you've exited (how?)
    - Hint: update process state and wake up parent (be careful with synchronization!)
  - Should not return (How? Hint: look through proc.c functions)

wait()

- wait(): Sleep until a child process terminates, then return that child’s PID.
  - Can only wait for child
  - Need to reclaim child's kernel resources
    - Child's PCB, vspace (vspace destroy), kernel stack
      - erase parent/child relationship
    - Why can't these be freed by the child?
  - Process shouldn’t return from here until a child has exited
  - **Process shouldn't block if any child already exited**

wait() & exit() Relationship

- Parent cleans up child's data on wait(), but parent may not ever call wait
  - Who should clean up the child then?
    - what needs to be done for the adoption to happen?
    - init process calls wait in a loop
  - Keep in mind when you implement exit, you can be both a parent and a child!
- All of lab2 tests rely on wait and exit, so you won't pass any tests until wait and exit are implemented

Wait/Exit Synchronization: How?

"Ok… so I'm starting to understand these system calls… But the spec mentioned that wait interacts with exit and fork. How do I support this?"
~ You, a well-intentioned student

Provided Synchronization API

- Sleep/Wakeup
  - Main API for synchronization in xk
  - How do we know what a process is sleeping on or waking up for?

```
// Per-process state
struct proc {
  struct vspace vspace;      // Virtual address space descriptor
  char *kstack;              // Kernel stack
  enum procstate state;      // Process state
  int pid;                   // Process ID
  struct proc *parent;       // Parent process
  struct trap_frame *tf;     // Trap frame for current syscall
  struct context *context;   // swtch() here to run process
  void *chan;                // If non-zero, sleeping on chan
  int killed;                // If non-zero, have been killed
  char name[16];             // Process name (debugging)
};
```

Provided Synchronization API

- Sleep/Wakeup
  - Main API for synchronization in xk
  - How do we know what a process is sleeping on or waking up for?
  - **chan**: just an arbitrary value (void*), can be anything
    - Processes coordinate sleeping and waking up with an agreed-upon value called chan

Sleep, Wakeup, and Chan

- sleep(void* chan, struct spinlock* lk)
  - atomically release your current lock and grabs the process table lock
  - sets myproc()->state to SLEEPING
  - sets myproc()->chan to whatever channel we are waiting on
  - yields so that scheduler can run another process

Sleep, Wakeup, and Chan

- wakeup(void* chan)
  - acquires the process table lock
  - looks for all SLEEPING processes with the given channel (chan)
    - sets each proc->state to RUNNABLE (ready)
    - proc->chan is also cleared to NULL
- Relevant files:
  - inc/proc.h
  - kernel/proc.c

Chan Example

Parent Process
Status: running

Child Process
Status: runnable

The parent process is currently running, child’s state is runnable

Chan Example

Parent Process
Status: sleeping
Chan: 0xdeadbeef

Process 2
Status: running

Parent process calls wait() and sees that it currently has children, yet no zombie children. Parent goes to sleep on some channel related to this condition (chan is agreed-upon value). Process 2 gets scheduled to run.

Chan Example

Wake up all processes sleeping on 0xdeadbeef!

Parent Process
Status: sleeping
Chan: 0xdeadbeef

Child Process
Status: zombie

Child process calls exit() and sets its state to ZOMBIE. It now needs to wake up up its parent that is currently sleeping on the chan.

Chan Example

Parent Process
Status: runnable

Child Process
Status: zombie

Parent’s now runnable, child process yields processor

Chan Example

Parent Process
Status: running

Child Process
Status: unused

Parent process is eventually scheduled and checks for zombie children. It now finds a zombie child and can successfully clean up its resources!

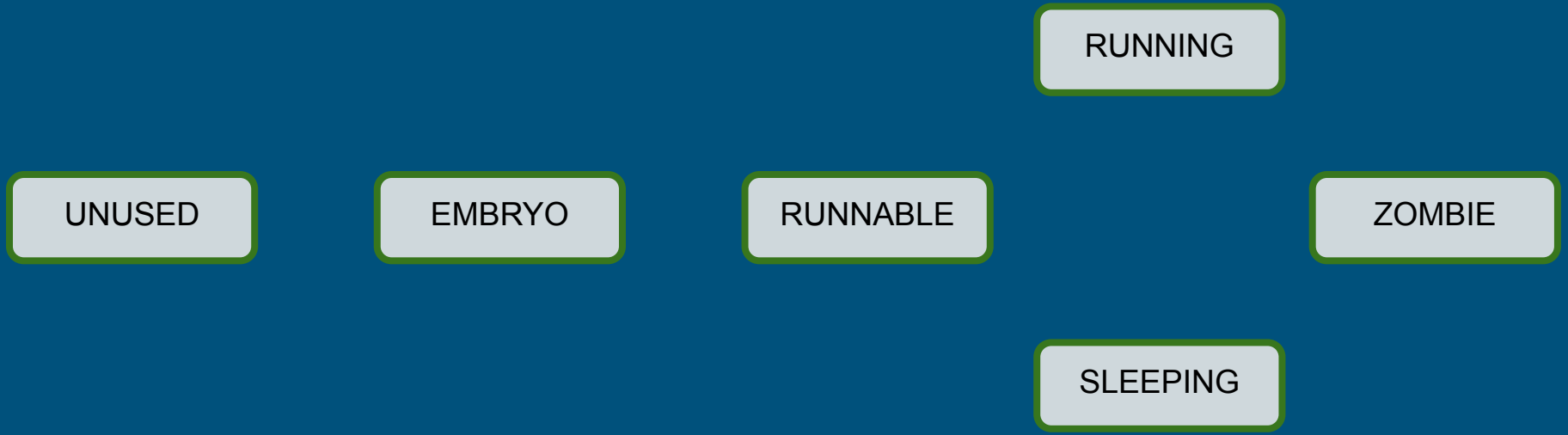
Process States: Transition Flow

“Huh, there seem to be so many process states! So how exactly do we transition between each state?”
~ You, a well-intentioned student

Process States

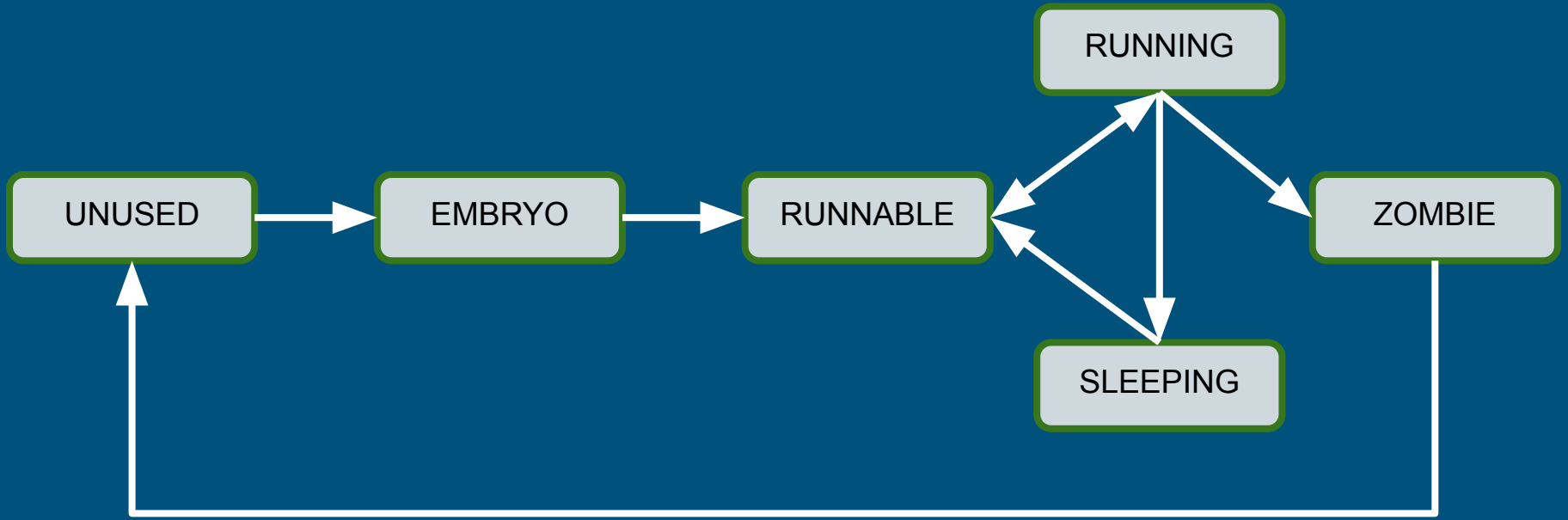
Process States

- Fill out the process state diagram below. Draw arrows from one state to another with the action that would result in that transition



Process States

- Fill out the process state diagram below. Draw arrows from one state to another with the action that would result in that transition



Autograder Tips

- Autograder runs each test individually and then all part1/part2 tests
- part1 and part2 tests are run with make ICOUNT=2/4/6/8/10
  - ICOUNT is an argument to the Makefile
    - should make your bug show up more consistently (per configuration)
    - vary the amount of instruction interleaving (with different icount values)
    - ICOUNT is default to 10 when you run make qemu
  - If your kernel fails on certain ICOUNT config, you can reproduce it locally with make qemu ICOUNT=2/4/6/8/10 to debug