Lab 2

Multiprocessing

Admin

- Lab 2 has 2 parts with separate design docs and due dates
  - part 1 design due 10/20 (next Friday, no late days) so we can give you timely feedback
  - part 1 code due 10/27 (with late days)
  - part 2 design due 10/27 (no late days)
  - part 2 code due 11/3
- You will write design docs starting with lab 2!
  - The better you fill it out, the more helpful we can be with our feedback!
  - Submit on Gradescope but also put design doc under repo/lab, 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 the spec
    Include whatever design choices that will help you succeed
- Knowing what to include is difficult
  - You'll learn as the quarter goes
  - Use designdoc & 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

- Busy waits until lock can be acquired
  - acquire(): while (can't acquire lock) { ; }
    - xchg(lock_status, 1) == 1 ⇒ can't acquire the lock
    - xchg ≈ test&set, lets you atomically exchange any value and returns the old value
  - release(): marks lock as free by setting lock_status to 0
- Relevant files
  - inc/spinlock.h
  - kernel/spinlock.c
- Pros/Cons of spinlocks?
  - Fast to acquire resource once it’s freed up
  - Wastes CPU while waiting, worse with more waiting threads

Sleeplocks

- Sleeps until lock can be acquired
  - acquire(): while (lock is busy) { sleep() }
  - release(): set lock as free, wakeup() ALL sleeping threads for this lock
    - It's a competition! Only 1 thread 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

Lab Advice: Spinlocks vs. Blocking Locks

Use spinlock 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)
  - in xk, interrupt handlers are the trap switch cases with a lapiceoi() call at the end

Blocking locks

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

Lab 2- Synchronization

Synchronization

- Lab1: initial kernel thread + user init process
  - only one process to make system calls
  - no need for synchronization for any global data used by syscalls
- Lab2: 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

Lab2: Fork, Wait, Exit

- Fork: return twice
  - parent resumes execution by restoring registers saved in the trapframe
  - child can "resume" in a similar fashion
  - the only difference is their return value, which is stored in ??
- Wait:
  - if children already exited, no blocking needed
    - how do you tell whether a child has exited? does the check need protection?
- Exit:
  - exit as a parent = pass your children to someone else
    - why can't we do this the other way around? can child check whether its parent has exited?

Sleep, Wakeup, and Chan

- sleep/wakeup
  - main API for synchronization in xk
  - how do we know what a process is sleeping on or waking up for?
  - **chan**: just a pointer, can be anything
    - in sleeplock's case this would be the address of the lock
- sleep(void* chan, struct spinlock* lk)
  - sets myproc()->state to SLEEPING
  - sets myproc()->chan to whatever channel we are waiting on
  - atomically release your current lock and grabs the process table lock
  - 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

Linux: keeps a linked list of threads waiting on a chan (more efficient)

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?
  - 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)
    - Return execution flow to the correct place with the correct context (trap frame)
    - Anything else?

wait()/exit()

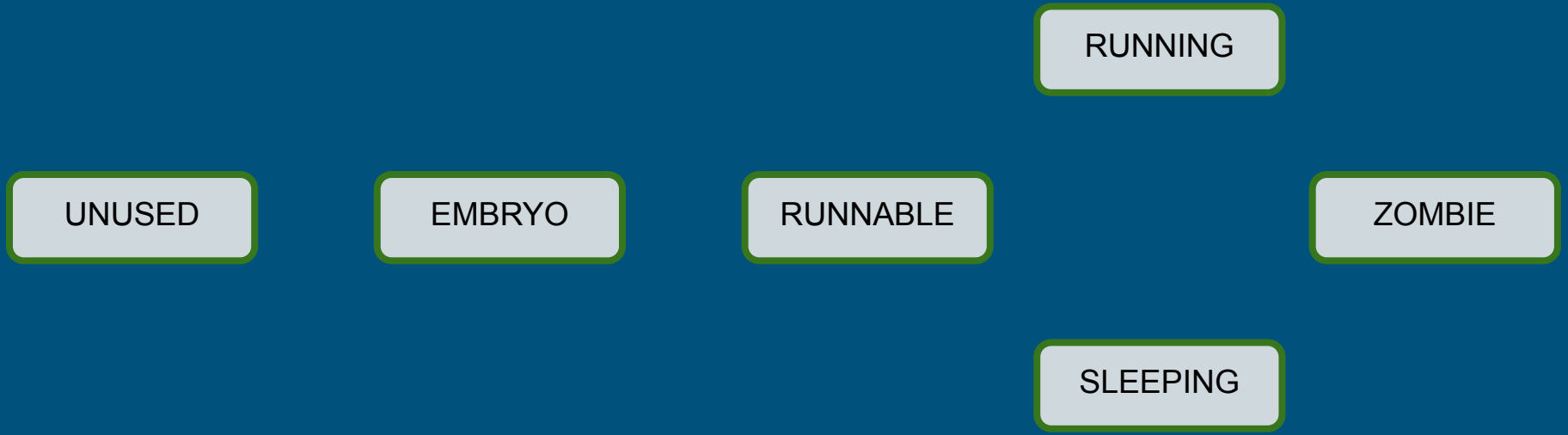
- exit(): Halts program and sets state to have its resources reclaimed
  - should clean up as much resources as possible (eg. close all open files)
  - let your parent know you've exited (how?)
    - hint: update process state and wake up parent
  - should not return (how? hint: look through proc.c functions)
- 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
    - 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()

- 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!
- Almost all of lab2 tests rely on wait and exit, so you won't pass any tests until wait and exit are implemented

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