Lab 2

Multiprocessing

Admin

- Lab 2 design due 10/25 (next Tuesday, no late days!)
  - so we can give you timely feedback
- Lab 2 due 11/2 (~two weeks from now)
  - you need all the time
- 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!
  - 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)

Blocking locks

- Everything else

In some places, xk disables interrupts (eg, before acquiring a spinlock). This prevents just this core from taking an interrupt (eg, taking a time slice). Other cores can still execute, take interrupts, etc.

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)

Monitors in xk

- Monitor: main synchronization primitive to coordinate processes
  - implemented using lock, sleep, wakeup, and chan that we just saw
  - a lock + state variables + condition variables
    - in xk, the lock must be a spinlock (an impl. choice)
- State variables
  - variables that track the current state of things, often use to check condition
  - while (state_var1 == 0 && state_var2 == false)
- Condition variables
  - manage waiters for a condition
    - all procs with the same chan are waiters for that condition
  - in xk, CVs = chan + sleep + wakeup

Monitors in xk

- You will use monitors to implement wait(), exit(), pipe() for lab2
- sleep in synch.c is not the sleep system call

sleep = wait
wakeup = broadcast
no equivalent in xk = signal

```c
struct fridge {
  struct spinlock lk; // assume initialized
  int yogurt = 0;
  int strawberry = 0;
}

void make_breakfast(struct fridge* fridge) {
  acquire(&fridge->lk);
  while (fridge->yogurt == 0 && fridge->strawberry < 2) {
    // temporarily release the lk when we sleep
    // so that the fridge state may be accessed and modified
    // when sleep returns, lk is acquired again (implicitly)
    sleep(fridge, &fridge->lk);
  }
  // consume the yogurt and strawberry
  fridge->yogurt = 0;
  fridge->strawberry -= 2;
  release(&fridge->lk);
}

void fill_fridge(struct fridge* fridge) {
  acquire(&fridge->lk);
  fridge->yogurt += 1;
  fridge->strawberry += 2;
  wakeup(fridge);
  release(&fridge->lk);
}
```

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?

Lab 2 - Processes

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?)
- 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, page tables, 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()

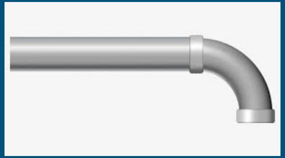
- Parent cleans up child's data on wait(), but parent may not ever call wait
  - Who should clean up the child then?
  - Keep in mind when you implement exit, you can be both a parent and a child!
- Almost all of lab2 tests rely on wait and exit, so you won't pass any tests until wait and exit are implemented

Lab 2 - Pipe

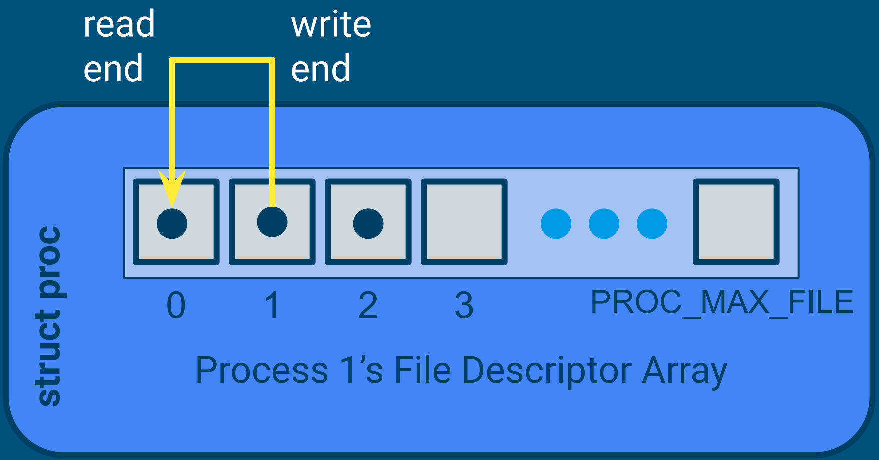
pipe(fds)

- Creates a pipe (kernel buffer) for process to read and write
- From the user perspective: returns two new file descriptors
  - fds[0] = “read end”, not writable
  - fds[1] = “write end”, is not readable
- You’ll want to make this compatible with existing file syscall interface
- Pipe allows processes to communicate with each other
  - parent opens a pipe, forks a child, and now they both have access to the pipe ends
  - typically one process only leaves one end open (closes the read end or the write end)

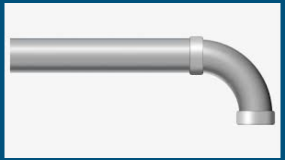
Pipes



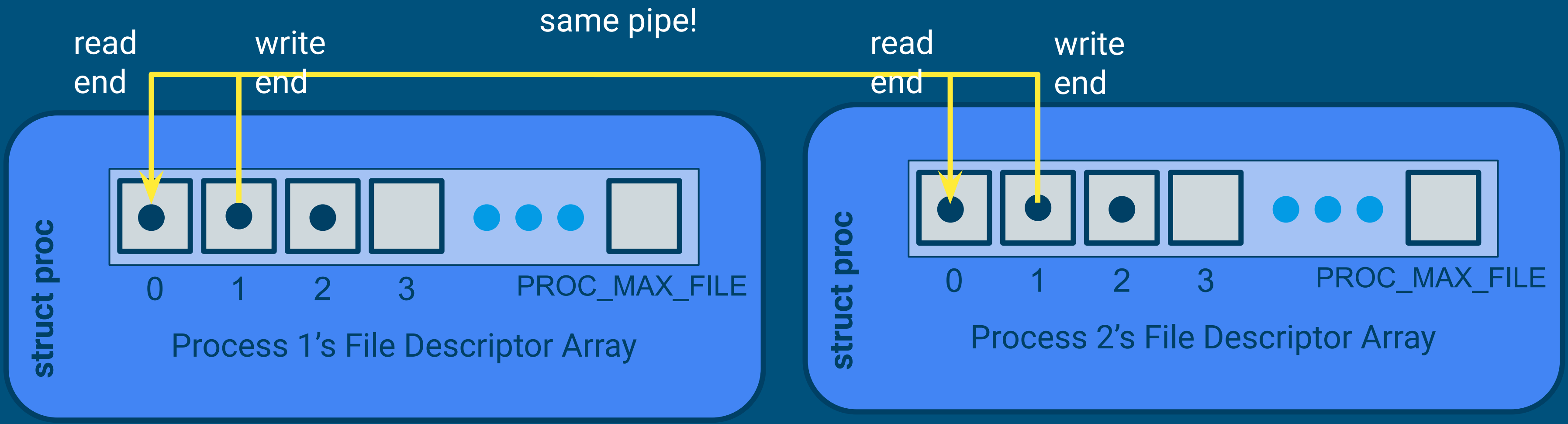
- A mechanism for process communication
- By calling sys_pipe, a process sets up a writing and reading end to a "holding area" where data can be passed between processes



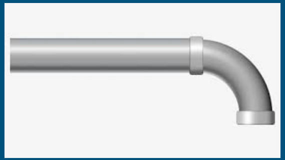
Pipes



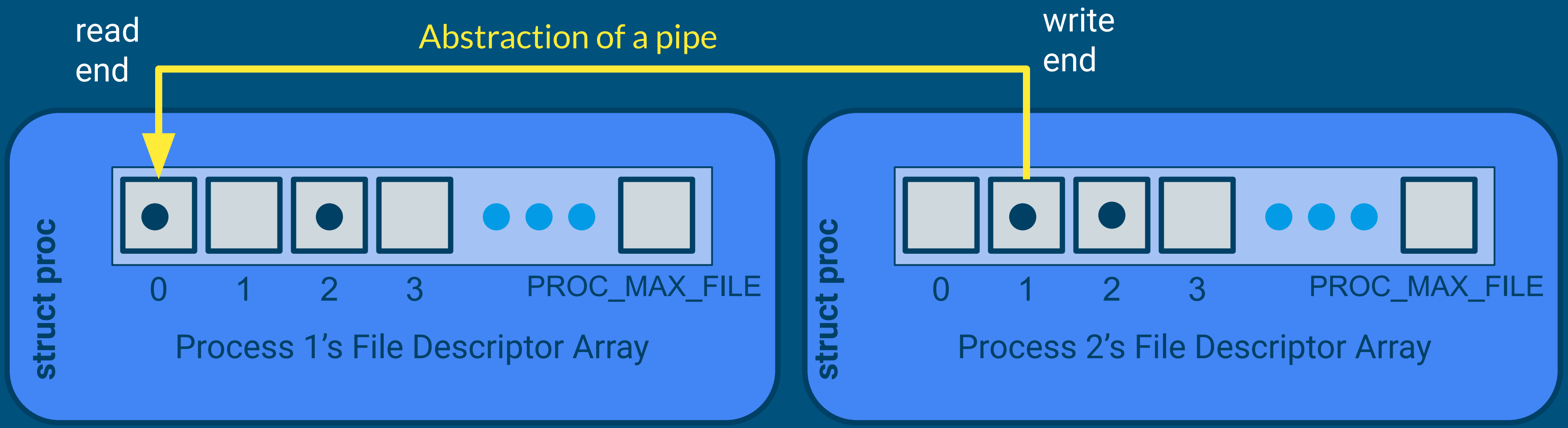
- Process 1 calls fork(), fd table is duplicated



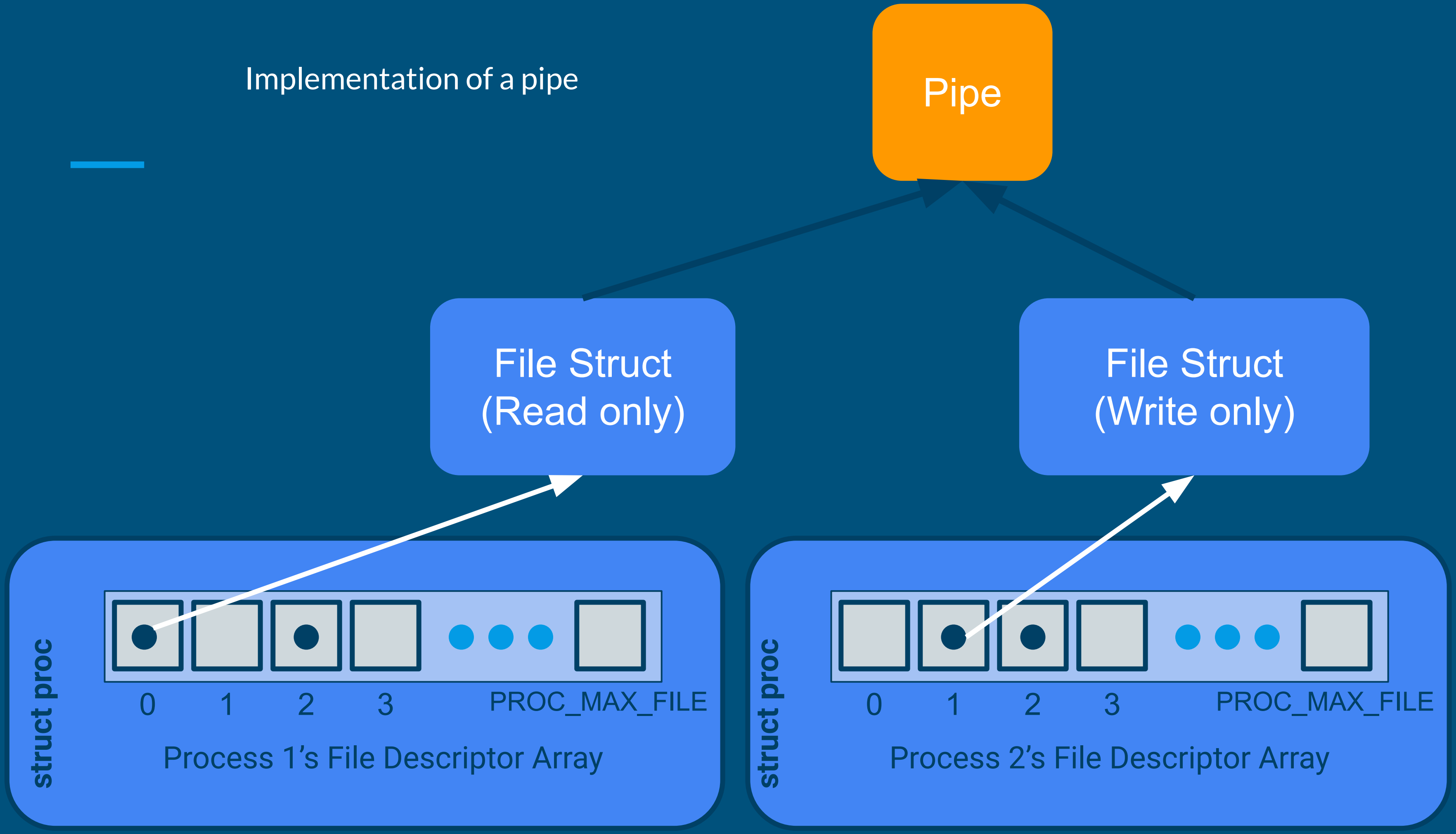
Pipes



- Process 1 close(1), process 2 close(0)
- And now we have a pipe across processes



Implementation of a pipe



Pipes

- Where should pipe be allocated?
  - pipes should be allocated at runtime, as requested
  - how does xk do dynamic memory allocation?
    - (hint: kstack is also dynamically allocated)
- When can you free the pipe and its buffer?
  - remember there may be multiple read ends and write ends
- Can we always write to or read from the buffer? (Hint: bounded buffer sync)
  - What if there's no room to write, or no data to read?
  - What happens if all read/write ends are closed?
- Pipe operations go through file syscall
  - Need a way to determine if a struct file is an inode or a pipe

exec(progname, args)

- Fully replaces the current process; it does not create a new one
- How to replace the current process?
  - need to set up a new virtual address space and new registers states
  - and then switch to using the new VAS and register states
  - file descriptors and pid remain the same

exec(progname, args)

- Setting up a new virtual address space
  - `vspaceinit` for initialization
  - `vspaceloadcode` to load code
  - `vspaceinitstack` to allocate stack vregion
    - you still need to populate user stack with arguments
    - `vspacewritetova` to write data into the stack of the new VAS
  - `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

Questions?