Monitors in xk

- **Lock**
  - xk condition variable API only supports spinlock (an impl. choice)

- **Condition**
  - the shared data that threads are synchronizing on
  - for wait/exit this would be child's state

- **Condition Variable**
  - the waiter list is tracked by the process table
  - proc in SLEEPING state with the same `chan` are part of the same CV
  - `chan` is a pointer, can be anything (think of it as a cv identifier)
Sleep, Wakeup, and Chan

- `sleep(void* chan, struct spinlock* lk)`
  - atomically release your current lock and grabs the process table (ptable) lock
    - if your current lock is the ptable lock do nothing
    - why might your current lock be the ptable 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
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
pipe(fds)

- Creates a pipe (kernel buffer) for process to read and write
- From the user perspective: returns two new file descriptors
  - $\text{fds}[0] = \text{“read end”}, \text{not writable}$
  - $\text{fds}[1] = \text{“write end”}, \text{is not readable}$
- 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

```
read end
write end
same pipe!
```
Pipes

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

Pipe

File Struct (Read only)

File Struct (Write only)

Process 1’s File Descriptor Array

Process 2’s File Descriptor Array
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
Pipes Impl. Tips

- What metadata/information do you need for pipe?
  - offset to read from
  - offset to write to
  - whether the read end is still open
  - whether the write end is still open
  - # of bytes available in the buffer
  - lock and condition variables

- Similar to the bounded buffer problem
Lab 2 - Exec
**exec**(*program, 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(program, 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
exec(program, args): args setup

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
X86_64 Calling Conventions

- %rdi: holds the first argument
- %rsi: holds the second argument
  - %rdx, %rcx, %r8, %r9 comes next
  - overflows (arg7, arg8 …) onto the 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
**Stack For User Process**

- 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.

**Why? Alignment**

**Why NULL pointer?**

**Convention**
Questions?
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
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 *0xfffffffff80102f27
Line 41 of "kernel/sysfile.c"
    starts at address 0xfffffffff80102f23 <sys_write+85>
    and ends at 0xfffffffff80102f2d <sys_write+95>.
```

For more details, check out debugging.md
Debugging Tips: Record & Replay

Starting with lab2, there are multiple processes, meaning more concurrent accesses to the kernel code, which might make bugs harder to reproduce.

`make qemu-record`

record all external events to a log file

helpful if you can record the race condition

`make qemu-gdb-replay`  (pair with `make gdb`)

replay according to the log file, but with gdb (similar to `make qemu-gdb`)