Review: Thread package API

- `tid thread_create (void (*fn) (void *), void *arg);`
  - Create a new thread that calls `fn` with `arg`
- `void thread_exit ();`
- `void thread_join (tid thread);`
- **The execution of multiple threads is interleaved**
- **Can have *non-preemptive* threads:**
  - One thread executes exclusively until it makes a blocking call
- **Or *preemptive* threads (what we usually mean in this class):**
  - May switch to another thread between any two instructions.
- **Using multiple CPUs is inherently preemptive**
  - Even if you don't take CPU_0 away from thread T, another thread on CPU_1 can execute "between" any two instructions of T

Program A

```
int flag1 = 0, flag2 = 0;

void p1 (void *ignored) {
  flag1 = 1;
  if (!flag2) { critical_section_1 (); }
}

void p2 (void *ignored) {
  flag2 = 1;
  if (!flag1) { critical_section_2 (); }
}

int main () {
  tid id = thread_create (p1, NULL);
  p2 ();
  thread_join (id);
}
```

Q: Can both critical sections run?

Program B

```
int data = 0;
int ready = 0;

void p1 (void *ignored) {
  data = 2000;
  ready = 1;
}

void p2 (void *ignored) {
  while (!ready)
    ;
  use (data);
}

int main () { ... }
```

Q: Can `use` be called with value 0?

Program C

```
int a = 0;
int b = 0;

void p1 (void *ignored) {
  a = 1;
}

void p2 (void *ignored) {
  if (a == 1)
    b = 1;
}

void p3 (void *ignored) {
  if (b == 1)
    use (a);
}
```

Q: If `p1–3` run concurrently, can `use` be called with value 0?

Correct answers

[git push slides to web site now]

Correct answers

- **Program A: I don't know**

Correct answers

- **Program A: I don't know**
- **Program B: I don't know**

Correct answers

- **Program A: I don't know**
- **Program B: I don't know**
- **Program C: I don't know**
- **Why don't we know?**
  - It depends on what machine you use
  - If a system provides *sequential consistency*, then answers all No
  - But not all hardware provides sequential consistency
- **Note: Examples, other content from [Adve & Gharachorloo]**
- **Another great reference: Why Memory Barriers**

Outline

1. Memory consistency

2. The critical section problem

3. Mutexes and condition variables

4. Implementing synchronization

5. Alternate synchronization abstractions

Sequential Consistency

> **Definition**
>
> *Sequential consistency*: The result of execution is as if all operations were executed in some sequential order, and the operations of each processor occurred in the order specified by the program.
> – Lamport

- **Boils down to two requirements on loads and stores:**
  1. Maintaining *program order* of on individual processors
  2. Ensuring *write atomicity*

- **Without SC (Sequential Consistency), multiple CPUs can be "worse"—i.e., less intuitive—than preemptive threads**
  - Result may not correspond to *any* instruction interleaving on 1 CPU

- **Why doesn't all hardware support sequential consistency?**

SC thwarts hardware optimizations

- **Complicates write buffers**
  - E.g., read flag$n$ before flag$(3 - n)$ written through in Program A
- **Can't re-order overlapping write operations**
  - Concurrent writes to different memory modules
  - Coalescing writes to same cache line
- **Complicates non-blocking reads**
  - E.g., speculatively prefetch `data` in Program B
- **Makes cache coherence more expensive**
  - Must delay write completion until invalidation/update (Program B)
  - Can't allow overlapping updates if no globally visible order (Program C)

SC thwarts compiler optimizations

- **Code motion**
- **Caching value in register**
  - Collapse multiple loads/stores of same address into one operation
- **Common subexpression elimination**
  - Could cause memory location to be read fewer times
- **Loop blocking**
  - Re-arrange loops for better cache performance
- **Software pipelining**
  - Move instructions across iterations of a loop to overlap instruction latency with branch cost

x86 consistency [intel 3a, §8.2]

- **x86 supports multiple consistency/caching models**
  - Memory Type Range Registers (MTRR) specify consistency for ranges of physical memory (e.g., frame buffer)
  - Page Attribute Table (PAT) allows control for each 4K page
- **Choices include:**
  - **WB**: Write-back caching (the default)
  - **WT**: Write-through caching (all writes go to memory)
  - **UC**: Uncacheable (for device memory)
  - **WC**: Write-combining – weak consistency & no caching (used for frame buffers, when sending a lot of data to GPU)
- **Some instructions have weaker consistency**
  - String instructions (written cache-lines can be re-ordered)
  - Special "non-temporal" store instructions (`movnt*`) that bypass cache and can be re-ordered with respect to other writes

x86 WB consistency

- **Old x86s (e.g, 486, Pentium 1) had almost SC**
  - Exception: A read could finish before an earlier write to a different location
  - Which of Programs A, B, C might be affected?

x86 WB consistency

- **Old x86s (e.g, 486, Pentium 1) had almost SC**
  - Exception: A read could finish before an earlier write to a different location
  - Which of Programs A, B, C might be affected?   *Just A*
- **Newer x86s also let a CPU read its own writes early**

```
volatile int flag1;          volatile int flag2;

int p1 (void)                int p2 (void)
{                            {
  register int f, g;           register int f, g;
  flag1 = 1;                   flag2 = 1;
  f = flag1;                   f = flag2;
  g = flag2;                   g = flag1;
  return 2*f + g;              return 2*f + g;
}                            }
```

  - E.g., *both* p1 and p2 can return 2:
  - Older CPUs would wait at "`f = ...`" until store complete

x86 atomicity

- `lock` **prefix makes a memory instruction atomic**
  - Historically locked bus for duration of instruction (expensive!)
  - Now requires exclusively caching memory, synchronizing with other memory operations
  - All lock instructions totally ordered
  - Other memory instructions cannot be re-ordered with locked ones
- `xchg` **instruction is always locked (even without prefix)**
- **Special barrier (or "fence") instructions can prevent re-ordering**
  - `lfence` – can't be reordered with reads (or later writes)
  - `sfence` – can't be reordered with writes (e.g., use after non-temporal stores, before setting a *ready* flag)
  - `mfence` – can't be reordered with reads or writes

Outline

Assuming sequential consistency

- **Often we reason about concurrent code assuming SC**
- **But for low-level code, know your memory model!**
  - May need to sprinkle barrier/fence instructions into your source
  - Or may need compiler barriers to restrict optimization
- **For most code, avoid depending on memory model**
  - Idea: If you obey certain rules (discussed later) ...system behavior should be indistinguishable from SC
- **Let's for now say we have sequential consistency**
- **Example concurrent code: Producer/Consumer**
  - `buffer` stores `BUFFER_SIZE` items
  - `count` is number of used slots
  - `out` is next empty buffer slot to fill (if any)
  - `in` is oldest filled slot to consume (if any)

```
void producer (void *ignored) {
    for (;;) {
        item *nextProduced = produce_item ();
        while (count == BUFFER_SIZE)
            /* do nothing */;
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
    }
}

void consumer (void *ignored) {
    for (;;) {
        while (count == 0)
            /* do nothing */;
        item *nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        consume_item (nextConsumed);
    }
}
```

Q: What can go wrong in above threads (even with SC)?

Data races

- count **may have wrong value**
- **Possible implementation of** count++ **and** count--

  register←count   register←count
  register←register + 1   register←register − 1
  count←register   count←register
- **Possible execution (count one less than correct):**

  register←count
  register←register + 1

      register←count
      register←register − 1

  count←register

      count←register

Data races (continued)

- **What about a single-instruction add?**
  - E.g., i386 allows single instruction addl $1,_count
  - So implement count++/-- with one instruction
  - Now are we safe?

Data races (continued)

- **What about a single-instruction add?**
  - E.g., i386 allows single instruction addl $1,_count
  - So implement count++/-- with one instruction
  - Now are we safe? Not on multiprocessors!
- **A single instruction may encode a load and a store operation**
  - S.C. doesn't make such *read-modify-write* instructions atomic
  - So on multiprocessor, suffer same race as 3-instruction version
- **Can make x86 instruction atomic with** lock **prefix**
  - But lock potentially very expensive
  - Compiler assumes you don't want penalty, doesn't emit it
- **Need solution to** *critical section* **problem**
  - Place count++ and count-- in critical section
  - Protect critical sections from concurrent execution

Desired properties of solution

- *Mutual Exclusion*
  - Only one thread can be in critical section at a time
- *Progress*
  - Say no process currently in critical section (C.S.)
  - One of the processes trying to enter will eventually get in
- *Bounded waiting*
  - Once a thread *T* starts trying to enter the critical section, there is a bound on the number of times other threads get in
- **Note progress vs. bounded waiting**
  - If no thread can enter C.S., don't have progress
  - If thread *A* waiting to enter C.S. while *B* repeatedly leaves and re-enters C.S. *ad infinitum*, don't have bounded waiting

Peterson's solution

- **Still assuming sequential consistency**
- **Assume two threads,** T_0 **and** T_1
- **Variables**
  - int not_turn; // not this thread's turn to enter C.S.
  - bool wants[2]; // wants[i] indicates if T_i wants to enter C.S.
- **Code:**

```
for (;;) { /* assume i is thread number (0 or 1) */
  wants[i] = true;
  not_turn = i;
  while (wants[1-i] && not_turn == i)
    /* other thread wants in and not our turn, so loop */;
  Critical_section ();
  wants[i] = false;
  Remainder_section ();
}
```

Does Peterson's solution work?

```
for (;;) { /* code in thread i */
  wants[i] = true;
  not_turn = i;
  while (wants[1-i] && not_turn == i)
    /* other thread wants in and not our turn, so loop */;
  Critical_section ();
  wants[i] = false;
  Remainder_section ();
}
```

- **Mutual exclusion – can't both be in C.S.**
  - Would mean `wants[0] == wants[1] == true`,
    so `not_turn` would have blocked one thread from C.S.
- **Progress – given demand, one thread can always enter C.S.**
  - If T_{1-i} doesn't want C.S., `wants[1-i] == false`, so T_i won't loop
  - If both threads want in, one thread is not the `not_turn` thread
- **Bounded waiting – similar argument to progress**
  - If T_i wants lock and T_{1-i} tries to re-enter, T_{1-i} will set
    `not_turn = 1 - i`, allowing T_i in

Outline

1. Memory consistency

2. The critical section problem

3. Mutexes and condition variables

4. Implementing synchronization

5. Alternate synchronization abstractions

Mutexes

- **Peterson expensive, only works for 2 processes**
  - Can generalize to *n*, but for some fixed *n*
- **Must adapt to machine memory model if not SC**
  - If you need machine-specific barriers anyway, might as well take
    advantage of other instructions helpful for synchronization
- **Want to insulate programmer from implementing
  synchronization primitives**
- **Thread packages typically provide *mutexes*:**
  ```
  void mutex_init (mutex_t *m, ...);
  void mutex_lock (mutex_t *m);
  int mutex_trylock (mutex_t *m);
  void mutex_unlock (mutex_t *m);
  ```
  - Only one thread acquires `m` at a time, others wait

Thread API contract

- **All global data should be protected by a mutex!**
  - Global = accessed by more than one thread, at least one write
  - Exception is initialization, before exposed to other threads
  - This is the responsibility of the application writer
- **If you use mutexes properly, behavior should be
  indistinguishable from Sequential Consistency**
  - This is the responsibility of the threads package (& compiler)
  - Mutex is broken if you use properly and don't see SC
- **OS kernels also need synchronization**
  - Some mechanisms look like mutexes
  - But interrupts complicate things (incompatible w. mutexes)

Same concept, many names

- **Most popular application-level thread API: Pthreads**
  - Function names in this lecture all based on Pthreads
  - Just add `pthread_` prefix
  - E.g., `pthread_mutex_t`, `pthread_mutex_lock`,...
- **C11 uses `mtx_` instead of `mutex_`, C++11 uses methods on `mutex`**
- **Pintos uses `struct lock` for mutexes:**
  ```
  void lock_init (struct lock *);
  void lock_acquire (struct lock *);
  bool lock_try_acquire (struct lock *);
  void lock_release (struct lock *);
  ```
- **Extra Pintos feature:**
  - Release checks that lock was acquired by same thread
  - `bool lock_held_by_current_thread (struct lock *lock);`

Improved producer

```
mutex_t mutex = MUTEX_INITIALIZER;

void producer (void *ignored) {
    for (;;) {
        item *nextProduced = produce_item ();

        mutex_lock (&mutex);
        while (count == BUFFER_SIZE) {
          mutex_unlock (&mutex);
          thread_yield ();
          mutex_lock (&mutex);
        }

        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        mutex_unlock (&mutex);
    }
}
```

Improved consumer

```
void consumer (void *ignored) {
    for (;;) {
        mutex_lock (&mutex);
        while (count == 0) {
          mutex_unlock (&mutex); /* <--- Why? */
          thread_yield ();
          mutex_lock (&mutex);
        }

        item *nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        mutex_unlock (&mutex);

        consume_item (nextConsumed);
    }
}
```

Condition variables

- **Busy-waiting in application is a bad idea**
  - Consumes CPU even when a thread can't make progress
  - Unnecessarily slows other threads/processes or wastes power
- **Better to inform scheduler of which threads can run**
- **Typically done with *condition variables***
- struct cond_t;   (pthread_cond_t **or** condition **in Pintos**)
- void cond_init (cond_t *, ...);
- void cond_wait (cond_t *c, mutex_t *m);
  - Atomically unlock m and sleep until c signaled
  - Then re-acquire m and resume executing
- void cond_signal (cond_t *c);
  void cond_broadcast (cond_t *c);
  - Wake one/all threads waiting on c

Improved producer

```
mutex_t mutex = MUTEX_INITIALIZER;
cond_t nonempty = COND_INITIALIZER;
cond_t nonfull = COND_INITIALIZER;

void producer (void *ignored) {
    for (;;) {
        item *nextProduced = produce_item ();

        mutex_lock (&mutex);
        while (count == BUFFER_SIZE)
          cond_wait (&nonfull, &mutex);

        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        cond_signal (&nonempty);
        mutex_unlock (&mutex);
    }
}
```

Improved consumer

```
void consumer (void *ignored) {
    for (;;) {
        mutex_lock (&mutex);
        while (count == 0)
          cond_wait (&nonempty, &mutex);

        item *nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        cond_signal (&nonfull);
        mutex_unlock (&mutex);

        consume_item (nextConsumed);
    }
}
```

Re-check conditions

- **Always re-check condition on wake-up**
  ```
  while (count == 0) /* not if */
    cond_wait (&nonempty, &mutex);
  ```

- **Otherwise, breaks with spurious wakeup or two consumers**
  - Start where Consumer 1 has mutex but buffer empty, then:

Consumer 1	Consumer 2	Producer
cond_wait (...);		mutex_lock (...);
		⋮
		count++;
		cond_signal (...);
	mutex_lock (...);	mutex_unlock (...);
	if (count == 0)	
	⋮	
	use buffer[out] ...	
	count--;	
	mutex_unlock (...);	
use buffer[out] ... ⟵ No items in buffer		

Condition variables (continued)

- **Why must cond_wait both release mutex & sleep?**
- **Why not separate mutexes and condition variables?**
  ```
  while (count == BUFFER_SIZE) {
    mutex_unlock (&mutex);
    cond_wait (&nonfull);
    mutex_lock (&mutex);
  }
  ```

Condition variables (continued)

- **Why must** `cond_wait` **both release mutex & sleep?**
- **Why not separate mutexes and condition variables?**
  ```
  while (count == BUFFER_SIZE) {
    mutex_unlock (&mutex);
    cond_wait (&nonfull);
    mutex_lock (&mutex);
  }
  ```
- **Can end up stuck waiting when bad interleaving**

Producer	Consumer
`while (count == BUFFER_SIZE)` `mutex_unlock (&mutex);`	
	`mutex_lock (&mutex);` `...` `count--;` `cond_signal (&nonfull);`
`cond_wait (&nonfull);`	

- **Problem:** `cond_wait` **&** `cond_signal` **do not commute**

Other thread package features

- **Alerts – cause exception in a thread**
- **Timedwait – timeout on condition variable**
- **Shared locks – concurrent read accesses to data**
- **Thread priorities – control scheduling policy**
  - Mutex attributes allow various forms of *priority donation* (will be familiar concept after lab 1)
- **Thread-specific global data**
  - Need for things like `errno`
- **Different synchronization primitives (later in lecture)**

Outline

1. Memory consistency

2. The critical section problem

3. Mutexes and condition variables

4. Implementing synchronization

5. Alternate synchronization abstractions

Implementing synchronization

- **Implement mutex as straight-forward data structure?**
  ```
  typedef struct mutex {
    bool is_locked;        /* true if locked */
    thread_id_t owner;     /* thread holding lock, if locked */
    thread_list_t waiters; /* threads waiting for lock */

  };
  ```

Implementing synchronization

- **Implement mutex as straight-forward data structure?**
  ```
  typedef struct mutex {
    bool is_locked;        /* true if locked */
    thread_id_t owner;     /* thread holding lock, if locked */
    thread_list_t waiters; /* threads waiting for lock */
    lower_level_lock_t lk; /* Protect above fields */
  };
  ```
  - Fine, so long as we avoid data races on the mutex itself
- **Need lower-level lock** `lk` **for mutual exclusion**
  - Internally, `mutex_*` functions bracket code with `lock(&mutex->lk) ... unlock(&mutex->lk)`
  - Otherwise, data races! (E.g., two threads manipulating `waiters`)
- **How to implement** `lower_level_lock_t`**?**
  - Could use Peterson's algorithm, but typically a bad idea (too slow and don't know maximum number of threads)

Approach #1: Disable interrupts

- **Only for apps with** $n : 1$ **threads (1 kthread)**
  - Cannot take advantage of multiprocessors
  - But sometimes most efficient solution for uniprocessors
- **Typical setup: periodic timer signal caught by thread scheduler**
- **Have per-thread "do not interrupt" (DNI) bit**
- `lock (lk)`**: sets thread's DNI bit**
- **If timer interrupt arrives**
  - Check interrupted thread's DNI bit
  - If DNI clear, preempt current thread
  - If DNI set, set "interrupted" (I) bit & resume current thread
- `unlock (lk)`**: clears DNI bit** *and* **checks I bit**
  - If I bit is set, immediately yields the CPU

Approach #2: Spinlocks

- **Most CPUs support atomic read-[modify-]write**
- **Example:** `int test_and_set (int *lockp);`
  - Atomically sets `*lockp = 1` and returns old value
  - Special instruction – no way to implement in portable C99 (C11 supports with explicit `atomic_flag_tet_and_set` function)
- **Use this instruction to implement** *spinlocks*:
  ```
  #define lock(lockp)    while (test_and_set (lockp))
  #define trylock(lockp) (test_and_set (lockp) == 0)
  #define unlock(lockp)  *lockp = 0
  ```
- **Spinlocks implement mutex's** `lower_level_lock_t`
- **Can you use spinlocks instead of mutexes?**
  - Wastes CPU, especially if thread holding lock not running
  - Mutex functions have short C.S., less likely to be preempted
  - On multiprocessor, sometimes good to spin for a bit, then yield

Synchronization on x86

- **Test-and-set only one possible atomic instruction**
- **x86 `xchg` instruction, exchanges reg with mem**
  - Can use to implement test-and-set

  ```
  _test_and_set:
      movl   4(%esp), %edx   # %edx = lockp
      movl   $1, %eax        # %eax = 1
      xchgl  %eax, (%edx)    # swap (%eax, *lockp)
      ret
  ```
- **CPU locks memory system around read and write**
  - Recall `xchgl` always acts like it has implicit `lock` prefix
  - Prevents other uses of the bus (e.g., DMA)
- **Usually runs at memory bus speed, not CPU speed**
  - Much slower than cached read/buffered write

Synchronization on alpha

- `ldl_l` – **load locked**
  `stl_c` – **store conditional (reg←0 if not atomic w. `ldl_l`)**

  ```
  _test_and_set:
      ldq_l  v0, 0(a0)        # v0 = *lockp (LOCKED)
      bne    v0, 1f           # if (v0) return
      addq   zero, 1, v0      # v0 = 1
      stq_c  v0, 0(a0)        # *lockp = v0 (CONDITIONAL)
      beq    v0, _test_and_set # if (failed) try again
      mb
      addq   zero, zero, v0   # return 0
  1:
      ret    zero, (ra), 1
  ```
- **Note: Alpha memory consistency weaker than x86**
  - Want all CPUs to think memory accesses in C.S. happened after acquiring lock, before releasing
  - *Memory barrier* instruction `mb` ensures this (c.f. `mfence` on x86)
  - See Why Memory Barriers for why alpha still worth understanding

Kernel Synchronization

- **Should kernel use locks or disable interrupts?**
- **Old UNIX had 1 CPU, non-preemptive threads, no mutexes**
  - Interface designed for single CPU, so `count++` etc. not data race
  - ...*Unless* memory shared with an interrupt handler

  ```
  int x = splhigh (); /* Disable interrupts */
  /* touch data shared with interrupt handler ... */
  splx (x);           /* Restore previous state */
  ```
  - C.f., `intr_disable` / `intr_set_level` in Pintos, and `preempt_disable` / `preempt_enable` in linux
- **Used arbitrary pointers like condition variables**
  - `int [t]sleep (void *ident, int priority, ...);`
    put thread to sleep; will wake up at `priority` (∼`cond_wait`)
  - `int wakeup (void *ident);`
    wake up all threads sleeping on `ident` (∼`cond_broadcast`)

Kernel locks

- **Nowadays, should design for multiprocessors**
  - Even if first version of OS is for uniprocessor
  - Someday may want multiple CPUs and need *preemptive* threads
  - That's why Pintos uses sleeping locks
    (*sleeping* locks means mutexes, as opposed to *spin*locks)
- **Multiprocessor performance needs fine-grained locks**
  - Want to be able to call into the kernel on multiple CPUs
- **If kernel has locks, should it ever disable interrupts?**

Kernel locks

- **Nowadays, should design for multiprocessors**
  - Even if first version of OS is for uniprocessor
  - Someday may want multiple CPUs and need *preemptive* threads
  - That's why Pintos uses sleeping locks
    (*sleeping* locks means mutexes, as opposed to *spin*locks)
- **Multiprocessor performance needs fine-grained locks**
  - Want to be able to call into the kernel on multiple CPUs
- **If kernel has locks, should it ever disable interrupts?**
  - Yes! Can't sleep in interrupt handler, so can't wait for lock
  - So even modern OSes have support for disabling interrupts
  - Often uses DNI trick when cheaper than masking interrupts in hardware

Outline

Semaphores [Dijkstra]

- **A *Semaphore* is initialized with an integer *N***
- **Provides two functions:**
  - `sem_wait (S)`   (originally called *P*, called `sema_down` in Pintos)
  - `sem_signal (S)`   (originally called *V*, called `sema_up` in Pintos)
- **Guarantees `sem_wait` will return only *N* more times than `sem_signal` called**
  - Example: If *N* == 1, then semaphore acts as a mutex with `sem_wait` as lock and `sem_signal` as unlock
- **Semaphores give elegant solutions to some problems**
  - Unlike condition variables, wait & signal commute
- **Linux primarily uses semaphores for sleeping locks**
  - `sema_init`, `down_interruptible`, `up`, . . .
  - Also weird reader-writer semaphores, `rw_semaphore` [Love]

Semaphore producer/consumer

- **Initialize `full` to 0 (block consumer when buffer empty)**
- **Initialize `empty` to *N* (block producer when queue full)**

```
void producer (void *ignored) {
    for (;;) {
        item *nextProduced = produce_item ();
        sem_wait (&empty);
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        sem_signal (&full);
    }
}
void consumer (void *ignored) {
    for (;;) {
        sem_wait (&full);
        item *nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        sem_signal (&empty);
        consume_item (nextConsumed);
    }
}
```

Various synchronization mechanisms

- **Other more esoteric primitives you might encounter**
  - Plan 9 used a rendezvous mechanism
  - Haskell uses MVars (like channels of depth 1)
- **Many synchronization mechanisms equally expressive**
  - Pintos implements locks, condition vars using semaphores
  - Could have been vice versa
  - Can even implement condition variables in terms of mutexes
- **Why base everything around semaphore implementation?**
  - High-level answer: no particularly good reason
  - If you want only one mechanism, can't be condition variables (interface fundamentally requires mutexes)
  - Because `sem_wait` and `sem_signal` commute, eliminates problem of condition variables w/o mutexes