CS 475
Operating Systems

UNIVERSITY of PUGET SOUND
Est. 1888

Department of Mathematics
and Computer Science

Lecture 6
Synchronization (Part II)

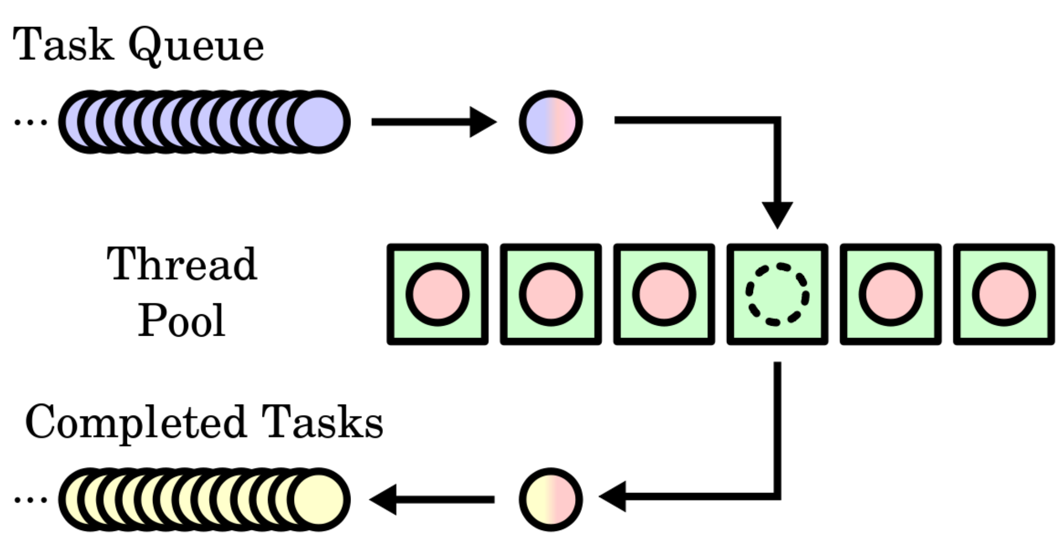
Review of Locks

- Locks are the simplest synchronization primitives, allowing 1 thread in a critical section at a time.
  - (The view is that locks are a "barrier" for threads to get past)

- Types
  - Spinning (busy waiting -- requires TaS(), doesn't make syscalls)
  - Blocking (put self to sleep -- requires TaS(), and making syscalls)
- *What if we needed more complicated coordination of threads?*
  - *What if I want one thread to tell another thread to go?*

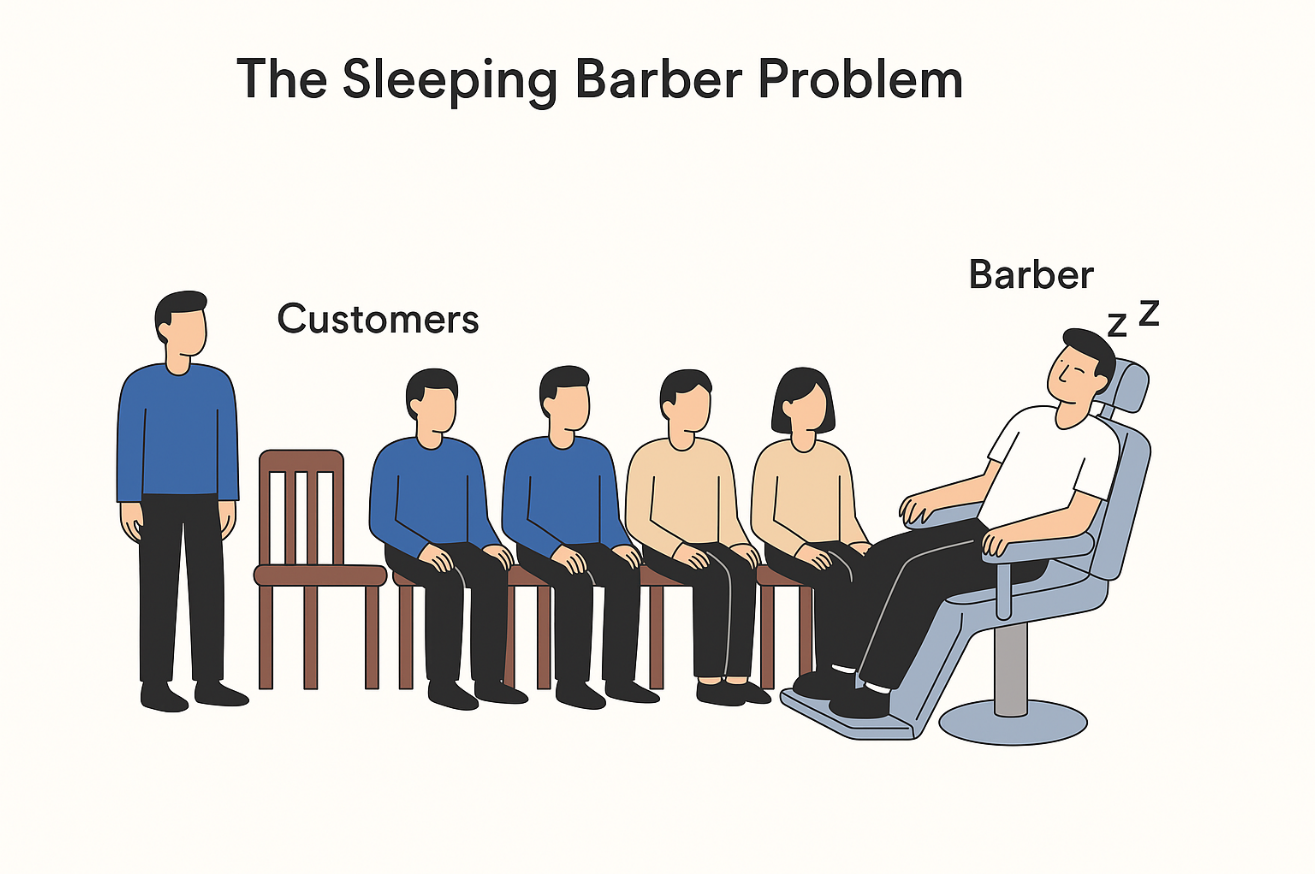
Motivation

- Application Example: Thread Pools
  - To save overhead of creating and re-creating threads, some applications create a "pool" of threads at startup.
  - Threads are initially idle and waiting for work
    - How do you get them all to **wait**?
  - If a task becomes available, dispatch a thread to handle it.
    - How do you **signal** a waiting thread to wake up?



Motivation (2)

- Example: Sleeping Barber Problem
  - 1 barber thread
  - N chairs for customer threads
- If there are no customers, the barber:
  - Goes to sleep.
- If a customer enters and:
  - The barber is asleep, the customer wakes them up.
  - The barber is busy, and there's a free chair, the customer waits.
  - The waiting room is full, the customer leaves.



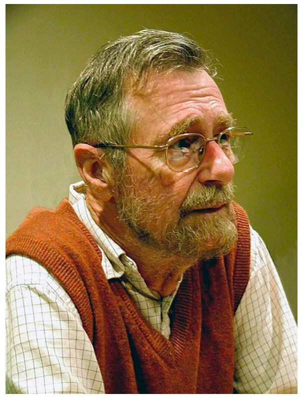
Goals for This Lecture...

- Motivation for Higher Level Synchronization
- Semaphores
  - Definition
  - Implementation of Blocking Semaphores
- Classical Synchronization Problems (Chap 7)
- Semaphore support in C

Semaphores

- We want a more general synchronization primitive for threads to coordinate and signal to each other when to go, when to stop.

- Semaphores can provide robust *coordination* between threads
  - Method of visual signaling, usually by means of flags or lights.
  - Before the invention of the telegraph, semaphore signaling from high towers was used to transmit messages between distant points.



Fun fact:

Semaphores were invented by Edsger Dijkstra for the "T.H.E. Operating System"

Semaphores (Struct)

Semaphore Structure

```
typedef struct sem_t {
  int val;
} sem_t;
```

Semaphores (Creation/Initialization)

Semaphore Structure

```
typedef struct sem_t {
  int val;
} sem_t;
```

```
/** Create */
sem_t* sem_open(int init_val) {
  sem_t *S = (sem_t*) malloc(sizeof(sem_t));
  S->val = init_val;
  return S;
}
```

Semaphores (Wait/Decrement)

Semaphore Structure

```
typedef struct sem_t {
  int val;
} sem_t;
```

```
/** Create */
sem_t* sem_open(int init_val) {
  sem_t *S = (sem_t*) malloc(sizeof(sem_t));
  S->val = init_val;
  return S;
}

/** Wait (spinning semaphore code below is assumed atomic) */
void wait(sem_t* S) {
  while (S->val == 0)
    ;
  S->val--;
}
```

Semaphores (Signal/Increment)

Semaphore Structure

```
typedef struct sem_t {
  int val;
} sem_t;
```

```
/** Create */
sem_t* sem_open(int init_val) {
  sem_t *S = (sem_t*) malloc(sizeof(sem_t));
  S->val = init_val;
  return S;
}

/** Wait (spinning semaphore code below is assumed atomic) */
void wait(sem_t* S) {
  while (S->val == 0)
    ;
  S->val--;
}

/** Signal (code below is assumed atomic) */
void signal(sem_t* S) {
  S->val++;
}
```

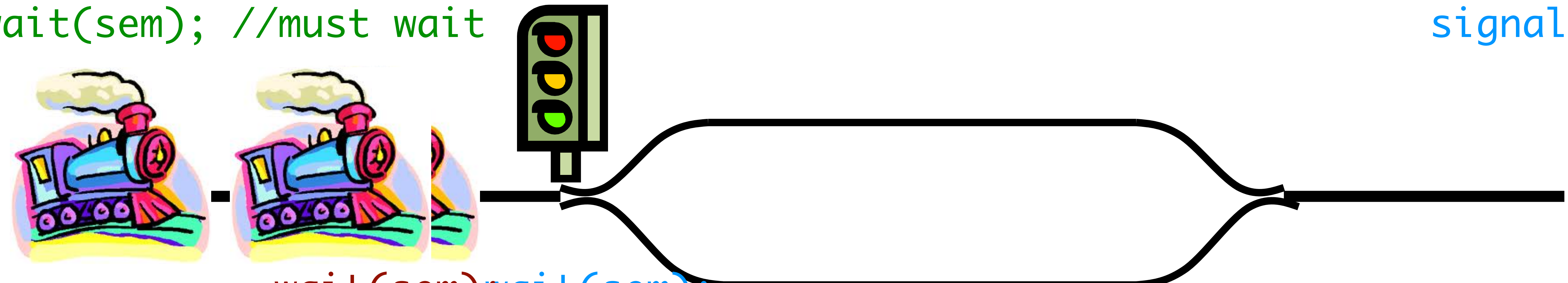
A Coordination Mechanism?

```
// suppose we want to let two trains in critical section
sem_t *sem = sem_open(2);
```

val = 0

wait(sem); //must wait

signal(sem);



wait(sem);wait(sem);

wait(sem); //must wait

Tracks = shared resources
Trains = processes/threads
Stop light = semaphore

Binary Semaphores (When initialized to 1...)

- Recall the semaphore operations (assume they're atomic):

```
/** Wait */
void wait(sem_t* S) {
  while (S->val == 0)
    ;
  S->val--;
}
```

```
/** Signal */
void signal(sem_t* S) {
  S->val++;
}
```

- A *binary semaphore* oscillates between 0 and 1
  - How does a semaphore that's been initialized to *1 behave?*
    - How many threads can get into the critical section at once..?

```
sem_t *S = sem_open(1);

wait(S);

<< critical section >>

signal(S);
```

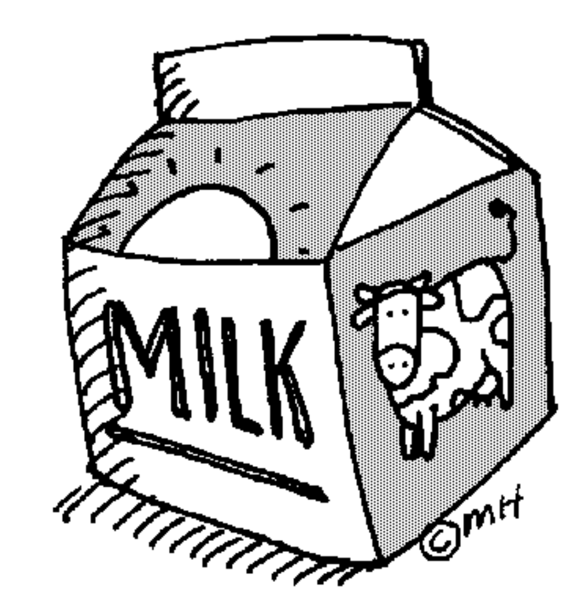
same as a lock!

```
lock_t *lock = newLock();

acquire(lock);

<< critical section >>

release(lock);
```

Too Much Milk Problem Revisited

- Semaphore-based solution
  - (Semaphore as a lock -- initialized to 1)

```
sem_t *ok_to_buy_milk = sem_open(1);

void roommate() {
  while (1) {
    wait(ok_to_buy_milk); // Try to buy milk
      if (noMilk) {
         buyMilk();
      }
    signal(ok_to_buy_milk); // Let someone else check fridge
  }
}
```



Binary Semaphores (When Initialized to 0)

- Recall the semaphore operations (assume they're atomic):

```
/** Wait */
void wait(sem_t* S) {
  while (S->val <= 0)
    ;
  S->val--;
}
```

```
/** Signal */
void signal(sem_t* S) {
  S->val++;
}
```

- We know: Semaphores initialized to 1 is equivalent to a lock.
  - What if we initialize a semaphore to `0`?
  - What would that allow us to do?

Binary Semaphores (Initialized to 0)

- Binary semaphores initialized to *0?*
  - Think of it as a *Gate Keeper*
  - Everyone waits here. I tell you when to pass.
- For instance,
  - In **pthread_join**(), the calling thread just wait until another thread exits.

```
sem_t* sem = sem_open(0);

void pthread_join() {
    wait(sem);
    reapThread();
}

void pthread_exit() {
    signal(sem);
}
```

We've Relied on Semaphores Before!

- Anytime we joined threads, the main (calling) thread had to wait until all threads exited!

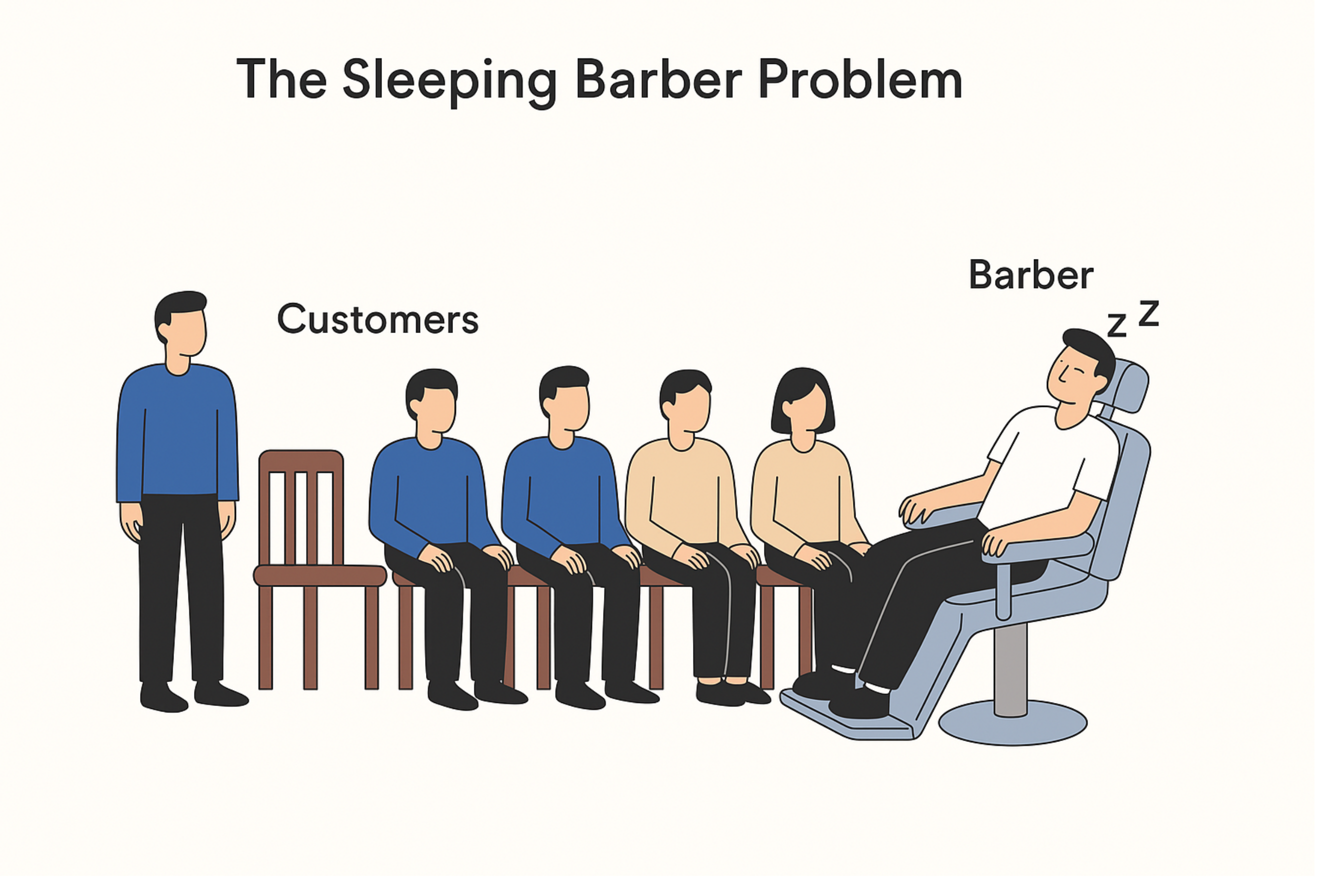
```
// spawn threads
for (int i = 0; i < NUM_THREADS; i++) {
   pthread_create(&threads[i], NULL, parallelFunc, NULL);
}


// wait for threads to finish
for (int i = 0; i < NUM_THREADS; i++) {
   pthread_join(threads[i], NULL);
}
```

Goals for This Lecture...

- Motivation for Higher Level Synchronization
- Semaphores
  - Definition
  - Implementation of Blocking Semaphores
- Classical Synchronization Problems (Chap 7)
- Semaphore support in C

Motivation (2)

- Example: Sleeping Barber Problem
  - 1 barber thread
  - N chairs for customer threads
- If there are no customers, the barber:
  - Goes to sleep.
- If a customer enters and:
  - The barber is asleep, the customer wakes them up.
  - The barber is busy, and there's a free chair, the customer waits.
  - The waiting room is full, the customer leaves.



Synchronize This!

- Hint: You'll need 3 semaphores
  - One for coordinating the barber's action
  - One for coordinating the customer's action
  - One for locking up the shared variable: waitingCustomers

```
const int CHAIRS = 5;
int waitingCustomers = 0;
```

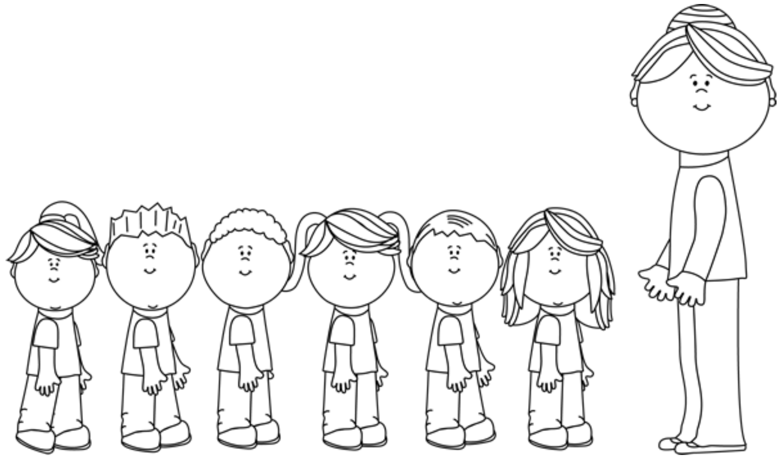
```
void barber() {
  while (1) {
    sleep() when no customers
    waitingCustomers -= 1;
    cutHair();
  }
}
```

```
void customer() {
   if (waitingCustomers < CHAIRS) {
      waitingCustomers++;
      wakeup(barber);
      getHaircut();
   }
   leave();
}
```

Helpful Professor Problem

- Correctness Properties: When do you wait?
  - Professor (One thread)
    - **Waits** for students to arrive
    - When there's a student waiting at the door, prof panics
      - Lets <u>one</u> student in at a time
    - **Waits** for student to ask question
    - After question is asked, prof yells at the student
    - **Waits** for student to leave
    - After student leaves, prof waves bye.

```
void prof() {
    while (1) {
        panic();
        yellAtStudent();
        waveGoodbye();
    }
}
```



- Full working code here:
  - https://github.com/davidtchiu/cs475-lec-helpfulprof

Helpful Professor Problem

- Correctness Properties: When do you wait?
  - Students (N threads)
    - If Professor isn't ready to take another student in his office, they must **wait**
    - After student enters office, they ask questions
    - **Waits** for Professor's answer
    - Leaves Professor's office

```
void student() {
    askQuestion();
    leave();
}
```

What Are All the Problems?

- What could go wrong without synchronization? (Hint: Everything)

```
void prof() {
    while (1) {
        panic();
        yellAtStudent();
        waveGoodbye();
    }
}
```

```
void student() {
    askQuestion();
    leave();
}
```

First Try (prof's Play: Acts I and II)

- Use a semaphore to coordinate 1-student access to my office:

```
sem_t* prof_available = sem_open(0); // track whether prof is ready
```

```
void prof() {
    while (1) {
        panic();
        signal(prof_available);
        yellAtStudent();
        waveGoodbye();
    }
}
```

```
void student() {
    wait(prof_available);
    askQuestion();
    leave();
}
```

Problems? prof shouldn't continuously signal that they're available! (That lets more than one student in!)

--- prof needs to <u>wait</u> if a student is not outside the office!

Second Attempt

- Suppose I define the following semaphores:

```
sem_t* student_outside = sem_open(0);  // track if student is waiting
sem_t* prof_available = sem_open(0);  // track whether prof is ready
```

```
void prof() {
    while (1) {
        wait(student_outside);
        panic();
        signal(prof_available);
        yellAtStudent();
        waveGoodbye();
    }
}
```

```
void student() {
    signal(student_outside);
    wait(prof_available);
    askQuestion();
    leave();
}
```

Problems? Sometimes prof yells at a student before they ask the question!

Sometimes prof waves goodbye before student leaves (or before they ask the question)

Sometimes student leaves before getting yelled at!

Helpful Professor (Correct)

```
sem_t* student_outside   = sem_open(0);
sem_t* prof_available    = sem_open(0);
sem_t* question_asked    = sem_open(0); //wait for question
sem_t* question_answered = sem_open(0); //wait for prof's answer
sem_t* student_leaving   = sem_open(0); //wait for student to leave
```

```
void prof() {
    while (1) {
        wait(student_outside);
        panic();
        signal(prof_available);
        wait(question_asked);
        yellAtStudent();
        signal(question_answered);
        wait(student_leaving);
        waveGoodbye();
    }
}
```

```
void student() {
    signal(student_outside);
    wait(prof_available);
    askQuestion();
    signal(question_asked);
    wait(question_answered);
    leave();
    signal(student_leaving);
}
```

Evaluation of Semaphores

- Pros:
  - More general and robust than locks
  - Can be used to control event handling
- Cons:
  - Asymmetric and extremely error prone
    - Flipping the order of `wait()` and `signal()` in the bounded-buffer problem leads toa deadlock
      (Was it obvious at the time?)
  - Would be nice if we had even higher-level language support to deal with synchronization.

Administrivia 3/28

- Reminders
  - Hwk 6 due 4/9! (No semaphores needed, but they can be used)
- Last time...
  - Blocking locks (vs. Spinlocks)
  - Implementation of locks
- Today:
  - Semaphores
  - Helpful Professor Problem