Announcements/Reminders

- HW 2 due tonight.
- Wednesday: we will have an opportunity to discuss any questions that you have on HW 2.
- Exam 1 on Friday:
  - Same room, same time
  - 1 page of notes allowed (must be your own notes)
  - closed book, closed notes, closed electronic device, opened mind
  - Only lecture material covered through today will be on the exam (so don't worry about "readers-writers favoring writers" and monitors).
- Lab 2: A demo executable is now available (see the web page).
- Lab 3: Is available (executable example coming soon).

Notes

- HW 2 may be handed in as a postscript file.
- "context switch" can sometimes be employed to mean different things.
- Why introduce an abstract notion of a lock? Why not just use Test&Set for this purpose?
- Semaphore programming question (Q 8 of HW 2): 3 different points of contention:
  1. For each entrance, there is contention for who will be next to try to enter the doorway
  2. Once a person has reached the doorway, there is still a question of which doorway will allow them to enter the building
  3. For each exit, there is also contention for who will be next to leave.

Last Class: Semaphores

- A semaphore S supports two atomic operations:
  - **S.Wait():** get a semaphore, wait if busy semaphore S is available.
  - **S.Signal():** release the semaphore, wake up a process if one is waiting for S.
- **Binary or Mutex Semaphore:** grants mutual exclusive access to a resource
- **Counting Semaphore:** useful for granting mutually exclusive access for a set of resources
- Semaphores are useful for mutual exclusion, progress and bounded waiting

Today: Synchronization for Readers/Writers Problem

- An object is shared among may threads, each belonging to one of two classes:
  - Readers: read data, never modify it
  - Writers: read data and modify it
- Using a single lock on the data object is overly restrictive
  - ⇒ Want *many readers* reading the object at once
  - Allow only *one writer* at any point
  - How do we control access to the object to permit this protocol?

Synchronization for Readers/Writers Problem (cont)

- Correctness criteria:
  - Each read or write of the shared data must happen within a critical section.
  - Guarantee mutual exclusion for writers.
  - Allow multiple readers to execute in the critical section at once.
- Two classic forms of the problem:
  - Any reader is allowed to enter the critical section as long as a writer is not in a critical section (favors readers).
  - If a writer is waiting to enter the critical section, readers are not allowed to enter their critical section (favors writers).

$\Rightarrow$ Both cases involve some form of starvation.

Readers/Writers Problem

```
class ReadWrite {
   public void Read();
   public void Write();
   private int       readers; // counts readers
   private Semaphore mutex;   // controls access to readers
   private Semaphore wrt;     // controls entry to first
}                           // writer or reader
ReadWrite {
   readers      = 0;
   mutex        = new Semaphore(1);
   wrt          = new Semaphore(1);
}
```

Readers/Writers Problem

```
Write(){
   wrt.Wait();        // any writers or readers?
   <perform write>
   wrt.Signal();      // enable others
}
Read(){
   mutex.Wait();      // ensure mutual exclusion for reader bookkeeping
      readers += 1;     // another reader
      if (readers == 1)
         wrt.Wait();  // block writers
   mutex.Signal();

   <perform read>

   mutex.Wait();      // ensure mutual exclusion for reader bookkeeping
      readers -= 1;     // reader done
      if (readers == 0)
         wrt.Signal();// enable writers
   mutex.Signal(); }
```

Readers/Writers: Scenario 1

```
R1:                R2:               W1:
Read ()
                  Read ()
                                  Write ()
```

Readers/Writers: Scenario 1

```
R1:              R2:              W1:
Read ()
  |             Read ()
  |               |             Write ()
  |               |               b
  |               |               b
  |               |               b
  |                               b
  |                               b
  |                               b
                                  |
                                  |
                                  |
```

Readers/Writers: Scenario 2

R1:	R2:	W1:
		Write ()
Read ()		
	Read ()	

Readers/Writers: Scenario 2

```
R1:                R2:               W1:
                                Write ()
Read ()                             |
 b              Read ()             |
 b                 b                |
 b                 b                |
 b                 b                |
 |                 |
 |                 |
 |                 |
 |                 |
                   |
                   |
```

Reader/Writers: Scenario 3

```
R1:                R2:              W1:
Read ()
                                  Write ()
                 Read ()
```

Reader/Writers: Scenario 3

```
R1:                 R2:                 W1:
Read ()
   |                                  Write ()
   |                Read ()             b
   |                  |                 b
   |                  |                 b
   |                  |                 b
                      |                 b
                      |                 b
                                        |
                                        |
                                        |
```

Readers/Writers Solution: Discussion

- Implementation notes:
  1. The first reader blocks if there is a writer; any other readers who try to enter block on `mutex`.
  2. The last reader to exit signals a waiting writer.
  3. When a writer exits, if there is both a reader and writer waiting, which goes next depends on the scheduler.
  4. If a writer exits and a reader goes next, then all readers that are waiting will fall through (at least one is waiting on `wrt` and zero or more can be waiting on `mutex`).
  5. Does this solution guarantee all threads will make progress?

- Alternative desirable semantics:
  - Let a writer enter its critical section as soon as possible.

Readers/Writers Solution Favoring Writers

```
class ReadWrite2{
   public void Read();
   public void Write();

   private int writers;
   private int readers;

   private Semaphore write_mutex;     // Mutex keeps this.writers safe
   private Semaphore read_mutex;      // Mutex keeps this.readers safe
   private Semaphore write_pending    // Mutex allows only one reader
                                      //   to initiate at a time
   private Semaphore write_block;     // Keep other writers from writing
   private Semaphore read_block;      // Keep other readers from reading

   public ReadWrite2(){...};
}
```

Readers/Writers Solution Favoring Writers

```
ReadWrite2(){
  writers = readers = 0;
  write_mutex   = new Semaphore(1);
  read_mutex    = new Semaphore(1);
  write_pending = new Semaphore(1);
  write_block   = new Semaphore(1);
  read_block   = new Semaphore(1);
}
```

Readers/Writers Solution Favoring Writers

```
Write(){
   write_mutex.Wait();  // ensure mutual exclusion for writer bookkeeping
      writers += 1;      // another pending writer
      if (writers == 1)  // block readers
         read_block.Wait();
   write_mutex.Signal();

   write_block.Wait();  // ensure mutual exclusion for data write
   <perform write>
   write_block.Signal();

   write_mutex.Wait();  // ensure mutual exclusion for writer bookkeeping
      writers -= 1;      // writer done
      if (writers == 0)  // enable readers
         read_block.Signal();
   write_mutex.Signal(); }
```

Readers/Writers Solution Favoring Writers

```
Read(){
   write_pending.Wait();  // ensures at most one reader will go
                           // before a pending write
     read_block.Wait();
       read_mutex.Wait();      // ensure mutual exclusion
         readers += 1;         // another reader
         if (readers == 1)     // synchronize with writers
           write_block.Wait();
       read_mutex.Signal();
     read_block.Signal();
   write_pending.Signal();

   <perform read>
   read_mutex.Wait();      // ensure mutual exclusion
      readers -= 1;         // reader done
      if (readers == 0)     // enable writers
         write_block.Signal();
   read_mutex.Signal(); }
```

Readers/Writers: Scenario 4

R1:	R2:	W1:	W2:
Read ()			
	Read ()		
		Write ()	
			Write ()

Readers/Writers: Scenario 4

```
R1:               R2:               W1:             W2:
Read ()
 |              Read ()
 |               |              Write ()
 |               |                b            Write ()
 |               |                b              b
                 |                b              b
                 |                b              b
                                  b              |
                                  b              |
                                  b              |
                                  |
                                  |
                                  |
```

Readers/Writers: Scenario 5

R1:	R2:	W1:	W2:
		`Write ()`	
`Read ()`			
	`Read ()`		
			`Write ()`

Readers/Writers: Scenario 5

```
R1:               R2:             W1:             W2:
                                Write ()
Read ()                           |
 b              Read ()           |
 b               b                |             Write ()
 b               b                |               b
 b               b                                |
 b               b                                |
 |               |
 |               |
 |               |
                 |
                 |
```

Reader/Writers: Scenario 6

```
R1:                 R2:                W1:                W2:
Read ()
                                    Write ()
                  Read ()
                                                        Write ()
```

Reader/Writers: Scenario 6

```
R1:        R2:        W1:        W2:
Read ()
 |                    Write ()
 |         Read ()     b
 |          b          b         Write ()
 |          b          b          b
 |          b          b          b
            b          |          b
            b          |          b
            b                     |
            b                     |
            |
            |
```

Other Synchronizations Problems: Dining Philosophers

- Five philosophers, each either eats or thinks
- Share a circular table with five chopsticks
- Thinking: do nothing
- Eating $\Rightarrow$need two chopsticks: try to pick up two closest chopsticks (e.g., left one, then right one).
  - Block if neighbor has already picked up a chopstick
- After eating, put down both chopsticks and go back to thinking

Dining Philosophers Implementation (with Semaphores)

```
Semaphore chopStick[] = new Semaphore[5];
```

::

```
while (true){
    // Left chopstick
    chopStick[i].Wait();
    // Right chopstick
    chopStick[(i + 1) % 5].Wait();

    eat();

    // Return left
    chopStick[i].Signal();
    // Return right
    chopStick[(i + 1) % 5].Signal();

    think();}
```

Dining Philosophers (cont)

- Deadlock: Possible to have the situation in which each philosopher is holding one chopstick and waiting for another.
- But we do not want any starving philosophers: a variety of solutions have been proposed (we will look at one technique next time).

Summary

- Readers/writers problem:
  - Allow multiple readers to concurrently access a data
  - Allow only one writer at a time
- Two possible solutions using semaphores
  - Favor readers
  - Favor writers
- Starvation is possible in either case!
- Dining Philosophers: classic deadlock example.

Next Time

- Monitors
- Review for exam

$\Rightarrow$ Come with questions