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?
- 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.

Readers/Writers Problem

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

Readers/Writers Problem

```
ReadWrite::Write(){
  wrt.Wait();        // any writers or readers?
  <perform write>
  wrt.Signal();      // enable others
}
```

```
ReadWrite::Read(){
  mutex.Wait();      // ensure mutual exclusion
  readers++;    // another reader
  if (readers == 1)
    wrt.Wait();  // block writers
  mutex.Signal();
  <perform read>
  mutex.Wait();      // ensure mutual exclusion
  readers--;    // reader done
  if (readers == 0)
    wrt.Signal(); // enable writers
  mutex.Signal();
}
```

Readers/Writers: Scenario 1

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

Readers/Writers: Scenario 2

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

Readers/Writers: Scenario 3

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

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

```
ReadWrite::Write(){
  write_mutex.Wait();  // ensure mutual exclusion
  writers++;       // another pending writer
  if (writers == 1)  // block readers
    read_block.Wait();
  write_mutex.Signal();
  write_block.Wait();  // ensure mutual exclusion
  <perform write>
  write_block.Signal();
  write_mutex.Wait();  // ensure mutual exclusion
  writers--;       // writer done
  if (writers == 0)  // enable readers
    read_block.Signal();
  write_mutex.Signal();
}
```

Readers/Writers Solution Favoring Writers

```
ReadWrite::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++;         // 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--;         // 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 5

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

Readers/Writers: Scenario 6

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

Readers/Writers using Monitors (Java)

```
class ReaderWriter {
  private int numReaders = 0;
  private int numWriters = 0;

  private synchronized void
      prepareToRead () {
    while ( numWriters > 0 ) wait ();
    numReaders++;
  }
```

```
private synchronized void doneReading () {
  numReaders--;
  if ( numReaders == 0 ) notify ();
}
public ... someReadMethod () {
  // reads NOT synchronized: multiple readers
  prepareToRead ();
  <do the reading>
  doneReading ();
}
```

Readers/Writers using Monitors (Java)

```
private void prepareToWrite () {
    numWriters++;
    while ( numReaders > 0 ) wait ();
  }
  private void doneWriting () {
    numWriters--;
    notify ();
  }
  public synchronized void someWriteMethod (...) {
    // synchronized => only one writer
    prepareToWrite ();
    <do the writing>
    doneWriting ();
  }
}
```

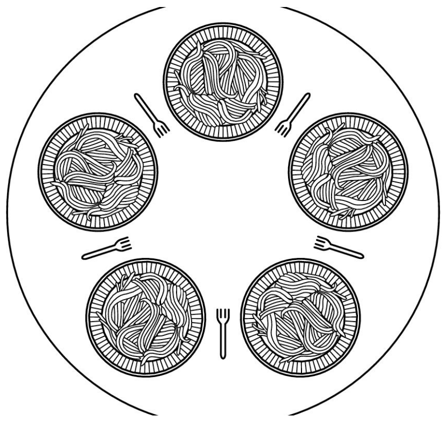
Read/Write Locks

- pthreads and Java support read/write locks
  - A thread can acquire a read lock or a write lock
    - Multiple threads can hold the same read lock concurrently
    - Only one thread can hold a write lock
    - Java: ReadWriteLock class
      - readLock()
      - writeLock()
    - pthread routines:

```
pthread_rwlock_init()
pthread_rwlock_rdlock()
pthread_rwlock_wrlock()
pthread_rwlock_unlock()
```

Dining Philosophers

- It's lunch time in the philosophy dept
- Five philosophers, each either eats or thinks
- Share a circular table with five chopsticks
- Thinking: do nothing
- Eating => need two chopsticks, try to pick up two closest chopsticks
  - Block if neighbor has already picked up a chopstick
- After eating, put down both chopsticks and go back to thinking



Dining Philosophers v1

```
Semaphore  chopsticks[5];

do{
   wait(chopstick[i]);  // left chopstick
   wait(chopstick[(i+1)%5 ]); // right chopstick
      // eat
   signal(chopstick[i]);  // left chopstick
   signal(chopstick[(i+1)%5 ]); // right chopstick
      // think
   } while(TRUE);
```

Dining Philosophers v2 (monitors)

```
monitor DP  {
        enum { THINKING; HUNGRY,
EATING) state [5] ;
        condition self [5];

void synchronized pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING)
            self[i].wait;
        }

void synchronized putdown (int i) {
           state[i] = THINKING;
       //test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
        }
```

```
void test (int i) {
if ( (state[(i + 4) % 5] != EATING)&&
(state[i] == HUNGRY) &&
     (state[(i + 1) % 5] != EATING) ) {
                  state[i] = EATING ;
                  self[i].signal () ;
  }
}

       initialization_code() {
          for (int i = 0; i < 5; i++)
               state[i] = THINKING;
        }
}
```

Dining Philosophers (semaphores)

```
#define N            5              /* number of philosophers */
#define LEFT         (i+N–1)%N      /* number of i’s left neighbor */
#define RIGHT        (i+1)%N        /* number of i’s right neighbor */
#define THINKING     0              /* philosopher is thinking */
#define HUNGRY       1              /* philosopher is trying to get forks */
#define EATING       2              /* philosopher is eating */
typedef int semaphore;              /* semaphores are a special kind of int */
int state[N];                       /* array to keep track of everyone’s state */
semaphore mutex = 1;                /* mutual exclusion for critical regions */
semaphore s[N];                     /* one semaphore per philosopher */

void philosopher(int i)             /* i: philosopher number, from 0 to N–1 */
{
    while (TRUE) {                  /* repeat forever */
        think( );                   /* philosopher is thinking */
        take_forks(i);              /* acquire two forks or block */
        eat( );                     /* yum-yum, spaghetti */
        put_forks(i);               /* put both forks back on table */
    }
}
```

Dining Philosophers (contd)

```
void take_forks(int i)              /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                   /* enter critical region */
    state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
    test(i);                        /* try to acquire 2 forks */
    up(&mutex);                     /* exit critical region */
    down(&s[i]);                    /* block if forks were not acquired */
}
void put_forks(i)                   /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                   /* enter critical region */
    state[i] = THINKING;            /* philosopher has finished eating */
    test(LEFT);                     /* see if left neighbor can now eat */
    test(RIGHT);                    /* see if right neighbor can now eat */
    up(&mutex);                     /* exit critical region */
}

void test(i)                        /* i: philosopher number, from 0 to N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

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: mutually exclusive access to multiple resources