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.

```cpp
ReadWrite::ReadWrite()
{
    readers = 0;
    mutex->value = 1;
    wrt->value = 1;
}
```

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

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

Readers/Writers: Scenario 1

R1: Read ()
R2: Read ()
W1: Write ()
Readers/Writers: Scenario 2

R1:        R2:        W1:
          Write ()

Read ()    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 += 1;       // 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 -= 1;        // writer done
    if (writers == 0)   // enable readers
        read_block.Signal();
    write_mutex.Signal();
}
Readers/Writers Solution Favoring Writers

```c
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 += 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

<table>
<thead>
<tr>
<th>R1:</th>
<th>R2:</th>
<th>W1:</th>
<th>W2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read ()</td>
<td>Read ()</td>
<td>Write ()</td>
<td>Write ()</td>
</tr>
</tbody>
</table>

Readers/Writers: Scenario 5

<table>
<thead>
<tr>
<th>R1:</th>
<th>R2:</th>
<th>W1:</th>
<th>W2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read ()</td>
<td>Write ()</td>
<td>Read ()</td>
<td>Write ()</td>
</tr>
</tbody>
</table>

Readers/Writers: Scenario 6

<table>
<thead>
<tr>
<th>R1:</th>
<th>R2:</th>
<th>W1:</th>
<th>W2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read ()</td>
<td>Write ()</td>
<td>Read ()</td>
<td>Write ()</td>
</tr>
</tbody>
</table>
Readers/Writers using Monitors (Java)

```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 void someReadMethod () {
        // reads NOT synchronized: multiple readers
        prepareToRead ();
        <do the reading>
        doneReading ();
    }
}
```

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()`

Readers/Writers using Monitors (Java)

```java
private synchronized 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 ();
}
```

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

Readers/Writers using Monitors (Java)

```java
private synchronized void doneWriting () {
    numWriters--;
    notify ();
}

public synchronized void someWriteMethod (...) {
    // synchronized => only one writer
    prepareToWrite ();
    <do the writing>
    doneWriting ();
}
```
Dining Philosophers v1

Semaphore chopstick[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;
            signal(self[i]);
        }
        // initialization_code() {
        
    }
    void initialization_code() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }

Dining Philosophers (semaphores)

#define N 5
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N
#define THINKING 0
#define HUNGRY 1
#define EATING 2
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i){
    /* i: philosopher number, from 0 to N-1 */
    while(TRUE) {
        * think()
        * philosopher is thinking *
        take_forks(i);
        * array to keep track of everyone's state *
        * mutual exclusion for critical regions *
        * one semaphore per philosopher *
        eat();
        * yumb-yum, spaghetti *
        * put both forks back on table *
        put_forks(i);
    }
}

Dining Philosophers (contd)

void take_forks(int i){
    /* i: philosopher number, from 0 to N-1 */
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}

void put_forks()
{
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}

void test()
{
    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