Concurrent Programming Issues & Readers/Writers
Summary of Our Discussions

- Developing and debugging concurrent programs is hard
  - Non-deterministic interleaving of instructions
- Safety: isolation and atomicity
- Scheduling: busy-waiting and blocking
- Synchronization constructs
  - Locks: mutual exclusion
  - Condition variables: wait while holding a lock
  - Semaphores: Mutual exclusion (binary) and condition synchronization (counting)
- How can you use these constructs effectively?
  - Develop and follow strict programming style/strategy
Programing Strategy

- Decompose the problem into objects
- Object-oriented style of programming
  - Identify shared chunk of state
  - Encapsulate shared state and synchronization variables inside objects
- Don’t manipulate shared variables or synchronization variables along with the logic associated with a thread
- Programs with race conditions always fail.
  - A. True, B. False
General Programming Strategy

- **Two step process**

- **Threads:**
  - Identify units of concurrency – these are your threads
  - Identify chunks of shared state – make each shared “thing” an object; identify methods for these objects (how will the thread access the objects?)
  - Write down the main loop for the thread

- **Shared objects:**
  - Identify synchronization constructs
    - Mutual exclusion vs. conditional synchronization
  - Create a lock/condition variable for each constraint
  - Develop the methods – using locks and condition variables – for coordination
Coding Style and Standards

- Always do things the same way
- Always use locks and condition variables
- Always hold locks while operating on condition variables
  - If it does not make sense to do this → split your procedures further
- Always use while to check conditions, not if

```c
while (predicate on state variable) {
    conditionVariable->wait(&lock);
};
```

- (Almost) never sleep(), yield(), or isLocked() in your code
  - Use condition variables to synchronize
- Note that printf() internally uses locks, and may hide race conditions
Readers/Writers: A Complete Example

- **Motivation**
  - Shared databases accesses
    - Examples: bank accounts, airline seats, …

- **Two types of users**
  - Readers: Never modify data
  - Writers: read and modify data

- **Problem constraints**
  - Using a single lock is too restrictive
    - Allow multiple readers at the same time
    - …but only one writer at any time
  - **Specific constraints**
    - Readers can access database when there are no writers
    - Writers can access database when there are no readers/writers
    - Only one thread can manipulate shared variables at any time
Readers/Writer: Solution Structure

- **Basic structure: two methods**

  ```
  Database::Read() {
    Wait until no writers;
    Block any writers;
    Access database;
    Let in one writer or reader;
  }
  
  Database::Write() {
    Wait until no readers/writers;
    Write database;
    Let all readers/writers in;
  }
  ```
Solution Details

Public Database::Read() {
    dbLock.lock();
    while(writer) {
        dbAvail.wait();
    }
    reader++;
    dbLock.unlock();
    Read database;
    dbLock.lock();
    reader--;
    if(reader == 0) {
        dbAvail.signal();
    }
    dbLock.unlock();
}

Public Database::Write() {
    dbLock.lock();
    while(reader > 0 || writer){
        dbAvail.wait();
    }
    writer = true;
    dbLock.unlock();
    Write database;
    dbLock.lock();
    writer = false;
    dbAvail.signalAll();
    dbLock.unlock();
}

This solution favors
1. Readers
2. Writers
3. Neither, it is fair
Self-criticism can lead to self-understanding

- Our solution works, but it favors readers over writers.
  - Any reader blocks all writers
  - All readers must finish before a writer can start
  - Last reader will wake any writer, but a writer will wake readers and writers (statistically which is more likely?)
  - If a writer exits and a reader goes next, then all readers that are waiting will get through

- Are threads guaranteed to make progress?
  - A. Yes  B. No
Readers/Writer: Using Monitors

- Basic structure: two methods

```cpp
Database::Read() {
    Wait until no writers;
    Access database;
    Wake up waiting writers;
}
```

```cpp
Database::Write() {
    Wait until no readers/writers;
    Access database;
    Wake up waiting readers/writers;
}
```

- State variables

```cpp
class RWFairLock {
    int AR = 0; // # of active readers
    bool AW = false; // is there an active writer
    Condition okToRead;
    Condition okToWrite;
    LinkedList<RWFairLock> q;
    Lock lock;
}
```
Solution Details: Readers

Class RWFairLock {
  AR = 0; // # of active readers
  AW = false; // is there an active writer
  public bool iRead;
  Condition okToRead;
  Condition okToWrite;
  LinkedList<RWFairLock> q;
  Lock lock;
}

Public Database::Read() {
  StartRead();
  Access database;
  DoneRead();
}

Private Database::StartRead() {
  lock.Acquire();
  iRead = true;
  q.add(this);
  while (AW || !q.peek().iRead) {
    okToRead.wait(&lock);
  }
  AR++;
  lock.Release();
}

Private Database::DoneRead() {
  lock.Acquire();
  AR--;
  q.remove(this);
  if(q.size() > 0) {
    if (q.peek().iRead == false) {
      okToWrite.notify();
    }
  }
  lock.Release();
}
Solution Details: Writers

Class RWFairLock {
    AR = 0; // # of active readers
    AW = false; // is there an active writer
    public bool iRead;
    Condition okToRead;
    Condition okToWrite;
    LinkedList<RWFairLock> q;
    Lock lock;
}

Database::Write() {
    StartWrite();
    Access database;
    DoneWrite();
}

Private Database::StartWrite() {
    lock.Acquire();
    iRead = false;
    q.add(this);
    while (AW || AR > 0 || q.peek().isRead) {
        okToWrite.wait(&lock);
    }
    AW = true;
    lock.Release();
}

Private Database::DoneWrite() {
    lock.Acquire();
    AW = false;
    q.remove(this);
    if(q.size() > 0) {
        if (q.peek().isRead) {
            okToRead.notifyAll();
        } else {
            okToWrite.notify();
        }
    } else {
    }
    lock.Release();
}
Summary

- Allowing concurrent reader execution is a common concurrent programming pattern
- Naïve implementations can starve writers
- Bookkeeping to ensure fair queuing is tricky, but not impossible
  - A lot of effort to reason about all possible interleavings of operations