

	<h2>Concurrent Programming Issues &amp; Readers/Writers</h2>
--	--

	<h3>Summary of Our Discussions</h3> <ul style="list-style-type: none"> <li>◆ Developing and debugging concurrent programs is hard <ul style="list-style-type: none"> <li>➢ Non-deterministic interleaving of instructions</li> </ul> </li> <li>◆ Safety: isolation and atomicity</li> <li>◆ Scheduling: busy-waiting and blocking</li> <li>◆ Synchronization constructs <ul style="list-style-type: none"> <li>➢ Locks: mutual exclusion</li> <li>➢ Condition variables: wait while holding a lock</li> <li>➢ Semaphores: Mutual exclusion (binary) and condition synchronization (counting)</li> </ul> </li> <li>◆ How can you use these constructs effectively? <ul style="list-style-type: none"> <li>➢ Develop and follow strict programming style/strategy</li> </ul> </li> </ul>
--	--

	<h3>Programming Strategy</h3> <ul style="list-style-type: none"> <li>◆ Decompose the problem into objects</li> <li>◆ Object-oriented style of programming <ul style="list-style-type: none"> <li>➢ Identify shared chunk of state</li> <li>➢ Encapsulate shared state and synchronization variables inside objects</li> </ul> </li> <li>◆ Don't manipulate shared variables or synchronization variables along with the logic associated with a thread</li> <li>◆ Programs with race conditions always fail. <ul style="list-style-type: none"> <li>➢ A. True, B. False</li> </ul> </li> </ul>
--	--

	<h3>General Programming Strategy</h3> <ul style="list-style-type: none"> <li>◆ Two step process</li> <li>◆ Threads: <ul style="list-style-type: none"> <li>➢ Identify units of concurrency – these are your threads</li> <li>➢ Identify chunks of shared state – make each shared “thing” an object; identify methods for these objects (how will the thread access the objects?)</li> <li>➢ Write down the main loop for the thread</li> </ul> </li> <li>◆ Shared objects: <ul style="list-style-type: none"> <li>➢ Identify synchronization constructs <ul style="list-style-type: none"> <li>✦ Mutual exclusion vs. conditional synchronization</li> </ul> </li> <li>➢ Create a lock/condition variable for each constraint</li> <li>➢ Develop the methods –using locks and condition variables – for coordination</li> </ul> </li> </ul>
--	--

	<h3>Coding Style and Standards</h3> <ul style="list-style-type: none"> <li>◆ Always do things the same way</li> <li>◆ Always use locks and condition variables</li> <li>◆ Always hold locks while operating on condition variables</li> <li>◆ Always acquire lock at the beginning of a procedure and release it at the end <ul style="list-style-type: none"> <li>➢ If it does not make sense to do this → split your procedures further</li> </ul> </li> <li>◆ Always use while to check conditions, not if <div style="border: 1px solid black; padding: 5px; margin: 5px 0;"> <pre>while (predicate on state variable) {     conditionVariable-&gt;wait(&amp;lock); }</pre> </div> </li> <li>◆ (Almost) never sleep(), yield(), or isLocked() in your code <ul style="list-style-type: none"> <li>➢ Use condition variables to synchronize</li> </ul> </li> <li>◆ Note that printf() internally uses locks, and may hide race conditions</li> </ul>
--	---

	<h3>Readers/Writers: A Complete Example</h3> <ul style="list-style-type: none"> <li>◆ Motivation <ul style="list-style-type: none"> <li>➢ Shared databases accesses <ul style="list-style-type: none"> <li>✦ Examples: bank accounts, airline seats, ...</li> </ul> </li> </ul> </li> <li>◆ Two types of users <ul style="list-style-type: none"> <li>➢ Readers: Never modify data</li> <li>➢ Writers: read and modify data</li> </ul> </li> <li>◆ Problem constraints <ul style="list-style-type: none"> <li>➢ Using a single lock is too restrictive <ul style="list-style-type: none"> <li>✦ Allow multiple readers at the same time</li> <li>✦ ...but only one writer at any time</li> </ul> </li> <li>➢ Specific constraints <ul style="list-style-type: none"> <li>✦ Readers can access database when there are no writers</li> <li>✦ Writers can access database when there are no readers/writers</li> <li>✦ Only one thread can manipulate shared variables at any time</li> </ul> </li> </ul> </li> </ul>
--	---

### 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

```
Lock dbLock;
Condition dbAvail;
int reader = 0;
bool writer = false;
```

```
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

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

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

- State variables

```
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;
}
```

### 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;
}
```

```
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();
}
```

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

### 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;
}
```

```
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();
        }
    }
    lock.Release();
}
```

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

	Summary
	<ul style="list-style-type: none"><li>◆ Allowing concurrent reader execution is a common concurrent programming pattern</li><li>◆ Naïve implementations can starve writers</li><li>◆ Bookkeeping to ensure fair queuing is tricky, but not impossible<ul style="list-style-type: none"><li>➢ A lot of effort to reason about all possible interleavings of operations</li></ul></li></ul>