Race Conditions

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Load commonSum</td>
<td>Load commonSum</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+=</td>
<td>+=</td>
</tr>
<tr>
<td></td>
<td>Store commonSum</td>
<td>Store commonSum</td>
</tr>
</tbody>
</table>

- If “Thread 2” is not aware that “Thread 1” has begun to start its addition, it will likely overwrite the addition that Thread 1 is performing.
Consistency

• Hmm... “while the other is in the process of editing it...”
  – What we desire is some way to notate that an edit is in progress.
  – Unfortunately, the semantics of “edit in progress” is not possible to determine automatically by a compiler.
  – While our example “edit” is simple, there are situations where entire algorithms must run to produce the desired edit.
Consistency

• The idea of noting this “edit in progress” would be to ensure that data may only be accessed when it is in a consistent state.

  – That is, it may only be accessed whenever any edits have been completed.

  – Think of it as “encapsulation meets concurrency.”
Consistency

The idea of noting this “edit in progress” would be to ensure that data may only be accessed when it is in a consistent state.

– Part of this means that the data may not be edited whenever it is being accessed for use elsewhere – otherwise, it may cause inconsistencies like that from the example.
Mutexes

• One construct designed explicitly for these situations is that of the mutex.
  – Mutexes exist in various forms in many different programming languages.
  – Java calls this a lock.
  – There are two fundamental operations for a mutex/lock.
    – lock() – acquires the right to access the protected area.
    – unlock() – relinquishes said right.
Mutexes

• When the lock() method is called by a thread, one of two things happens:
  1. If no thread owns the lock access rights, the thread gains access rights without question.
  2. For certain kinds of mutexes, this will also work properly if the current thread already owns them.
Mutexes

• When the lock() method is called by a thread, one of two things happens:

  2. If another thread owns the access rights, the current thread stalls and waits until it is granted those rights.

  3. In order for this to have a chance to happen, the thread with access rights (generally) must relinquish those rights through the unlock method.
Mutexes

- C++’s mutexes are defined in the `<mutex>` header file. (C++11)
  - In Java, Lock is an interface.
  - ReentrantLock is the default implementation of said interface.
    - *For the general case that the current* thread may already own the access rights being permitted for the lock() method. Thus, the lock is reentrant.
  - In C++, there exist mutex and recursive_mutex, for starters.
Mutexes

- One must be careful when using mutexes.
  - If the lock() method is called without a corresponding unlock() later, entire threads could be shut down as a result.
  - Both methods must be manually called whenever and wherever they are needed – leaving out protective lock() calls could easily be done by accident.
Condition Variables

• One alternate concurrency tool is that of the *condition variable*.
  – This structure is designed to allow threads to cease operation temporarily and *wait* for some *condition* before continuing.
Condition Variables

- One alternate concurrency tool is that of the condition variable.
  - Suppose one thread (a “producer”) is designed to dispatch work for other threads (“consumers”) to do.
  - When no work is available, “consumer” threads should stall, waiting on new work to be available.
  - The “producer” will notify them when new tasks are ready.
Condition Variables

• In C++, the condition_variable class from the std namespace fulfills this role.
  – This class has two primary methods:
    – wait()
    – notify()
Condition Variables

• Both methods require a mutex to be passed in as a parameter.
  – Loosely speaking, at least. There’s an intermediate object.
  – This mutex must be owned by threads calling the wait() method.
  – wait() will unlock the mutex before pausing the thread, and re-lock the mutex once resumed.
Condition Variables

• Note – once one thread has called a `condition_variable`’s `wait()` method, it can only resume if `notify()` is called at a later time by another thread.
# Race Conditions

If “Thread 2” is not aware that “Thread 1” has begun to start its addition, it will likely overwrite the addition that Thread 1 is performing.

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Load commonSum</td>
<td>Load commonSum</td>
</tr>
<tr>
<td></td>
<td>+=</td>
<td>+=</td>
</tr>
<tr>
<td></td>
<td>Store commonSum</td>
<td>Store commonSum</td>
</tr>
</tbody>
</table>

**Table:**

- **Thread 1:**
  - Load `commonSum`
  - `+=`
  - Store `commonSum`

- **Thread 2:**
  - Load `commonSum`
  - `+=`
  - Store `commonSum`
Locks

• One construct designed to prevent race conditions is that of the Lock.
  – Locks exist in various forms in many different programming languages.
  – There are two fundamental operations for a Lock.
    – lock() – acquires the right to access the protected area.
    – unlock() – relinquishes said right.
Locking Problems

• Unfortunately, locking is not something that’s easy to get totally right.

• There are some known issues inherent to locking which need to be examined.
  – The critical point to remember: if a thread desires access to a lock that is owned by another thread, it *stalls*. 
Locking Problems

• First off, locks are not ideal in places where contention is expected to occur frequently.
  – *Contention*: two or more threads competing for concurrent access to some object.
  – Locking can cause threads to temporarily stop execution, reducing the potential parallelism of the program.
Locking Problems

• The *real* problem with locks – when not handled carefully, they can backfire and cause a program to completely hang. (Or, “lock up.”)

• Example – Suppose some program has two locks: **A & B**.
  – Each lock protects certain critical objects within the program.
Locking Problems

- Now suppose that, for some operation within the program, both locks A & B must be owned by a single thread, simultaneously.
  - Other parts would require only A or only B.
Locking Problems

• The problem: what if one thread is holding lock A while the second thread needs both A and B?
  – In particular, if thread 2 already holds B...
  – Clearly, thread 2 will be forced to wait for A.
  – ... so if thread 1 then wants to perform the same operation, and needs B, what happens?
Deadlock

• So, in essence, we have the following:
  – Thread 1 starts with lock A permissions.
  – Thread 2 starts with lock B permissions.
  – Each thread \textit{then} tries to acquire the other lock.

• This situation results in \textit{deadlock}.
Deadlock

- *Deadlock* is the situation where two (or more) threads have a special form of contention regarding more than one lock.
  - Each thread needs the same two locks, and starts with only one of them, trying to acquire the other.
  - As a result, they both end up permanently locked out. Dead-lock.
Deadlock Avoidance

- Obviously, deadlock is a condition that must be avoided by a program.
  - Unfortunately, this is easier said than done: avoiding deadlocks is a context-sensitive process.
Deadlock Avoidance

• One technique is to require that any locks to be acquired must be acquired in some well-established order.

• Another requires mass locking and unlocking.
  – All needed locks are acquired at the same time, and to get a new one requires releasing those currently held.
A Different Strategy

• All of the locking techniques we’ve seen so far are often called blocking techniques.
  – This is because the progress of (at least) one thread in contention will be blocked.

• There also exist, correspondingly, nonblocking techniques.