Concurrency

Recitation – 2/24
Nisarg Raval
Slides by Prof. Landon Cox
Threads

- **Light weight**
- **Easy and Fast**
- **Each thread has its own**
  - SP, PC, Registers
- **But they share**
  - Heap, Code, Library
  - Slow devices, interactive tasks, prioritization
  - Server addressing multiple clients
Example

- **Two threads accessing account balance**
- **Thread 1 deposit money**
- **Thread 2 withdraw money**
- **Initial balance is 100**
  - What is the final balance?

```
balance = balance * 2;
printf("%d\n", balance);
```
```
balance = balance - 110;
printf("%d\n", balance);
```
Debugging is Hard

- **Worst case reasoning**
  - Think of all ways threads can be scheduled

- **Difficult to test all possible interleavings**

- **Bugs may appear only sometimes**
  - Rerunning the program may make them disappear!
Constraining concurrency

- **Synchronization**
  - Controlling thread interleavings

- **Some events are independent**
  - No shared state
  - Relative order of these events don’t matter

- **Other events are dependent**
  - Output of one can be input to another
  - Their order can affect program results
Goals of synchronization

1. All interleavings must give correct result
   - Correct concurrent program
     - Works no matter how fast threads run
   - Important for lab3!

2. Constrain program as little as possible
   - Why?
     - Constraints slow program down
     - Constraints create complexity
“Too much milk” principals
“Too much milk” rules

- The fridge must be stocked with milk
  - Milk expires quickly, so never > 1 milk
- Landon and Melissa
  - Can come home at any time
  - If either sees an empty fridge, must buy milk
  - Code (no synchronization)

```java
if (noMilk) {
    buy milk;
}
```
<table>
<thead>
<tr>
<th>Time</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00</td>
<td>Look in fridge (no milk)</td>
</tr>
<tr>
<td>3:05</td>
<td>Go to grocery store</td>
</tr>
<tr>
<td>3:10</td>
<td>Look in fridge (no milk)</td>
</tr>
<tr>
<td>3:15</td>
<td>Buy milk</td>
</tr>
<tr>
<td>3:20</td>
<td>Go to grocery store</td>
</tr>
<tr>
<td>3:25</td>
<td>Arrive home, stock fridge</td>
</tr>
<tr>
<td>3:30</td>
<td>Buy milk</td>
</tr>
</tbody>
</table>
What broke?

- *Code worked sometimes, but not always*
  - Code contained a **race condition**
  - Processor speed caused incorrect result
- *First type of synchronization*
  - Mutual exclusion
  - Critical sections
Synchronization concepts

- **Mutual exclusion**
  - Ensure 1 thread doing something at a time
  - E.g. 1 person shops at a time
  - Code blocks are atomic w/re to each other
  - Threads can’t run code blocks at same time
Synchronization concepts

- **Critical section**
  - Code block that must run atomically
    - “with respect to some other pieces of code”

- *If A and B are critical w/re to each other*
  - Threads mustn’t interleave code from A and B
  - A and B mutually exclude each other

- **Conflicting code is often same block**
  - But executed by different threads
  - Reads/writes shared data (e.g. screen, fridge)
Back to “Too much milk”

- *What is the critical section?*

- *Landon and Melissa’s critical sections*
  - Must be atomic w/re to each other

```java
if (noMilk){
    buy milk;
}
```
“Too much milk” solution 1

- Assume only *atomic load/store*
  - Build larger atomic section from load/store
- Idea:
  1. Leave notes to say you’re taking care of it
  2. Don’t check milk if there is a note
Solution 1 code

- **Atomic operations**
  - Atomic load: check note
  - Atomic store: leave note

```java
if (noMilk) {
    if (noNote){
        leave note;
        buy milk;
        remove note;
    }
}
```
Does it work?

if (noMilk) {
    if (noNote) {
        leave note;
        buy milk;
        remove note;
    }
}

Is this better than no synchronization at all?
What if “if” sections are switched?
What broke?

- **Melissa’s events can happen**
  - After Landon checks for a note
  - Before Landon leaves a note

```java
if (noMilk) {
    if (noNote) {
        leave note;
        buy milk;
        remove note;
    }
}
```
Next solution

- *Idea:*
  - Change the order of “leave note”, “check note”
  - Requires labeled notes (else you’ll see your note)
Does it work?

leave noteLandon
if (no noteMelissa){
    if (noMilk){
        buy milk;
    }
}
remove noteLandon

leave noteMelissa
if (no noteLandon){
    if (noMilk){
        buy milk;
    }
}
remove noteMelissa

Nope. (Illustration of “starvation.”)
What about now?

```java
while (noMilk)
    leave noteLandon
if(no noteMelissa)
    if(noMilk)
        buy milk;
    
remove noteLandon

while (noMilk)
    leave noteMelissa
if(no noteLandon)
    if(noMilk)
        buy milk;
    
remove noteMelissa
```

Nope. (Same starvation problem as before)
Next solution

- We’re getting closer

Problem
- Who buys milk if both leave notes?

Solution
- Let Landon hang around to make sure job is done
Too much milk solution

```c
leave noteLandon
while (noteMelissa){
    do nothing
}
if (noMilk){
    buy milk;
}
remove noteLandon

leave noteMelissa
if (no noteLandon){
    if (noMilk){
        buy milk;
    }
}
remove noteMelissa
```
Downside of solution

- **Complexity**
  - Hard to convince yourself it works

- **Asymmetric**
  - Landon and Melissa run different code

- *Not clear if this scales to > 2 people*

- **Landon consumes CPU while waiting**
  - *Busy-waiting*

- *Note: only needed atomic load/store*
Raising the level of abstraction

- **Locks**
  - Also called *mutexes*
  - Provide mutual exclusion
  - Prevent threads from entering a critical section

- **Lock operations**
  - Lock (aka `Lock::acquire`)
  - Unlock (aka `Lock::release`
Lock operations

- **Lock**: wait until lock is free, then acquire it
  
  ```
  do {
    if (lock is free) {
      acquire lock
      break
    }
  } while (1)
  ```

  This is a busy-waiting implementation
  - We’ll improve on this in a few lectures

- **Unlock**: atomic *release* lock
Too much milk, solution 2

```java
if (noMilk) {
    if (noNote) {
        leave note;
        buy milk;
        remove note;
    }
}
```

Why doesn’t the note work as a lock?

Block is not atomic. Must atomically:
- check if lock is free
- grab it
Elements of locking

1. The lock is initially free
2. Threads acquire lock before an action
3. Threads release lock when action completes
4. Lock() must wait if someone else has lock

- Key idea
  - All synchronization involves waiting
  - Threads are either running or blocked
Too much milk with locks?

- Waiting for lock while other buys milk
Too much milk “w/o waiting”?

lock ()
if (noNote && noMilk) {
    leave note “at store”
    unlock ()
    buy milk
    lock ()
    remove note
    unlock ()
} else {
    unlock ()
}

Not holding lock

lock ()
if (noNote && noMilk) {
    leave note “at store”
    unlock ()
    buy milk
    lock ()
    remove note
    unlock ()
} else {
    unlock ()
}

Only hold lock while handling shared resource.
lock ()
if (noMilk && noNote){
  leave note “at store”
  unlock ()
  buy milk
  stock fridge
  remove note
} else {
  unlock ()
}

lock ()
if (noMilk && noNote){
  leave note “at store”
  unlock ()
  buy milk
  stock fridge
  remove note
} else {
  unlock ()
}
What about Java? Too much milk

- Every object is a lock
- Use `synchronized` key word
  - Lock: `{", unlock:"`
Concurrency in Java

- Implement a Runnable Interface
  - `public class MyThread implements Runnable{
    public void run(){}
  }

- Subclass Thread
  - `public class MyThread extends Thread{
    public void run(){}
  }`
Example

// Custom thread class
public class MyThread
    Implements Runnable{
    public MyThread(..){
        ...
    }
    // Override thread run method
    public void run() {
        // run the thread
        ...
    }
}

// Client class
public class Client {
    public void action(..) {
        // Create thread
        Thread t1 = new Thread(new myThread(..));
        // Start thread
        t1.start();
        Thread t2 = new Thread(new myThread(..));
        t2.start();
    }
}