Concurrency: Locks and synchronization

Slides by Prof. Cox
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 your projects!

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>
<tr>
<td>3:35</td>
<td>Arrive home, stock fridge</td>
</tr>
</tbody>
</table>

Too much milk!
What broke?

• Code worked sometimes, but not always
  • Code contained a race condition
  • Processor speed caused incorrect result

• First type of synchronization
  • Mutual exclusion inside 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?**

```java
if (noMilk){
    buy milk;
}
```

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

```
Next solution

• **Idea:**
  
  • Change the order of “leave note”, “check note”
  
  • Kind of like a reservation
  
  • 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.”)
while (noMilk)
    {
        leave noteLandon
        if(no noteMelissa)
            {
                if(noMilk)
                    {
                        buy milk;
                    }
            }
        remove noteLandon
    }

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
Does it work?

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

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

Yes! It does work! Can you show it?
Downside of solution

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

- **Asymmetric**
  - Landon and Melissa run different code
  - Approach doesn’t apply to > 2 people

- **Landon consumes CPU while waiting**
  - **Busy-waiting**
  - However, only needed atomic load/store
Raising the level of abstraction

• **Mutual exclusion with atomic load/store**
  • Painful to program
  • Wastes resources
  • Need more HW support
  • Will be covered later

• **OS can provide higher level abstractions**
Too much milk solution

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
  • Approach doesn’t apply to > 2 people

• **Landon consumes CPU while waiting**
  • Busy-waiting
  • However, 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)
  ```

  Must be atomic with respect to other threads calling this code

- This is a busy-waiting implementation

- We’ll fix 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?

Problem?

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

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

Not holding lock

```java
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 ()
}
Example: thread-safe queue

enqueue () {
    lock (qLock)
    // ptr is private
    // head is shared
    new_element = new node();
    if (head == NULL) {
        head = new_element;
    } else {
        node *ptr;
        // find queue tail
        for (ptr=head;
            ptr->next!=NULL;
            ptr=ptr->next){}
        ptr->next=new_element;
    }
    new_element->next=0;
    unlock(qLock);
}

dehqueue () {
    lock (qLock);
    element=NULL;
    if (head != NULL) {
        // if queue non-empty
        if (head->next!=0) {
            // remove head
            element=head->next;
            head->next=
            head->next->next;
        } else {
            element = head;
            head = NULL;
        }
    }
    unlock (qLock);
    return element;
}

What can go wrong?
Thread-safe queue

- Can enqueue unlock anywhere?
  - No
- Must leave shared data
  - In a consistent/sane state
- Data invariant
  - “consistent/sane state”
  - “always” true

```java
lock (qLock)
// ptr is private
// head is shared
new_element = new node();
if (head == NULL) {
    head = new_element;
} else {
    node *ptr;
    // find queue tail
    for (ptr=head;
        ptr->next!=NULL;
        ptr=ptr->next){}
    ptr->next=new_element;
}
unlock(qLock); // safe?
new_element->next=0;
```
Invariants

- **What are the queue invariants?**
  - Each node appears once (from head to null)
  - Enqueue results in prior list + new element
  - Dequeue removes exactly one element

- **Can invariants ever be false?**
  - Must be
  - Otherwise you could never change states
More on invariants

• **So when is the invariant broken?**
  • Can only be broken while lock is held
  • And only by thread holding the lock
BROKEN INVARIANT
(CLOSE AND LOCK DOOR)
INVARIANT
RESTORED
(UNLOCK DOOR)

http://www.flickr.com/photos/jacobaaron/348964486
More on invariants

• **So when is the invariant broken?**
  • Can only be broken while lock is held
  • And only by thread holding the lock

• **Really a “public” invariant**
  • The data’s state in when the lock is free
  • Like having your house tidy before guests arrive

• **Hold a lock whenever manipulating shared data**
More on invariants

- What about reading shared data?
  - Still must hold lock
  - Else another thread could break invariant
  - (Thread A prints Q as Thread B enqueues)
I'm always holding a lock while accessing shared state.

enqueue () {
    lock (qLock)
    // ptr is private
    // head is shared
    new_element = new node();
    if (head == NULL) {
        head = new_element;
    } else {
        node *ptr;
        // find queue tail
        for (ptr=head;
            ptr->next!=NULL;
            ptr=ptr->next) {}
        unlock(qLock);
        lock(qLock);
        ptr->next=new_element;
    }
    new_element->next=0;
    unlock(qLock);
}

ptr may not point to tail after lock/unlock.

Lesson:
- Thinking about individual accesses is not enough
- Must reason about dependencies between accesses
What about Java? Too much milk

- Every object is a lock
- Use synchronized key word (lock ="{",
  unlock=""}")
public synchronized void put(int value) {
    contents = value;
}

• What does this mean? What is the lock?
• “this” is the lock
Synchronizing methods

```java
public class CubbyHole {
    private int contents;

    public int get() {
        return contents;
    }

    public void put(int value) {
        synchronized (this) {
            contents = value;
        }
    }
}
```

- Equivalent to “synchronized (this)” block
Intro to ordering constraints

• Say you want dequeue to wait while the queue is empty

• Can we just busy-wait?
  • No!
  • Still holding lock

```c
dequeue () {
    lock (qLock);
    element=NULL;
    while (head==NULL) {} // remove head
    element=head->next;
    head->next=NULL;
    unlock (qLock);
    return element;
}
```
Release lock before spinning?

What can go wrong? Head might be NULL when we try to remove entry

dequeue () {
    lock (qLock);
    element=NULL;
    unlock (qLock);
    while (head==NULL) {}
    lock (qLock);
    // remove head
    element=head->next;
    head->next=NULL;
    unlock (qLock);
    return element;
}
One more try

- **Does it work?**
  - Seems ok
- **Why?**
  - ShS protected
- **What’s wrong?**
  - Busy-waiting
  - Wasteful

```c
dequeue () {
    lock (qLock);
    element=NULL;
    while (head==NULL) {
        unlock (qLock);
        lock (qLock);
    }
    // remove head
    element=head->next;
    head->next=NULL;
    unlock (qLock);
    return element;
}
```
Ideal solution

- **Would like dequeueing thread to “sleep”**
  - Add self to “waiting list”
  - Enqueuer can wake up when Q is non-empty

- **Problem: what to do with the lock?**
  - Why can’t dequeueing thread sleep with lock?
  - Enqueuer would never be able to add
Release the lock before sleep?

 enqueue () {
    acquire lock
    find tail of queue
    add new element
    if (dequeueer waiting) {
        remove from wait list
        wake up dequeueer
    }
    release lock
}

dequeue () {
    acquire lock
    ...
    if (queue empty) {
        release lock
        add self to wait list
        sleep
        acquire lock
    }
    ...
    release lock
}

Does this work?
Release the lock before sleep?

enqueue () {
    acquire lock
    find tail of queue
    add new element
    if (dequeuer waiting){
        release lock
        wake up dequeuer
    }
    release lock
}

dequeue () {
    acquire lock
    ...  
    if (queue empty) {
        release lock
        add self to wait list
        sleep
        acquire lock
    }
    ...  
    release lock
}

Thread can sleep forever
Release the lock before sleep?

enqueue () {
    acquire lock
    find tail of queue
    add new element
    if (dequeuer waiting){
        remove from wait list
        wake up dequeuer
    }
    release lock
}

dehqueue () {
    acquire lock
    ...
    if (queue empty) {
        add self to wait
        release lock
        sleep
        acquire lock
    }
    ...
    release lock
}
Release the lock before sleep?

enqueue () {
    acquire lock
    find tail of queue
    add new element
    if (dequeueer waiting){
        remove from wait list
        wake up dequeueer
    }
    release lock
}

dequeue () {
    acquire lock
    ...
    if (queue empty) {
        add self to wait list
        release lock
        sleep
        acquire lock
    }
    ...
    release lock
}
In Monday's Class

• Mutual exclusion is necessary, but insufficient

• Still need **ordering constraints**
  • Often must wait for something to happen
  • Use something called “monitors”