Operating Systems
(Honor Track)

Synchronization 2: Lock Implementation

Xin Jin

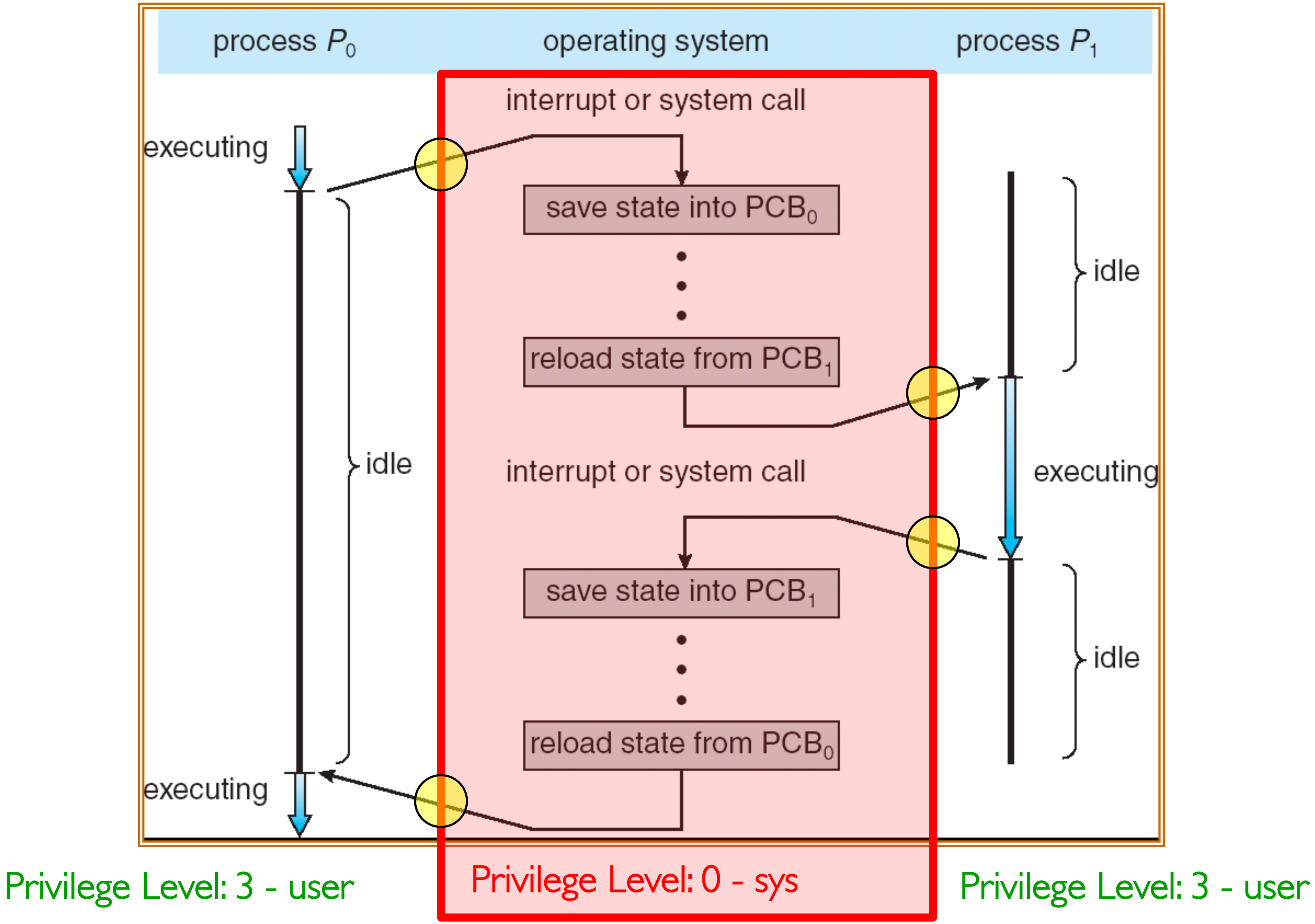
Spring 2023

Acknowledgments: Ion Stoica, Berkeley CS 162

Recap: Keshav's Three-Pass Approach

- A ten-minute scan to get the general idea
  - Title, abstract, and introduction
  - Section and subsection titles
  - Conclusion and bibliography
- A more careful, one-hour reading
  - Read with greater care, but ignore details like proofs
  - Figures, diagrams, and illustrations
  - Mark relevant references for later reading
- Several-hour virtual re-implementation of the work
  - Making the same assumptions, recreate the work
  - Identify the paper's innovations and its failings
  - Identify and challenge every assumption
  - Think how you would present the ideas yourself
  - Jot down ideas for future work

Recap: Context Switch



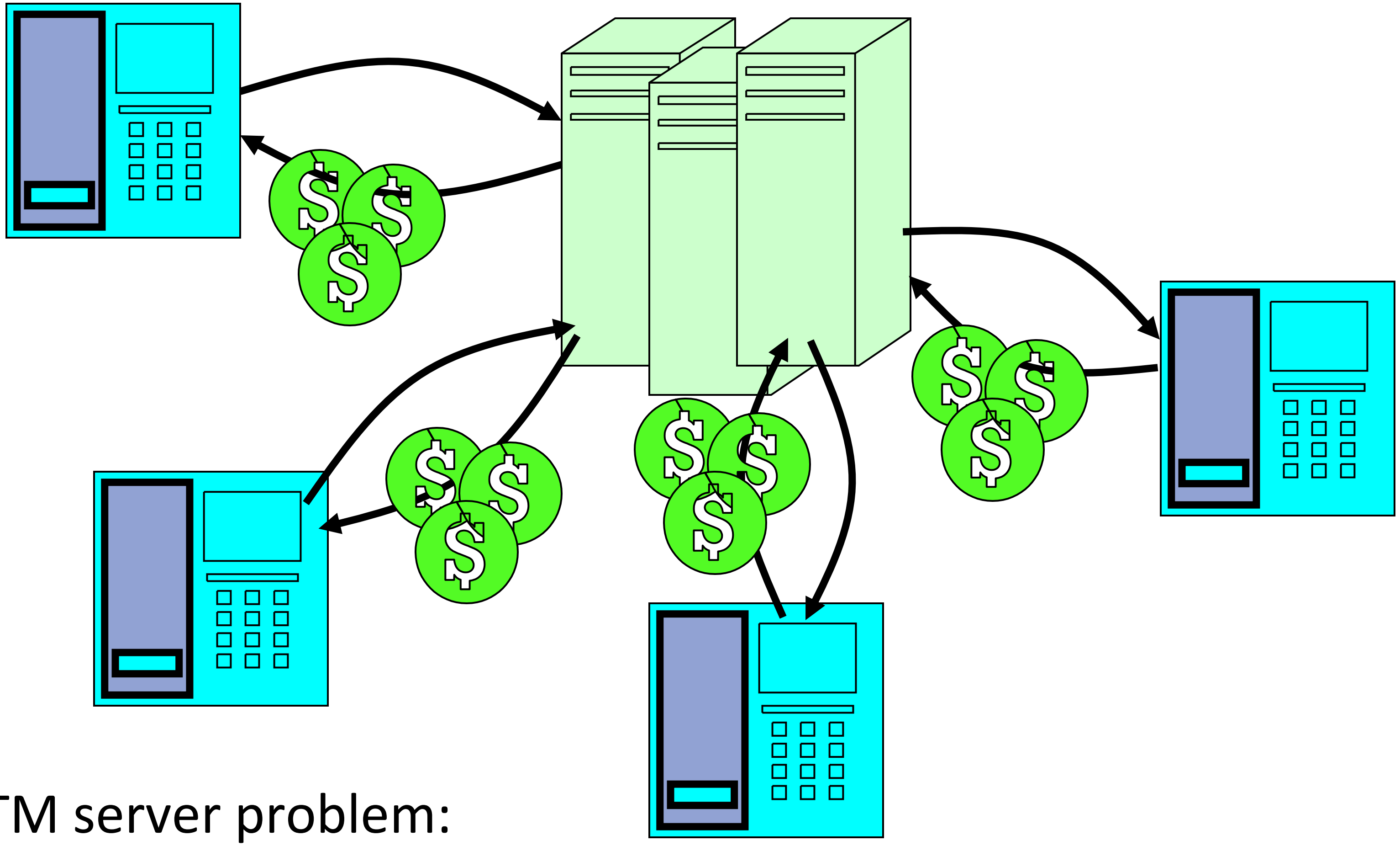
Recap: The Core of Concurrency: the Dispatch Loop

- Conceptually, the scheduling loop of the operating system looks as follows:

```
Loop {
   RunThread();
   ChooseNextThread();
   SaveStateOfCPU(curTCB);
   LoadStateOfCPU(newTCB);
}
```

- This is an *infinite* loop
  - – One could argue that this is all that the OS does

Recap: ATM Bank Server



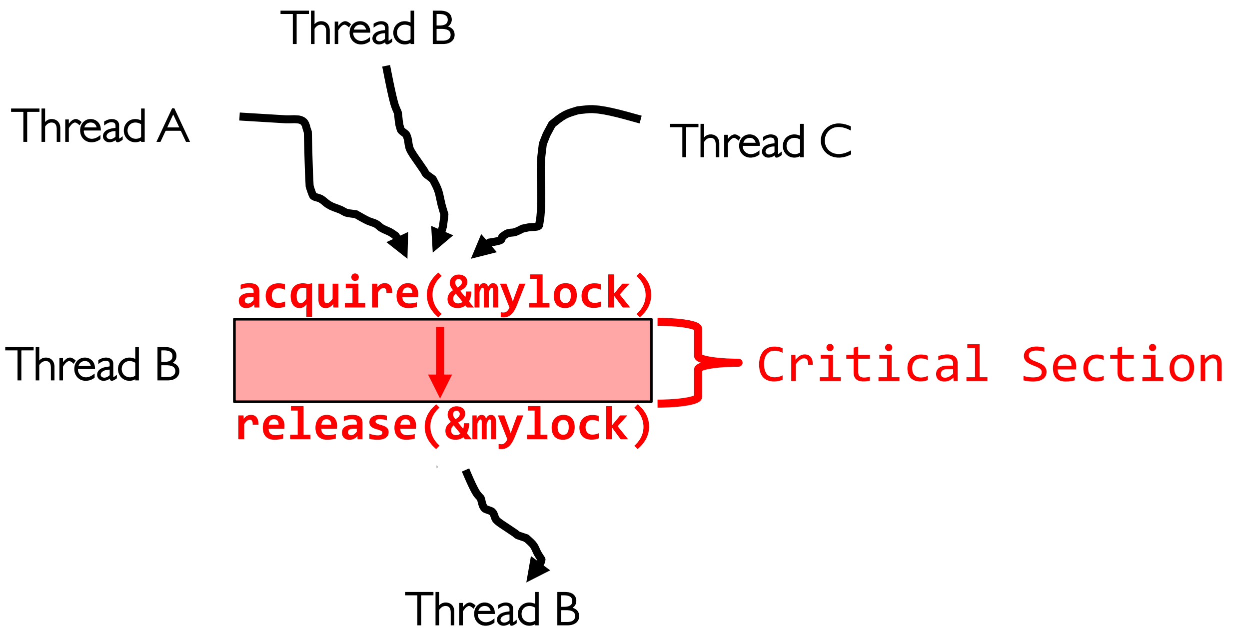
- ATM server problem:
  - Service a set of requests
  - Do so without corrupting database
  - Don’t hand out too much money

Recap: Fix banking problem with Locks!

- Identify critical sections (atomic instruction sequences) and add locking:

```
Deposit(acctId, amount) {
  acquire(&mylock)              // Wait if someone else in critical section!
  acct = GetAccount(actId);
  acct->balance += amount;
  StoreAccount(acct);
  release(&mylock)              // Release someone into critical section
}
```

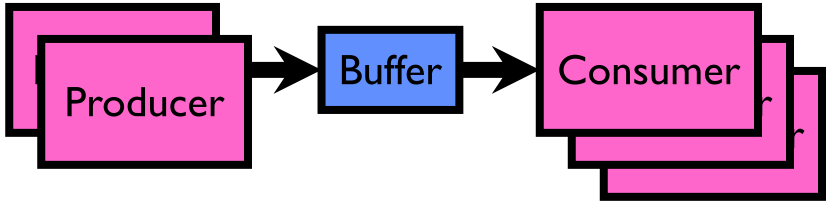
Critical Section



Threads serialized by lock through critical section.
Only one thread at a time

- Must use SAME lock (`mylock`) with all of the methods (Withdraw, etc…)
  – Shared with all threads!

Recap: Producer-Consumer with a Bounded Buffer



- Problem Definition
  - Producer(s) put things into a shared buffer
  - Consumer(s) take them out
  - Need synchronization to coordinate producer/consumer
- Don’t want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
  - Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty
- Example: Coke machine
  - Producer can put limited number of Cokes in machine
  - Consumer can’t take Cokes out if machine is empty
- Others: Web servers, Routers, ….

Recap: Circular Buffer – first cut

```
mutex buf_lock = <initially unlocked>

Producer(item) {
  acquire(&buf_lock);
  while (buffer full) {}; // Wait for a free slot
  enqueue(item);
  release(&buf_lock);
}


Consumer() {
  acquire(&buf_lock);
  while (buffer empty) {}; // Wait for arrival
  item = dequeue();
  release(&buf_lock);
  return item;
}
```

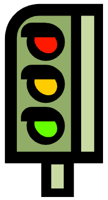
Will we ever come out of the wait loop?

Recap: Circular Buffer – 2nd cut

```
mutex buf_lock = <initially unlocked>

Producer(item) {
  acquire(&buf_lock);
  while (buffer full) {release(&buf_lock); acquire(&buf_lock);}
  enqueue(item);
  release(&buf_lock);
}

Consumer() {
  acquire(&buf_lock);
  while (buffer empty) {release(&buf_lock); acquire(&buf_lock);}
  item = dequeue();
  release(&buf_lock);
  return item;
}
```

What happens when one is waiting for the other?

Recall: Semaphores



- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX
- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - Down() or P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
    - » Think of this as the wait() operation
  - Up() or V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    - » This of this as the signal() operation
  - Note that P() stands for "*proberen*" (to test) and V() stands for "*verhogen*" (to increment) in Dutch

Group Discussion

```
Producer(item) {
  enqueue(item);
}

Consumer() {
  item = dequeue();
  return item;
}
```

- Topic: Circular Buffer
  - How to implement it with locks?
  - How to implement it with semaphores?
  - What are the pros and cons of each solution?

- Discuss in groups of two to three students
  - Each group chooses a leader to summarize the discussion
  - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

Revisit Bounded Buffer: Correctness constraints for solution

- Correctness Constraints:
  - Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
  - Producer must wait for consumer to empty buffers, if all full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)
- Remember why we need mutual exclusion
  - Because computers are stupid
  - Imagine if in real life: the delivery person is filling the machine and somebody comes up and tries to stick their money into the machine
- General rule of thumb: Use a separate semaphore for each constraint
  - `Semaphore fullBuffers; // consumer's constraint`
  - `Semaphore emptyBuffers;// producer's constraint`
  - `Semaphore mutex;       // mutual exclusion`

Full Solution to Bounded Buffer (coke machine)


```
Semaphore fullSlots = 0;        // Initially, no coke
Semaphore emptySlots = bufSize;
                                // Initially, num empty slots
Semaphore mutex = 1;            // No one using machine

Producer(item) {
    semaP(&emptySlots);         // Wait until space
    semaP(&mutex);              // Wait until machine free
    Enqueue(item);
    semaV(&mutex);
    semaV(&fullSlots);          // Tell consumers there is
                                // more coke
}
Consumer() {
    semaP(&fullSlots);          // Check if there's a coke
    semaP(&mutex);              // Wait until machine free
    item = Dequeue();
    semaV(&mutex);
    semaV(&emptySlots);         // tell producer need more
    return item;
}
```

`fullSlots` signals coke

`emptySlots` signals space

Critical sections using mutex protect integrity of the queue

Discussion about Solution

- Why asymmetry?
  - Producer does: **semaP(&emptyBuffer), semaV(&fullBuffer)**
  - Consumer does: **semaP(&fullBuffer), semaV(&emptyBuffer)**

Decrease # of empty slots

Increase # of occupied slots

Decrease # of occupied slots

Increase # of empty slots

- Is order of P's important?
  - Yes! Can cause deadlock
- Is order of V's important?
  - No, except that it might affect scheduling efficiency
- What if we have 2 producers or 2 consumers?
  - Do we need to change anything?

```
Producer(item) {
  semaP(&mutex);
  semaP(&emptySlots);
  Enqueue(item);
  semaV(&mutex);
  semaV(&fullSlots);
}
Consumer() {
  semaP(&fullSlots);
  semaP(&mutex);
  item = Dequeue();
  semaV(&mutex);
  semaV(&emptySlots);
  return item;
}
```

Where are we going with synchronization?

Programs	Shared Programs
Higher-level API	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Compare&Swap

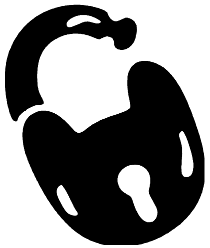
- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

Motivating Example: “Too Much Milk”

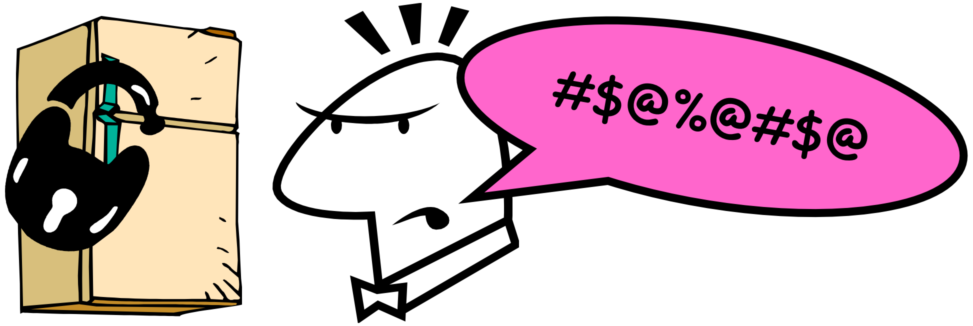
- Great thing about OS’s – analogy between problems in OS and problems in real life
  - Help you understand real life problems better
  - But, computers are much stupider than people
- Example: People need to coordinate:

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

Recall: What is a lock?



- Lock: prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - » Important idea: all synchronization involves waiting
- For example: fix the milk problem by putting a key on the refrigerator
  - Lock it and take key if you are going to go buy milk
  - Fixes too much: roommate angry if only wants orange juice



- Of Course – We don't know how to make a lock yet
  - Let's see if we can answer this question!

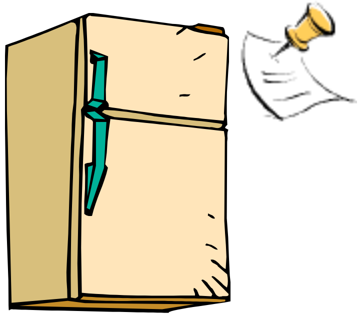
Too Much Milk: Correctness Properties

- Need to be careful about correctness of concurrent programs, since non-deterministic
  - Impulse is to start coding first, then when it doesn’t work, pull hair out
  - Instead, think first, then code
  - Always write down behavior first
- What are the correctness properties for the “Too much milk” problem???
  - Never more than one person buys
  - Someone buys if needed
- First attempt: Restrict ourselves to use only atomic load and store operations as building blocks

Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of "lock")
  - Remove note after buying (kind of "unlock")
  - Don't buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

```
if (noMilk) {
   if (noNote) {
      leave Note;
      buy Milk;
      remove Note;
   }
}
```



Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of "lock")
  - Remove note after buying (kind of "unlock")
  - Don't buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

<u>Thread A</u>

```
if (noMilk) {


   if (noNote) {
     leave Note;
     buy Milk;
     remove Note;
   }
}
```

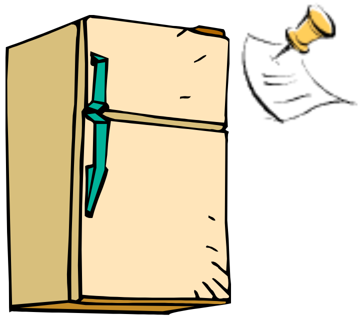
<u>Thread B</u>

```
if (noMilk) {
   if (noNote) {






      leave Note;
      buy Milk;
      remove Note;
   }
}
```

Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of "lock")
  - Remove note after buying (kind of "unlock")
  - Don't buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

```
if (noMilk) {
   if (noNote) {
      leave Note;
      buy Milk;
      remove Note;
   }
}
```



- Result?
  - Still too much milk but only occasionally!
  - Thread can get context switched after checking milk and note but before buying milk!
- Solution makes problem worse since fails intermittently
  - Makes it really hard to debug…
  - Must work despite what the dispatcher does!

Too Much Milk: Solution #1½

- Clearly the Note is not quite blocking enough
  - Let's try to fix this by placing note first
- Another try at previous solution:

```
leave Note;
if (noMilk) {
   if (noNote) {
      buy Milk;
   }
}
remove Note;
```

- What happens here?
  - Well, with human, probably nothing bad
  - With computer: no one ever buys milk

Too Much Milk Solution #2

- How about labeled notes?
  - Now we can leave note before checking
- Algorithm looks like this:

Thread A

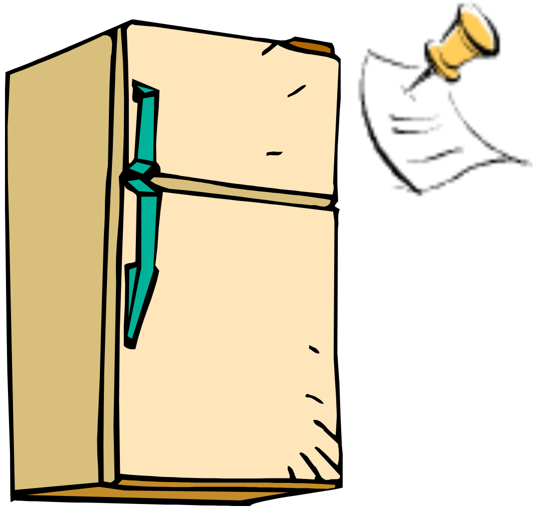
```
leave Note A;
if (noNote B) {
   if (noMilk) {
      buy Milk;
   }
}
remove Note A;
```

Thread B

```
leave Note B;
if (noNote A) {
   if (noMilk) {
      buy Milk;
   }
}
remove Note B;
```

- Does this work?
- Possible for neither thread to buy milk
  - Context switches at exactly the wrong times can lead each to think that the other is going to buy
- Really insidious:
  - Extremely unlikely this would happen, but will at worse possible time
  - Probably something like this in UNIX

Too Much Milk Solution #2: problem!



- *I'm* not getting milk, *You're* getting milk
- This kind of lockup is called "starvation!"

Too Much Milk Solution #3

- Here is a possible two-note solution:

Thread A

```
leave Note A;
while (Note B) {\\X
   do nothing;
}
if (noMilk) {
   buy Milk;
}
remove Note A;
```

Thread B

```
leave Note B;
if (noNote A) {\\Y
   if (noMilk) {
      buy Milk;
   }
}
remove Note B;
```

- Does this work? Yes. Both can guarantee that:
  - It is safe to buy, or
  - Other will buy, ok to quit
- At X:
  - If no note B, safe for A to buy,
  - Otherwise wait to find out what will happen
- At Y:
  - If no note A, safe for B to buy
  - Otherwise, A is either buying or waiting for B to quit

Case 1

- "`leave note A`" happens before "`if (noNote A)`"

```
leave Note A;
while (Note B) {\\X
    do nothing;
};




if (noMilk) {
    buy Milk;}
}
remove Note A;
```

happened before

```
leave Note B;
if (noNote A) {\\Y
    if (noMilk) {
        buy Milk;
    }
}
remove Note B;
```

Case 1

- "`leave note A`" happens before "`if (noNote A)`"

```
leave Note A;
while (Note B) {\\X
    do nothing;
};
```

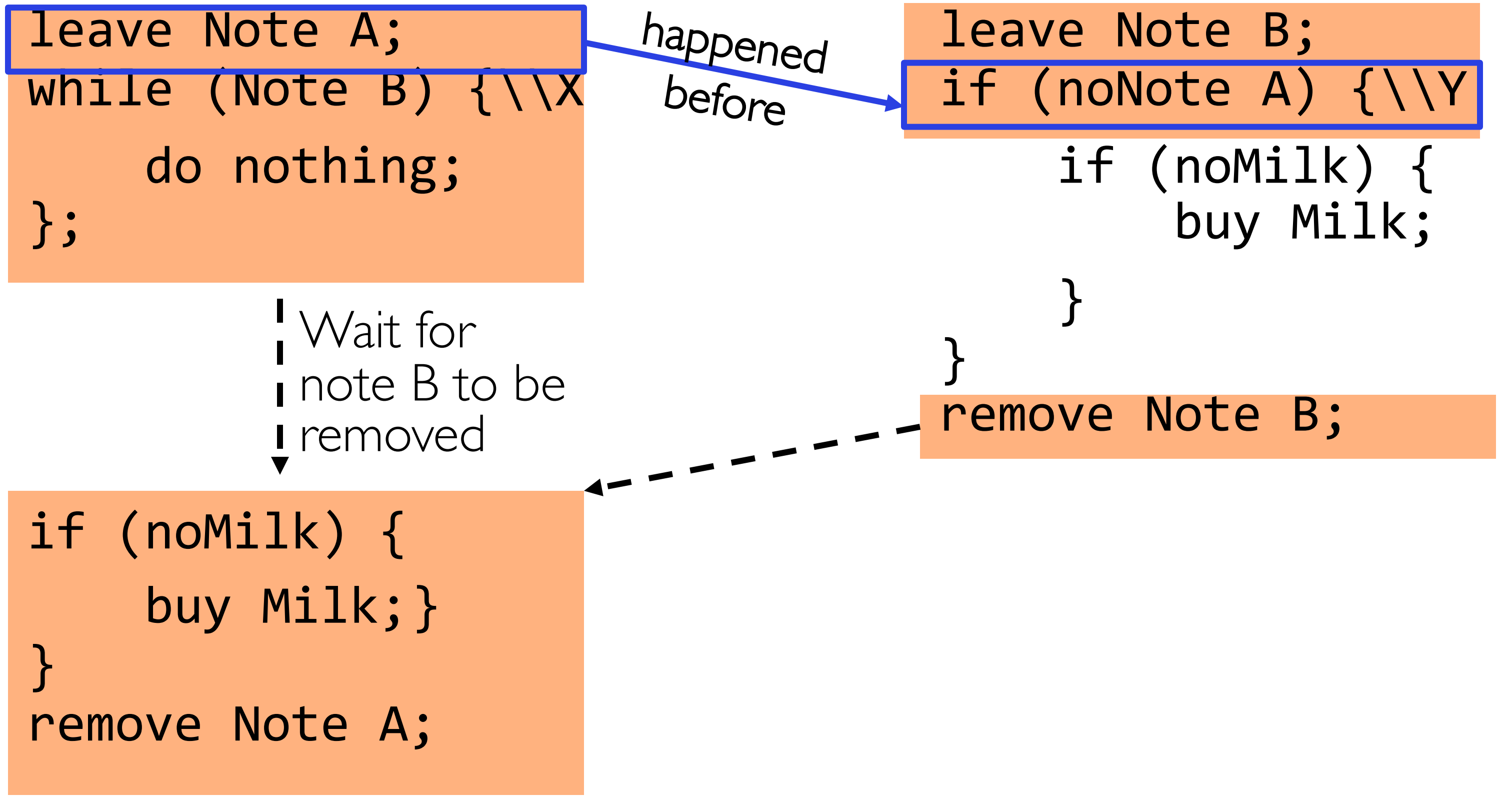
happened before

```
leave Note B;
if (noNote A) {\\Y
    if (noMilk) {
        buy Milk;
    }
}
remove Note B;
```

```
if (noMilk) {
    buy Milk;}
}
remove Note A;
```

Case 1

- "`leave note A`" happens before "`if (noNote A)`"



Case 2

- "`if (noNote A)`" happens before "`leave note A`"

```
leave Note A;
while (Note B) {\\X
    do nothing;
};


if (noMilk) {
    buy Milk;}
}
remove Note A;
```

happened before

```
leave Note B;
if (noNote A) {\\Y
        if (noMilk) {
            buy Milk;
        }
}
remove Note B;
```

Case 2

- "`if (noNote A)`" happens before "`leave note A`"

```
leave Note B;
if (noNote A) {\\Y
     if (noMilk) {
          buy Milk;
     }
}
remove Note B;
```

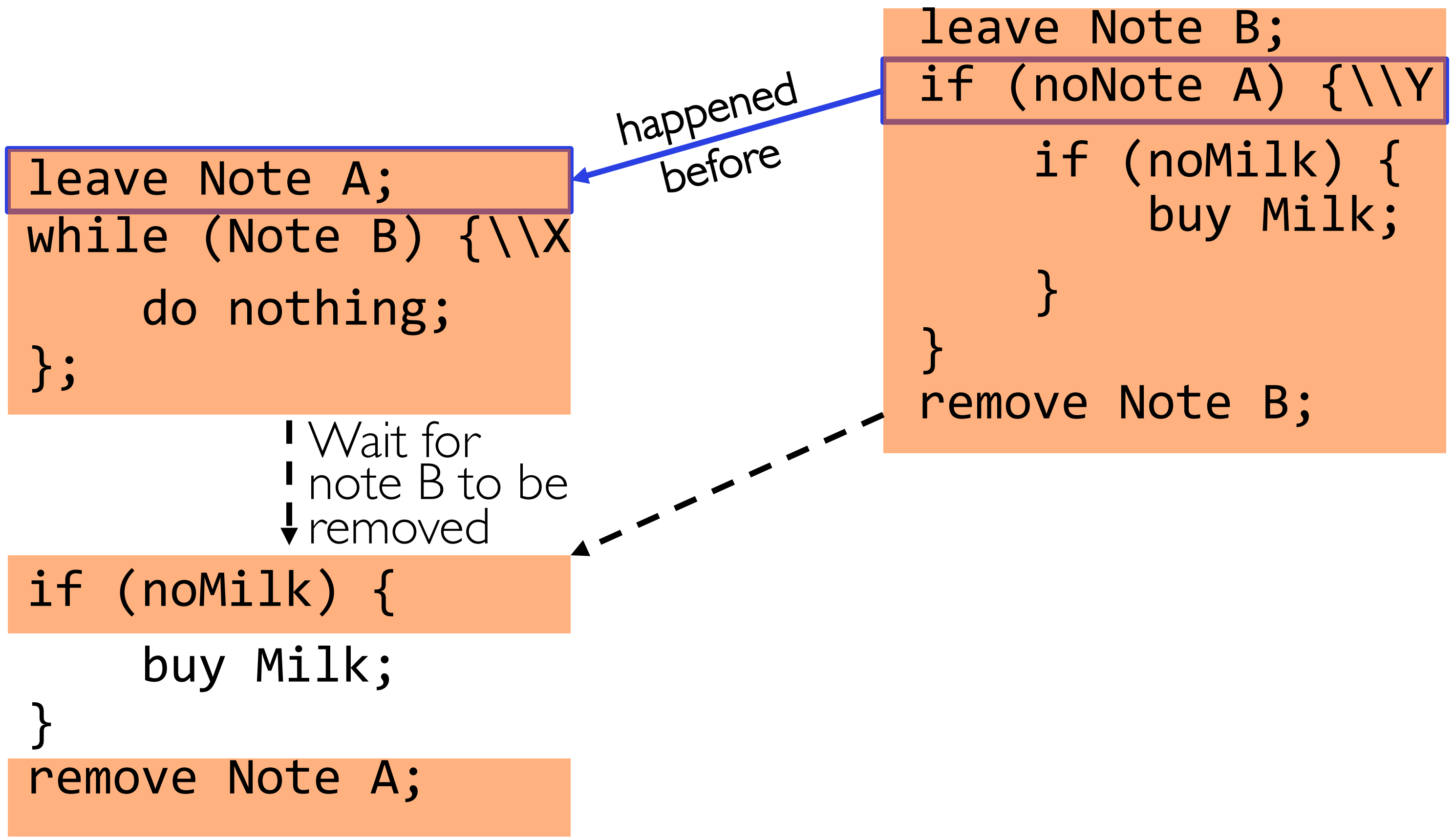
happened before

```
leave Note A;
while (Note B) {\\X
     do nothing;
};
```

```
if (noMilk) {
     buy Milk;
}
remove Note A;
```

Case 2

- "`if (noNote A)`" happens before "`leave note A`"



Solution #3 discussion

- Our solution protects a single "Critical-Section" piece of code for each thread:

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

- Solution #3 works, but it's really unsatisfactory
  - Really complex – even for this simple an example
    - » Hard to convince yourself that this really works
  - A's code is different from B's – what if lots of threads?
    - » Code would have to be slightly different for each thread
  - While A is waiting, it is consuming CPU time
    - » This is called "busy-waiting"
- There's got to be a better way!
  - Have hardware provide higher-level primitives than atomic load & store
  - Build even higher-level programming abstractions on this hardware support

Summary

- Synchronization: using atomic operations to ensure cooperation between threads
- Mutual Exclusion: ensuring that only one thread does a particular thing at a time
  - One thread *excludes* the other while doing its task
- Critical Section: piece of code that only one thread can execute at once. Only one thread at a time will get into this section of code
- Locks: synchronization mechanism for enforcing mutual exclusion on critical sections to construct atomic operations
- Semaphores: synchronization mechanism for enforcing resource constraints
- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives