Winter 2025

EECS 482: Introduction to Operating Systems

Lecture 3: Synchronization: Mutual Exclusion

Prof. Ryan Huang

Administration

Project 1 starts today (due in two weeks)

- Write a concurrent program
- Material covered this week. For now:
  - Read handout
  - Compile and run the sample program
  - Create threads

Declare project groups by January 22nd

Too much milk

Problem definition

- Alice and Bob want to keep their refrigerator stocked with at most one milk jug
- If either sees fridge empty, they go to buy milk

Solution #0

Alice

```
if (milk == 0) { // no milk
   milk++; // buy milk
}
```

Bob

```
if (milk == 0) { // no milk
   milk++; // buy milk
}
```

Problem?

A real-life scenario

	Alice	Bob
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. Oh no!

Mutual Exclusion

Problem: concurrent threads accessed a shared resource without any synchronization

- Known as a race condition

Mutual exclusion

- First type of synchronization we'll cover
- Ensure only one thread is doing a certain thing at a time
  - E.g., only 1 person goes shopping at a time
- Constrain interleavings of threads
  - No two threads can do the certain thing *at the same time*
- Allow us to have larger atomic blocks

Critical section

A section of code that uses mutual exclusion to synchronize its execution is called a critical section

- Only one thread at a time can execute in the critical section
- All other threads are forced to wait on entry
- When a thread leaves a critical section, another can enter

In Too much milk, critical section is:

```
if (no milk) {
    buy milk
}
```

Critical section requirements

1) Mutual exclusion

- If one thread is in the critical section, then no other is

2) Progress

- If some thread *T* is *not* in the critical section, then *T* cannot prevent some other thread *S* from entering the critical section
- A thread in the critical section will eventually leave it

3) Bounded waiting (no starvation)

- If some thread *T* is waiting on the critical section, then *T* will eventually enter the critical section

4) Performance

- The overhead of entering and exiting the critical section is small with respect to the work being done within it

About the requirements

There are three kinds of requirements that we'll use

Safety property: nothing bad happens

- Mutual exclusion

Liveness property: something good eventually happens

- Progress, Bounded Waiting

Performance requirement

Properties hold for each run, while performance depends on all the runs

- *Rule of thumb:* When designing a concurrent algorithm, worry about safety first, but don't forget about liveness!

Solution #1: Leave a note

Leave note that you're going to check on the milk, so the other person doesn't also buy

- Assume check note, leave note, and remove note each is atomic

Alice

```
if (note == 0) {      // if no note
    note = 1;         // leave note
    if (milk == 0) {  // if no milk
        milk++;       // buy milk
    }
    note = 0;         // remove note
}
```

Bob

```
if (note == 0) {
    note = 1;
    if (milk == 0) {
        milk++;
    }
    note = 0;
}
```

Is this safe?

Solution #1: Leave a note

Alice

```
if (note == 0) {


    note = 1;

    if (milk == 0) {

        milk++;
    }
    note = 0;

}
```

Bob

```
if (note == 0) {
    note = 1;

    if (milk == 0) {

        milk++;
    }


    note = 0;
}
```

Is this safe?

Solution #2: Leave two notes

Notes need to be labelled

- Otherwise you'll see your note and think the other person left it

Change the order of "leave note" and "check note"

Alice

```
noteA = 1;
if (noteB == 0) {
    if (milk == 0) {
        milk++;
    }
}
noteA = 0;
```

Bob

```
noteB = 1;
if (noteA == 0) {
    if (milk == 0) {
        milk++;
    }
}
noteB = 0;
```

Is this safe?

Solution #2: Leave two notes

Alice

```
noteA = 1;
if (noteB == 0) {


    if (milk == 0) {
        milk++;
    }
}
noteA = 0;
```

Bob

```
noteB = 1;
if (noteA == 0) {
    if (milk == 0) {
        milk++;
    }
}
noteB = 0;
```

Is this safe?

Solution #2: Leave two notes

Alice

```
noteA = 1;
if (noteB == 0) {
    if (milk == 0) {
        milk++;
    }
}
noteA = 0;
```

Bob

```
noteB = 1;
if (noteA == 0) {
    if (milk == 0) {
        milk++;
    }
}
noteB = 0;
```

Does it ensure liveness?

Solution #2: Leave two notes

Alice

```
noteA = 1;

if (noteB == 0) {
    if (milk == 0) {
        milk++;
    }
}
noteA = 0;
```

Bob

```
noteB = 1;

if (noteA == 0) {
    if (milk == 0) {
        milk++;
    }
}
noteB = 0;
```

No one buys milk!

Does it ensure liveness?

Solution #3: Monitor notes

Decide who will buy milk when both leave notes at the same time

Alice

```
noteA = 1;
while (noteB == 1) {
  // do nothing
}
if (milk == 0) {
   milk++;
}
noteA = 0;
```

Bob

```
noteB = 1;
if (noteA == 0) {
   if (milk == 0) {
      milk++;
   }
}
noteB = 0;
```

Is this safe?

Does it ensure liveness?

Solution #3: Monitor notes

Alice

```
noteA = 1;

while (noteB == 1) {
  // do nothing






}
if (milk == 0) {
   milk++;
}
noteA = 0;
```

Bob

```
noteB = 1;
if (noteA == 0) {
   if (milk == 0) {
      milk++;
   }
}
noteB = 0;
```

Does it ensure liveness?

Solution #3: Monitor notes

Alice

```
noteA = 1;
while (noteB == 1) {
  // do nothing




}
if (milk == 0) {
   milk++;
}
noteA = 0;
```

Bob

```
noteB = 1;
if (noteA == 0) {


   if (milk == 0) {
      milk++;
   }
}
noteB = 0;
```

Does it ensure liveness?

Proof of correctness

Bob

- If `noteA` is 0, Alice has not started yet → it is safe to buy
  - Alice will wait for Bob to be done before checking milk
- If `noteA` is 1, Alice has started and will buy milk if needed → it is safe to remove `noteB`
  - Alice might need to wait until `noteB` is removed

Alice

- If `noteB` is 0, it is safe to check & buy
  - Alice has set noteA to 1, and Bob will check noteA in the future
- If `noteB` is 1, Alice waits to see what Bob does
  1) Bob has checked `noteA` *before* Alice left note → Bob will buy milk
  2) Bob has checked `noteA` *after* Alice left note → Bob will *not* buy milk (Alice will)

Analysis of solution #3

Pros

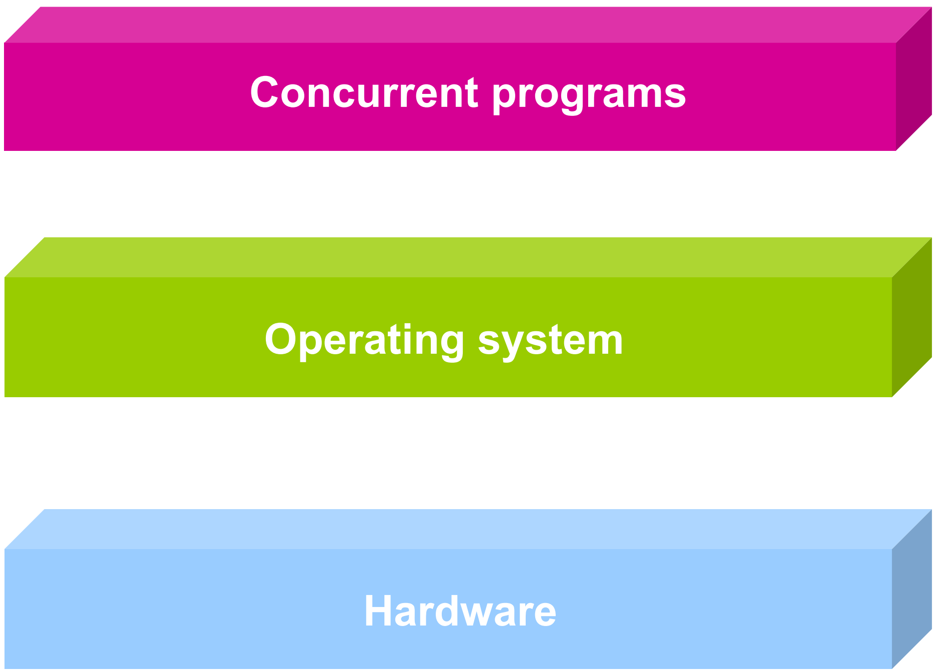
- It works!
- Works even if most operations are not atomic

Cons

- Complicated; not obviously correct
- Asymmetric
- Not obvious how to scale to three people
- Alice consumes CPU time while waiting. This is called busy-waiting.

Higher-level synchronization

Raise the level of abstraction to make life easier for programmers



Higher-level synchronization operations
(lock, condition variable, semaphore)

Atomic operations
(load, store, interrupt enable/disable, test&set)

Locks (mutexes)

A lock prevents another thread from entering a critical section

- *"Lock fridge while checking milk status and shopping"*

Two operations

- lock(): wait until lock is free, then acquire it

```
while (1) {
    if (lock is free) {
        acquire lock
        break
    }
}
```

atomic

- unlock(): `release lock`

atomic

Problems? Hint: why was the note in *Too much milk* (solutions #1 and #2) not a good lock?

Locks (mutexes)

How to use locks

- Lock is initialized to free
- Thread should acquire lock before entering critical section
- Thread that acquired lock should release lock when done with critical section

```
lock()
<critical section>
unlock()
```

All synchronization involves waiting

Thread can be running or blocked

Locks make “Too much milk” easy!

```
mutex milk_mu;
```

Alice

```
milk_mu.lock();
if (milk == 0) { // no milk
   milk++; // buy milk
}
milk_mu.unlock();
```

Bob

```
milk_mu.lock();
if (milk == 0) {
   milk++;
}
milk_mu.unlock();
```

Efficiency

But this prevents one from doing things while another is buying milk (which may take time)

```
milk_mu.lock();
if (milk == 0) { // no milk
   milk++; // buy milk
}
milk_mu.unlock();
```

```
milk_mu.lock();
if (milk == 0) {
   milk++;
}
milk_mu.unlock();
```

How to minimize the time the lock is held?

Efficiency – remember solution #1?

```
mutex note_mu;
```

```
note_mu.lock();
if (note == 0) {     // if no note
   note = 1;         // leave note
   note_mu.unlock();

   if (milk == 0) { // if no milk
      milk++;       // buy milk
   }

   note_mu.lock();
   note = 0;        // remove note
}
note_mu.unlock();
```