Today

- Synchronization
  - Formalizing synchronization
  - Locks

Project 3: Disk Directory Visualization

`floppya.img`

0	Bootloader
1	Disk Map
2	Disk Dir
3	KERNEL
…	…
19	message.txt
20	Bigfile
21	Bigfile
22	Bigfile
23	FILE
…	…
…	…

Disk Directory ←how big is this?

Conceptually, 32 characters "wide"

K	E	R	N	E	L	3	4	5	6	…
m	e	s	s	a	g	19	0	0	0	…
B	i	g	f	i	l	20	21	22	0	…
F	I	L	E	0	0	23	0	0	0	…
…										

16 entries

You can see what your disk directory looks like by using `hexdump -C floppya.img`

Review

- What problem were we trying to solve?
  - How did we try to solve it? What were some problems with the proposed solutions?
- What is
  - A schedule?
  - A race condition?
  - A critical section?
  - An atomic operation?
- Given a proposed solution, how do we know if it's right?

Review: Terminology

- ***Schedule***: an ordering/interleaving of instructions/events
- ***Atomic Operation***: an operation that is uninterruptible
- ***Synchronization***: Using atomic operations to ensure cooperation between threads
- ***Mutual Exclusion***: Exactly one thread (or process) is doing a particular activity at a time. Usually related to critical sections
- ***Critical Section***: A piece of code that only one thread can execute at a time

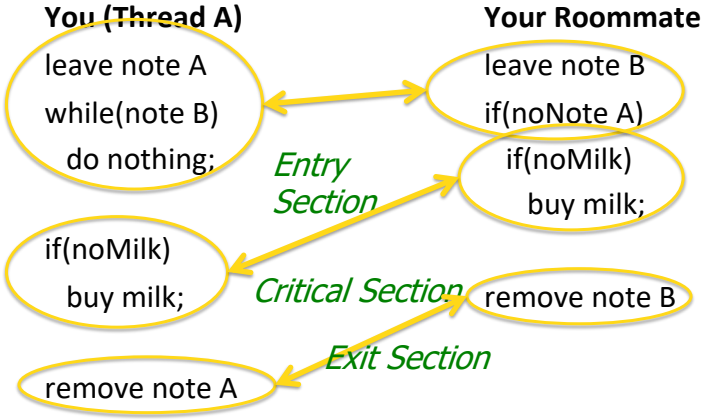
Formalizing "Responsible Roommates"

- Shared variable: `noMilk`
- Operations on shared variable
  - "Look in the fridge for milk" – check a variable
  - "Put milk away" – update a variable

Critical Section Problem

- Consider system of *n* processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has ***critical section*** segment of code
  - Process may be changing shared variables, updating table, writing file, etc.
- When one process is in critical section, **no** other may be in its critical section
- ***Critical section problem*** is to design protocol to ensure atomic execution of critical section

Formalizing "Responsible Roommates"



Critical Sections and Correctness

Four properties are required for correctness:

1. ***Safety/Mutual Exclusion***: only one thread in the critical section
2. ***Liveness/Progress***: if no threads are executing a critical section and a thread wishes to enter a critical section, that thread must be guaranteed to *eventually* enter the critical section
3. ***Bounded waiting***: if a thread wishes to enter a critical section, then there exists a bound on the number of other threads that may enter the critical section before that thread does
4. ***Failure atomicity***: it's okay for a thread to die in the critical section

What do "safety" and "liveness" mean in the outcome of the Responsible Roommate problem?

Formalizing "Responsible Roommates"

- Shared variable: `noMilk`
- Operations on shared variable
  - "Look in the fridge for milk" – check a variable
  - "Put milk away" – update a variable
- Safety property
  - At most one person buys milk
- Liveness
  - Someone buys milk when needed

Safety and Liveness, More Generally

- Properties defined over the execution of a program
- Safety: "nothing bad happens"
  - Holds in every finite execution prefix
    - Windows never crashes
    - No patient is ever given the wrong medication
    - A program never terminates with the wrong answer
- Liveness: "something good eventually happens"
  - No partial execution is irremediable
    - Windows always reboots
    - Medications are eventually distributed to patients
    - A program eventually terminates

Mutual Exclusion

- Exactly one thread (or process) is doing a particular activity at a time. Usually related to critical sections.
  - Active thread excludes its peers
- Some computer resources cannot be accessed by multiple threads at the same time
  - E.g., a printer can't print two documents at once
- For shared memory architectures, data structures are often mutually exclusive
  - Two threads adding to a linked list can corrupt the list

Critical Sections: When You Want Mutual Exclusion

Anytime you access shared data

- If a thread checks a value
  - Even if it is "just a quick" read
- If a thread updates a piece of shared data
  - *What data is shared?*

Atomic Operations

- Operations that are uninterruptible
  - Indivisible operations that cannot be interleaved with or split by other operations
  - Run to completion or not at all
- What operations are uninterruptible?
  - Essentially, only `load` and `store`
    - But not necessarily – depends on architecture

Assume not-explicitly-atomic statements are **not** atomic when you need synchronization

Our Ideal Solution

- Satisfies correctness properties
  - Safety, liveness, bounded wait
  - Easy to convince ourselves it does so
- No busy waiting (spin locks)
  - Threads should block when waiting and then be awakened when it is their turn (a *wait queue*)
- Extendable to many threads (not just two!)
  - Symmetric

Support for Synchronization

Most systems provide support for atomic routines for synchronization

- **Locks**: One thread holds a lock at a time, executes the critical section, releases the lock
- **Semaphores**: More general version of locks
- **Monitors**: Connects shared data to synchronization primitive

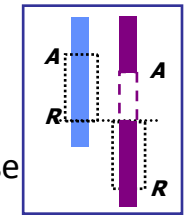
All require some hardware support (and waiting!)

Or Mutexes or Monitors

LOCKS

A Lock or Mutex

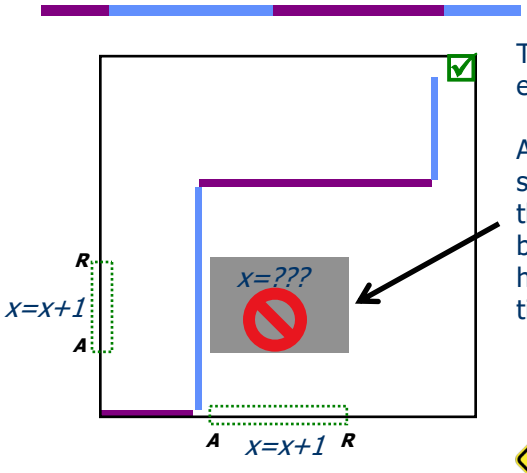
- Locks enforce **mutual exclusion** in conflicting critical sections
  - A special data item in memory
- API methods: `Acquire` and `Release`
  - Also called `Lock` and `Unlock`
- A thread should pair calls to Acquire and Release
  - `Acquire` upon entering a critical section
  - `Release` upon leaving a critical section
- Between `Acquire`/`Release`, the thread holds the lock
- `Acquire` does not return until any previous holder releases
- Waiting locks can spin (a spinlock) or block (a mutex)



Definition of a lock (mutex)

- Acquire + Release ops on lock L are strictly paired
  - After acquire completes, the caller holds (owns) the lock L until the matching release
- Acquire + release pairs on each lock are ordered
  - Total order: each lock L has at most one holder at any given time
  - That property is mutual exclusion; L is a mutex

Portrait of a Lock in Motion



The program may fail if it enters the grey box.

A lock (mutex) prevents the schedule from ever entering the gray box, ever:
both threads would have to hold the same lock at the same time, and locks don't allow that

Locks and Responsible Roommates

Our solution used notes as locks:

1. Leave a note (acquire a lock)
2. Remove a note (release the lock)
3. Do not buy any milk if there is a note (wait)

What would it look like with actual locks?

Responsible Roommates: Lock Solution

You (Thread A)

```
Lock->Acquire();
if(noMilk)
  buy milk;
Lock->Release();
```

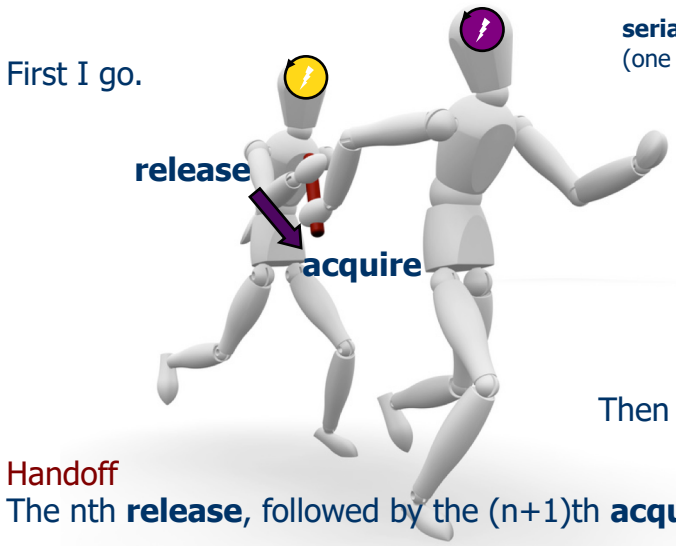
Your Roommate (Thread B)

```
Lock->Acquire();
if(noMilk)
  buy milk;
Lock->Release();
```

Do we know this works?

What if you wanted orange juice and your roommate wanted milk? How does the solution differ?

Handing off a lock



Handoff

The nth **release**, followed by the (n+1)th **acquire**

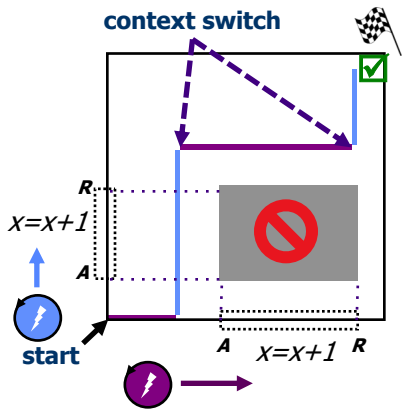
Mutual exclusion in Java

- Mutexes are built in to every Java object
- Every Java object is/has a ***monitor***
  - At most one thread may "own" a monitor at any given time.
- A thread becomes owner of an object's monitor by
  - executing an object method declared as `synchronized`
  - executing a block that is `synchronized` on the object

```
public synchronized void
    increment() {
        x = x + 1;
}
```

```
public void increment()  {
    synchronized(this) {
        x = x + 1;
    }
}
```


Lock It Down



Use a lock (**mutex**) to synchronize access to a data structure that is shared by multiple threads.

A thread **acquires** (*locks*) the designated mutex before operating on a given piece of shared data.

The thread **holds** the mutex. At most one thread can hold a given mutex at a time (**mutual exclusion**).

Thread **releases** (unlocks) the mutex when done. If another thread is waiting to acquire, then it wakes.

The mutex bars entry to the gray box: the threads cannot both hold the mutex.

Discussion: Why only Acquire/Release?

- The Lock API seems a little too simple
- Suppose we add a method to the Lock API that asks if the lock is free (`isFree`)
  - Suppose it returns true. Then what?

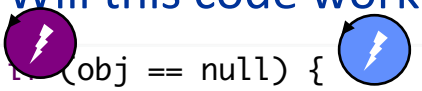
Will this code work?

Two threads are executing this [same] code.
Shared variable `obj` is initially `null`

```
if (obj == null) {
     lock.acquire();
      obj = newObj();
     lock.release();
}
obj.method();
```

```
Obj newObj() {
       obj = new Obj(…);
       obj.field1 = …
       obj.field2 = …
       return obj;
}
```

Will this code work?



```
if (obj == null) {
      lock.acquire();
       obj = newObj();
      lock.release();
}
obj.method();
```

```
Obj newobj() {
       obj = new Obj(…);
       obj.field1 = …
       obj.field2 = …
       return obj;
}
```

Consider:

- purple saw that obj was null, acquires the lock, and starts creating the new obj.
- Then purple is preempted.
- Blue then checks and sees that obj is *not null*, so it skips down to obj.method() *before* obj is done being initialized

Will this code work?

```
lock.acquire();
if (obj == null) {
      obj = newObj();
}
lock.release();
obj.method();
```

```
Obj newObj() {
       obj = new Obj(…);
       obj.field1 = …
       obj.field2 = …
       return obj;
}
```

Assume `method` is thread safe.

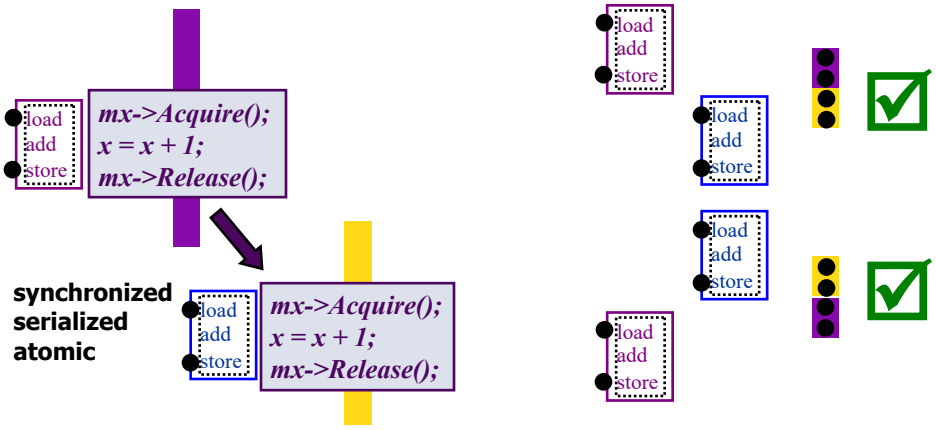
Will this code work? **YES!**

```
lock.acquire();
if (obj == null) {
      obj = newObj();
}
lock.release();
obj.method();
```

```
Obj newObj() {
      obj = new Obj(…);
      obj.field1 = …
      obj.field2 = …
      return obj;
}
```

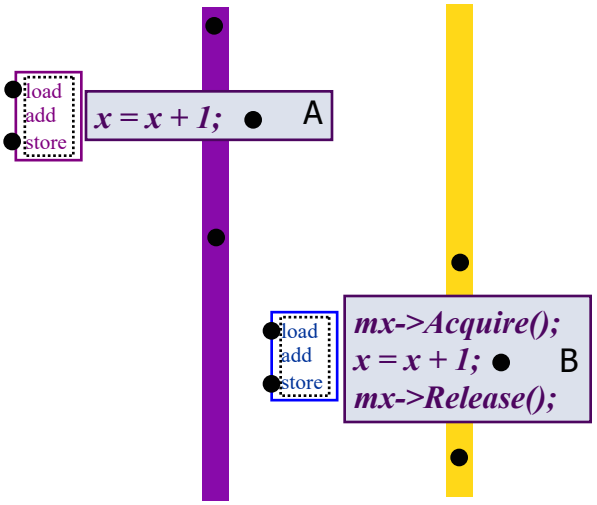
Assume `method` is thread safe.

Locking a critical section

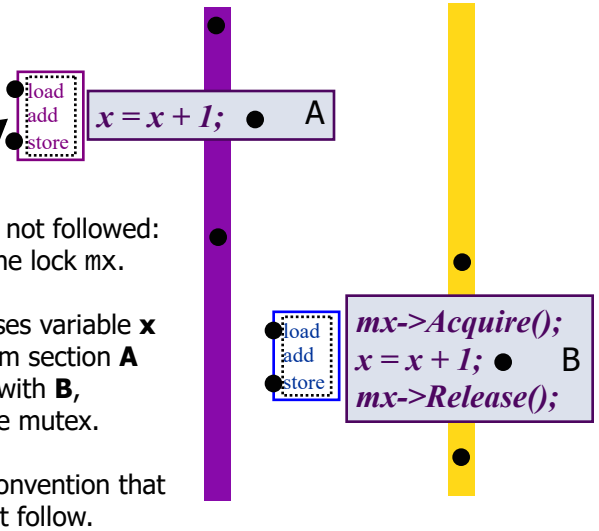


Holding a shared mutex prevents competing threads from entering a critical section. If the critical section code acquires the mutex, then its execution is serialized: only one thread runs it at a time.

Does this work?



Does this work? **NO!**

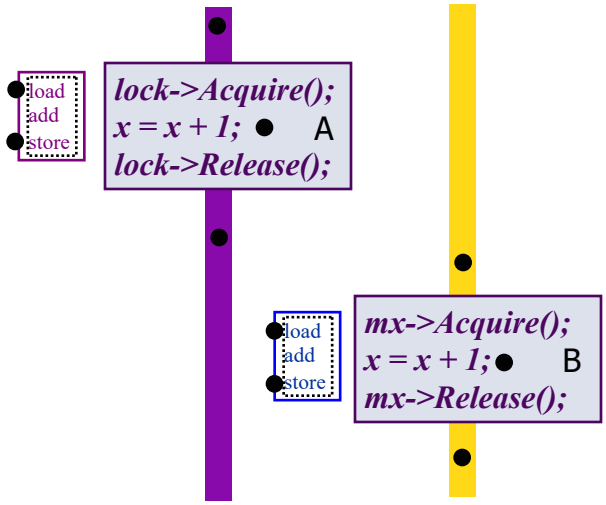


The locking discipline is not followed: purple fails to acquire the lock mx.

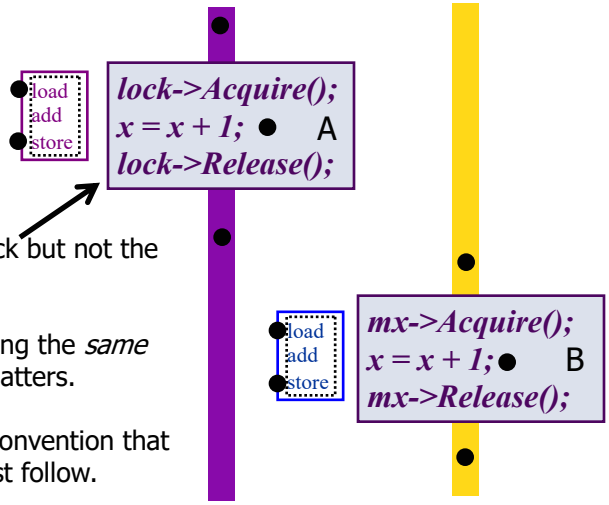
Or rather: purple accesses variable **x** through another program section **A** that is mutually critical with **B**, but does not acquire the mutex.

A locking scheme is a convention that the entire program must follow.

Does this work?



Does this work? **NO!**



Purple is acquiring *a* lock but not the "right" lock.

The threads are not using the *same* lock, and that's what matters.

A locking scheme is a convention that the entire program must follow.

Revisiting a (Seemingly) Simple Program

```
l = new Lock();
x = 5;
```

```
l.acquire();
x=x+1;
print(x);
l.release();
```

```
l.acquire();
x=x+1;
print(x);
l.release();
```

What is the output?

Rules for Using Locks

- Lock is initially free
- **Always** `acquire` lock before accessing shared data structure
  - Likely: Beginning of procedure
- **Always** `release` after finished with shared data
  - Likely: End of procedure
  - Only the lock holder can release
- **Never** access shared data without lock

Debugging non-determinism

- Requires worst-case reasoning
  - Eliminate all ways for program to break
- Debugging is hard
  - Can't test all possible interleavings
  - Bugs may only happen sometimes
- Heisenbug
  - Re-running program may make the bug disappear
  - Doesn't mean it isn't still there!

Looking Ahead

- Project 3 due next Friday
  - Suggested intermediate deadline: Step 3 by today
  - Reload the page, since I add clarifications