Today

- Synchronization
  - Race conditions

Review: Consider a (Seemingly) Simple Program

```
x = 5;
```

Thread 1

```
x=x+1;
print(x);
```

Thread 2

```
x=x+1;
print(x);
```

Possible outputs:
- 6 7
- 6 6

- Why were these the possible outputs?
- Could 7 6 be a possible output?
- What is a ***race condition***?

Review:
Consider a (Seemingly) Simple Program

```
x = 5;
```

Thread 1

```
x=x+1;
print(x);
```

Thread 2

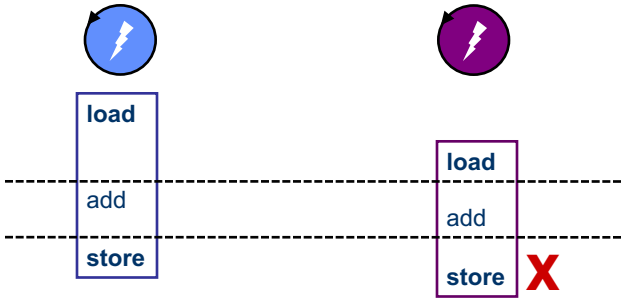
```
x=x+1;
print(x);
```

Possible outputs:
- 6 7
- 6 6

- Why were these the possible outputs?
  - x is a shared variable
  - x=x+1 is not an atomic operation
- Could 7 6 be a possible output?
  - Depends on if print(x) is an atomic operation
  - Don't assume atomicity

Review: Interleaving matters

```
load   x, R2        ; load global variable x
add    R2, 1, R2    ; increment: x = x + 1
store  R2, x        ; store global variable x
```

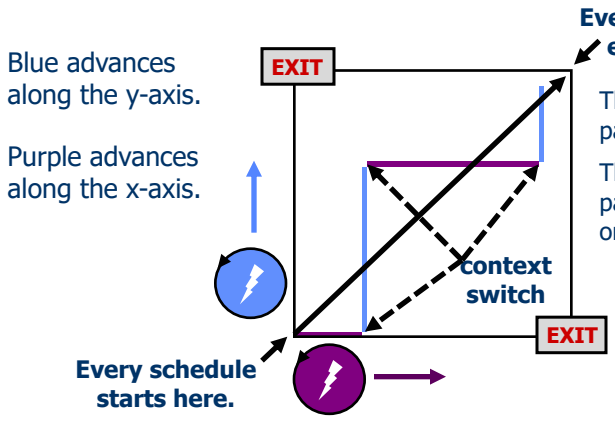


In this schedule, **x** is incremented only once: last writer wins.
The program breaks under this schedule. This bug is a ***race***.

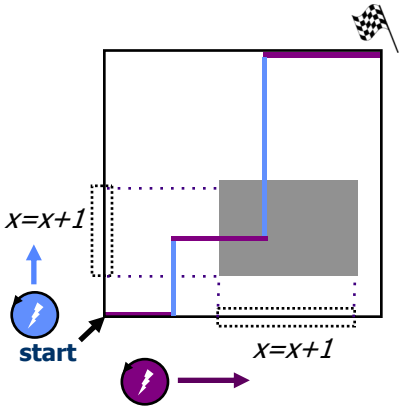
A ***race condition*** is any situation in which the order of execution affects the final result.

Resource Trajectory Graphs

This RTG depicts a schedule within the space of possible schedules for a simple program of two threads sharing one core.



A race



This is a valid schedule.
But the schedule interleaves the executions of "x = x + 1" in the two threads.

The variable **x** is shared.

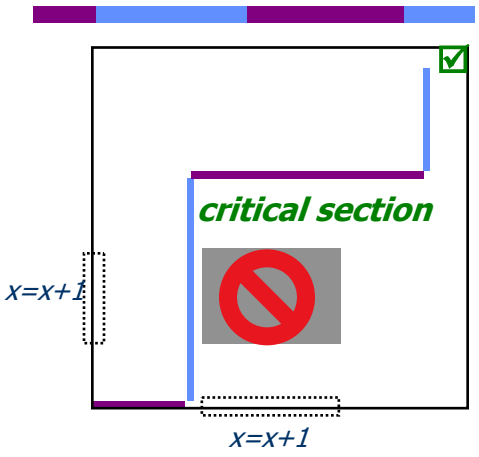
This schedule can corrupt the value of the shared variable x, causing the program to execute incorrectly.

This is an example of a ***race***: the behavior of the program depends on the schedule, and some schedules yield incorrect results.

Concurrency control

- The scheduler (and the machine) select the execution order of threads
- Each thread executes a sequence of instructions, but their sequences may be arbitrarily interleaved
  - E.g., from the point of view of loads/stores on memory
- Each possible execution order is a ***schedule***
- A ***thread-safe program*** must *exclude* schedules that lead to incorrect behavior
  - Called ***synchronization*** or ***concurrency control***

This is not a game

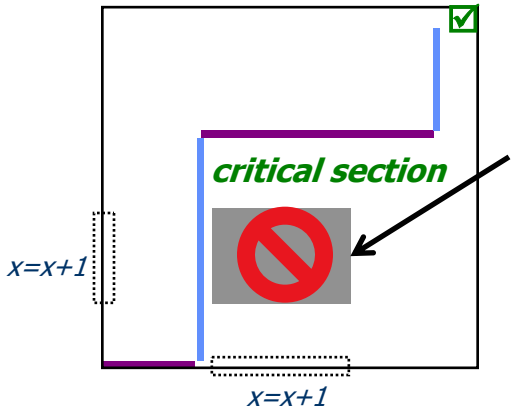


But we can think of it as a game

1. You write your program.
2. The game begins when you submit your program to your adversary: the scheduler.
3. The scheduler chooses all the moves while you watch.
4. Your program may constrain the set of legal moves.
5. The scheduler searches for a legal schedule that breaks your program.
6. If it succeeds, then you lose (your program has a **race**).
7. You win by not losing.

You should pretend to be the adversarial scheduler for your programs.

The need for mutual exclusion



The program may fail if the schedule enters the gray box, i.e., if two threads execute the critical section concurrently.

The two threads must not both operate on the shared global x "at the same time".

RESPONSIBLE ROOMMATES

Too Much Milk!

You	Your Roommate
• Arrive home	
• Look in the fridge; out of milk	
• Go to store	
	• Arrive home
• Buy milk	• Look in fridge; out of milk
	• Go to store
• Arrive home; put milk away	
	• Buy milk
	• Arrive home; put milk away
	• Oh, no!

What did we want to happen?
What happened?

Too Much Milk!

- What do we want to happen?
  - Only one person buys milk at a time AND
  - Someone buys milk if you need it

These are the correctness properties for this problem.

- What happened?
  - Lack of communication!

Analyzing Problem

- What would the result have been if:
  - your roommate had arrived home for the first time after you had come back from the store?
  - you arrived home after your roommate came back from the store?
  - you were at the store when your roommate came back, but your roommate looked in the fridge after you were back from the store?
- Example of a *race condition*
- What guarantees do we have about how our people/threads will be scheduled?
- How can we solve this problem?

Too Much Milk: Solution #1

noMilk = true
noNote = true

You (Thread A)

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

Your Roommate (Thread B)

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

Does this work?
How can you determine if it works?

Too Much Milk: Solution #1

noMilk = true

You (Thread A)

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

Your Roommate (Thread B)

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

How do you know if it works?

Create some schedules or show it does work.

This solution can work, **bUT**, not always.

And that's the issue.

Too Much Milk: Solution #2

noMilk = true
noteA = false
noteB = false

You (Thread A)

```
leave note A
if(no noteB)
  if(noMilk)
    buy milk;

remove note A
```

Your Roommate (Thread B)

```
leave note B
if(no noteA)
  if(noMilk)
    buy milk;

remove note B
```

Does this work?

Too Much Milk: Solution #2

noMilk = true
noteA = false
noteB = false

You (Thread A)

leave note A
if(no noteB)
  if(noMilk)
    buy milk;

remove note A

Your Roommate (Thread B)

leave note B
if(no noteA)
  if(noMilk)
    buy milk;

remove note B

Problem: Starvation!

Too Much Milk: Solution #3

noMilk = true
noteA = false
noteB = false

You (Thread A)

leave note A
while(noteB)
  do nothing;

if(noMilk)
  buy milk;

remove note A

Your Roommate (Thread B)

leave note B
if(no noteA)
  if(noMilk)
    buy milk;

remove note B

Does this work?

Too Much Milk: Solution #3

You (Thread A)

leave note A

while(note B)
  do nothing;

if(noMilk)
  buy milk;

remove note A

Your Roommate (Thread B)

leave note B

if(no noteA)
  if(noMilk)
    buy milk;

remove note B

Yes! Explain why it works! (harder than finding a schedule that breaks it)

Why is it correct?

Your Roommate (Thread B)

leave note B

if(noNote A)

  if(noMilk)

    buy milk;

remove note B

At this if, either there is a note A or not.

If there is a note, then thread A is checking and buying milk as needed or is waiting for B to quit, so B quits by removing note B.

If not, it is safe for B to check and buy milk, if needed. (Thread A has not started yet.)

Why is it correct?

You (Thread A)

leave note A

while(note B)
  do nothing;

if(noMilk)
  buy milk;

remove note A

At this while, either there is a note B or not.

If yes, A waits until there is no longer a note B, and either finds milk that B bought or buys it if needed.

If not, it is safe for A to buy since B has either not started yet or quit.

Why is it correct?

Thread B buys milk (which Thread A finds) or not, but either way it removes note B. Since Thread A loops, it waits for B to buy milk or not, and then if B did not buy it, it buys the milk.

So it's correct, but… is it good?

1. It is complicated. It was hard to convince ourselves this solution worked.
2. It is asymmetrical---thread A and thread B are different.
   - *What would we need to do to add new threads/roommates?*
3. A is *busy waiting*, or consuming CPU resources despite the fact it is not doing any useful work.

Too Much Milk: 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();
```

Acquiring and Releasing the Lock is an atomic operation

Terminology Review

- ***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

Critical Sections and Correctness

Four properties are required for correctness:

1. ***Safety***: only one thread in the critical section
2. ***Liveness***: 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

Looking Ahead

- Moved my Wed office hours to Thurs
- Project 3 due in two Fridays
  - Suggested intermediate deadline: Step 3 by Friday
  - Reload the page, since I add clarifications