Today

- Concurrency Problems
- Synchronization Mechanisms

Review

- Why should we considering writing multi-threaded programs?

Consider a (Seemingly) Simple Program

```
x = 5;
```

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

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

What is the output?

Consider a (Seemingly) Simple Program

```
x = 5;
```

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

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

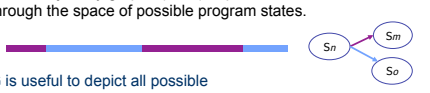
What is the output?

Possible outputs:

6 7
7 6
6 6

Resource Trajectory Graphs

Resource trajectory graphs (RTG) depict the "random walk" through the space of possible program states.

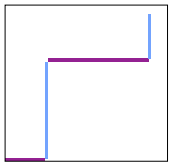


RTG is useful to depict all possible executions of multiple threads.
I will draw them for only two threads because slides are two-dimensional.

RTG for N threads is N-dimensional.

Thread *i* advances along axis *i*.

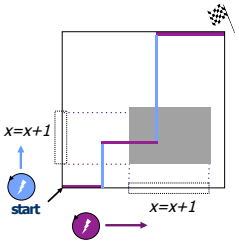
Each point represents one state in the set of all possible system states.



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.

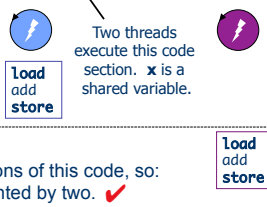
The variable x is shared.

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.

Reading Between the Lines of C

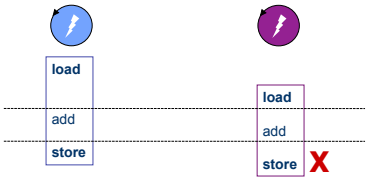
```
load   x, R2          ; load global variable x
add    R2, 1, R2      ; increment: x = x + 1
store  R2, x          ; store global variable x
```



Two executions of this code, so:
x is incremented by two. ✔

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***.

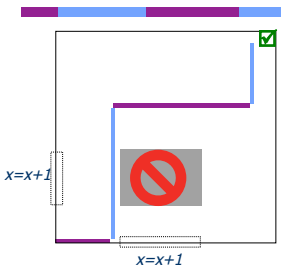
A ***race condition*** is any situation in which the order of execution affects the final result.

Concurrency control

The scheduler (and 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.
- ***Thread synchronization*** is the process of imposing synchronization constraints on otherwise concurrently executing threads.

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.

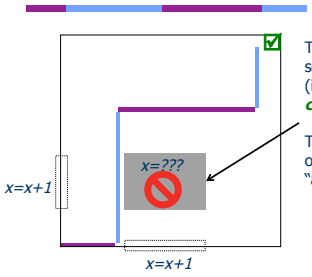
Discussion

- What do we need to do to prevent these issues?
- What are techniques you've seen that may apply to this problem?

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

The need for mutual exclusion

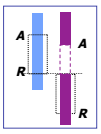


The program may fail if the schedule enters the grey box (i.e., if two threads execute the ***critical section*** concurrently).

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

A Lock or Mutex

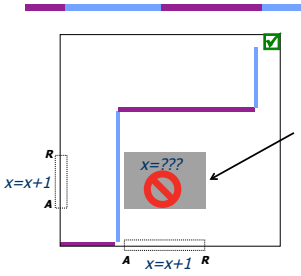
- Locks are the basic tools to enforce mutual exclusion in conflicting critical sections.
- A lock is a special data item in memory.
- API methods: `Acquire` and `Release`.
  - Also called `Lock` and `Unlock`.
- Threads pair calls to Acquire and Release.
  - `Acquire` upon entering a critical section.
  - `Release` upon leaving a critical section.
- Between `Acquire`/`Release`, the thread holds the lock.
- `Acquire` does not pass 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.

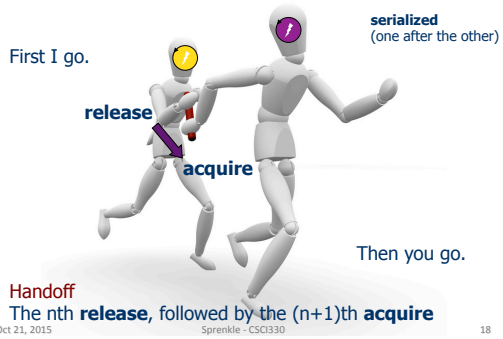
Portrait of a Lock in Motion



The program may fail if it enters the grey box.

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

Handing off a lock



serialized
(one after the other)

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

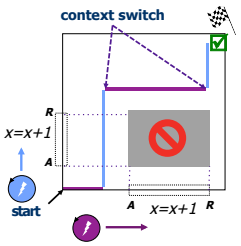
Mutual exclusion in Java

- 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 grey box: the threads cannot both hold the mutex.

Looking Ahead

- Project 3 – two Wednesdays