Today

- Project 3 - slides separate
- Threads
  - Thread Pools
- Synchronization
  - Race conditions

Review

- How do we create threads in Java?
- How does forking a process compare to spawning a thread?
  - How does the communication/collaboration change?

Example: Jabber

How many threads will run during the execution of this program?

```
class Jabber implements Runnable {
      String str;
      public Jabber(String s){ str = s; }
      public void run() {
            while (true) {
                  System.out.print(str);
                  System.out.println();
            }
      }
}
```

```
public class JabberTest {
      public static void main(String[] args) {
            Jabber jabber1 = new Jabber("1");
            Jabber jabber2 = new Jabber("2");
            Thread t1 = new Thread(jabber1);
            Thread t2 = new Thread(jabber2);
            t1.start();
            t2.start();
      }
}
```

JabberTest.java

Example: Jabber

How many threads will run during the execution of this program? **3**

```
class Jabber implements Runnable {
      String str;
      public Jabber(String s){ str = s; }
      public void run() {
            while (true) {
                  System.out.print(str);
                  System.out.println();
            }
      }
}
```

```
public class JabberTest {
      public static void main(String[] args) {
            Jabber jabber1 = new Jabber("1");
            Jabber jabber2 = new Jabber("2");
            Thread t1 = new Thread(jabber1);
            Thread t2 = new Thread(jabber2);
            t1.start();
            t2.start();
      }
}
```

JabberTest.java

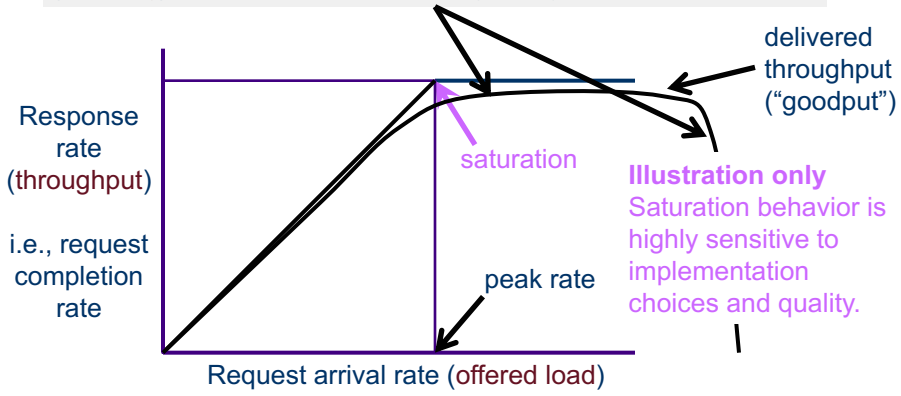
THREAD PERFORMANCE

Performance of Thread Maintenance

- Creating a thread is cheaper than creating a new process
  - But it's not free
- Maintaining thread information has an overhead cost
- Common performance issues
  - Creating threads on-the-fly incurs costs
    - increases latency of the task
  - Creating a lot of threads is costly (overhead, maintenance, switching)
    - Recall: throughput graph from scheduling

Review: Throughput: reality

Thrashing, also called congestion collapse
Real servers/devices often have some pathological behaviors at saturation. E.g., they abort requests after investing work in them (thrashing), which wastes work, reducing throughput.



Solution: Thread Pool

- Create a pool of threads before you need them
  - Incur that cost before work begins
- When a request comes in, assign to an available thread from the pool
  - When thread completes task, return thread to the thread pool
- In Java, *Executors*, *Executor* and *ExecutorService* classes
  - Executors: factory class to create Executor and ExecutorService objects

SYNCHRONIZATION

Consider a (Seemingly) Simple Program

```
x = 5;
```

Thread 1

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

Thread 2

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

What is the output?

Consider a (Seemingly) Simple Program

```
x = 5;
```

Thread 1

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

Thread 2

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

Possible outputs:

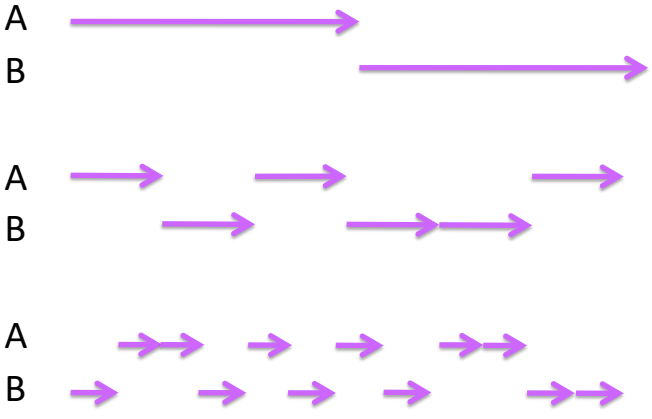
- 6 7
- 6 6

Why was this **not** an issue with processes and `fork`?

Threads and the Scheduler

(or, Why Multi-threaded Programming is Hard)

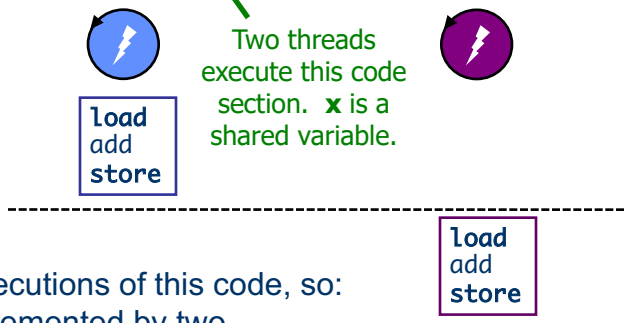
Given two threads, A and B, how might their executions be scheduled?



And, this is with just one processor…

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
```

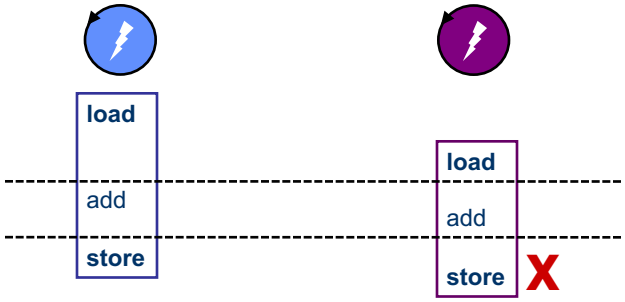


Two executions of this code, so:
x is incremented by two.

Other languages have similar low-level decompositions

Interleaving matters

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

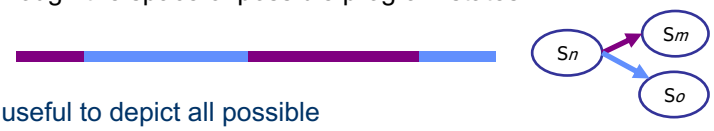


In this schedule, **x** is incremented only once: last writer wins.
The program breaks under this schedule. This bug is a ***race***.

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

Resource Trajectory Graphs

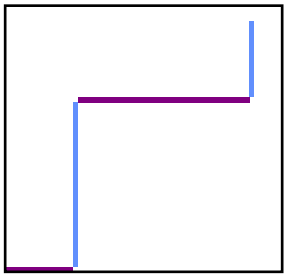
Resource trajectory graphs (RTG) depict the "random walk" through the space of possible program states.



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.

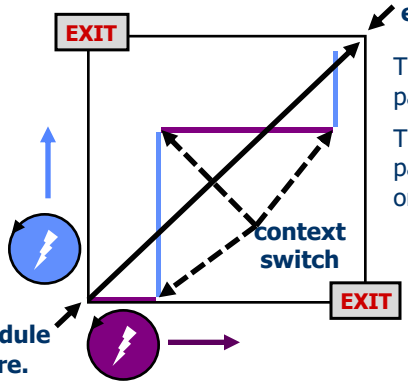


Resource Trajectory Graphs

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

Blue advances along the y-axis.

Purple advances along the x-axis.



Every schedule ends here.

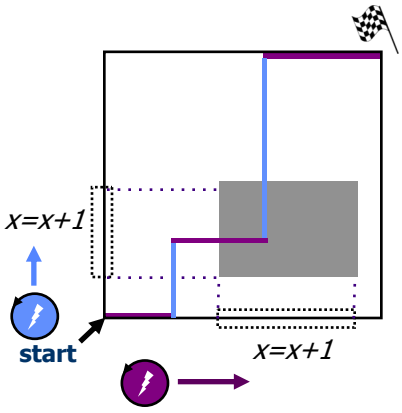
The diagonal is an idealized parallel execution (two cores).

The scheduler chooses the path (schedule, event order, or interleaving).

From the point of view of the program, the chosen path is **nondeterministic**.

Every schedule starts here.

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.

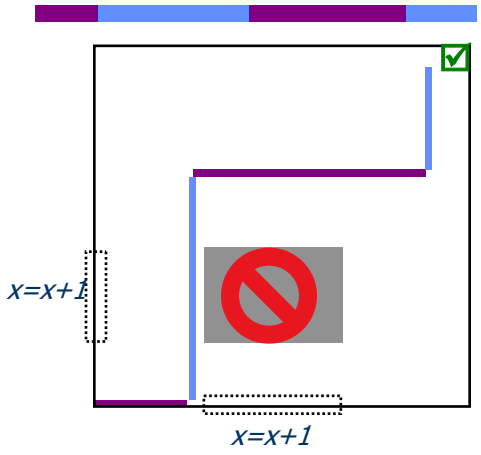
This schedule can corrupt the value of the shared variable x, causing the program to execute incorrectly.

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***

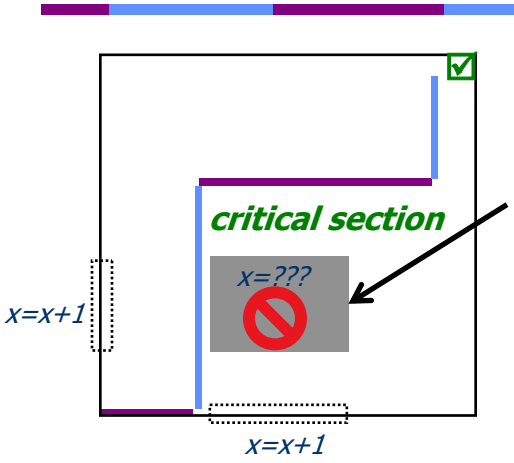
This is not a game



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.

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".

Looking Ahead

- Mechanisms to protect against race conditions
- Project 3 due in two Fridays