

Java Concurrency

Shan-Hung Wu & DataLab
CS, NTHU

Starting a New Thread

```
public class HelloRunnable implements Runnable {  
    @Override  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new Thread(new HelloRunnable())).start();  
    }  
}
```

or

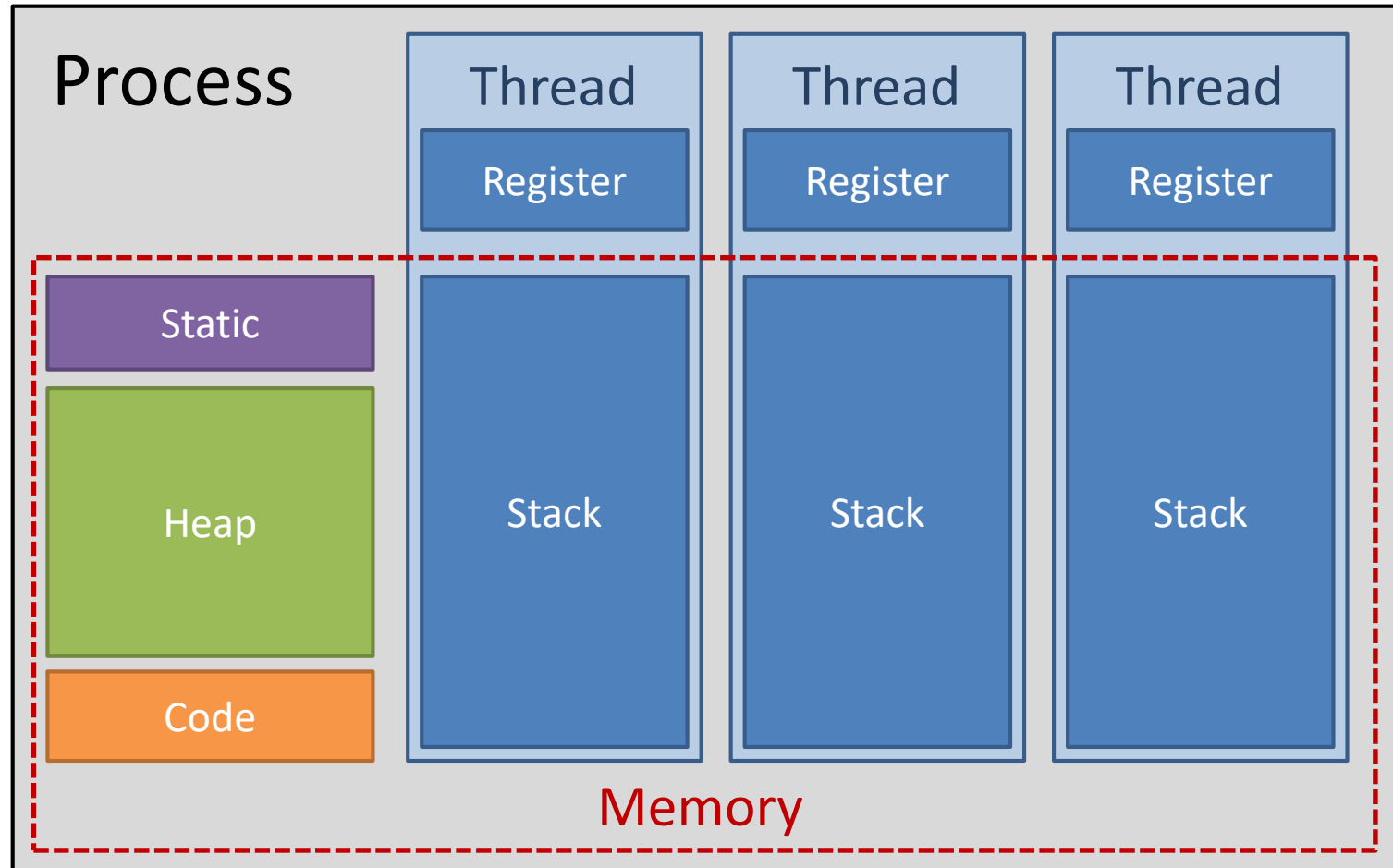
```
public class HelloThread extends Thread {  
    @Override  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new HelloThread()).start();  
    }  
}
```

What Happened?

```
public class HelloRunnable implements Runnable {  
    @Override  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new Thread(new HelloRunnable())).start();  
    }  
}
```

- A new stack is allocated for `run()`, in addition to that of `main()`
- Your CPU spends time on executing `run()` ***in parallel with*** `main()`

Memory Scheme in a Process



Multiple Stacks, Single Heap

- The heap in memory scheme?
 - Stores objects
 - *Shared by all threads*
- Can two threads access the same object? Yes
- How? Passing the same object to their constructors

```
public static void main(String args[]) {  
    Counter counter = ...;  
    (new Thread(new HelloRunnableA(counter))).start(); // thread A  
    (new Thread(new HelloRunnableB(counter))).start(); // thread B  
}
```

Concurrent Access

- Given the same object `counter`
- Suppose both threads execute in `run()`:

```
int c = counter.get();  
c++; // c--;  
counter.set(c);
```

```
class Counter {  
    private int c = 0;  
    public void set(int c) {  
        This.c = c;  
    }  
    public int get() {  
        return c;  
    }  
}
```

- Thread A's result will be lost if
 1. Thread A: Get c
 2. Thread B: Get c
 3. Thread A: Increment retrieved value; result is 1
 4. Thread B: Increment retrieved value; result is 1
 5. Thread A: Set result in c; c is now 1.
 6. Thread B: Set result in c; c is still 1.

Synchronization

```
public class SynchronizedCounter {  
    private int c = 0;  
    public synchronized void set(int c) {  
        this.c = c;  
    }  
    public synchronized int get() {  
        return c;  
    }  
}
```

- Only one thread can enter sync. block of an obj. at a time
- Problem solved?

- Same as...

```
public class SynchronizedCounter {  
    private int c = 0;  
    public void set(int c) {  
        synchronized (this) {  
            this.c = c;  
        }  
    }  
    public int get() {  
        synchronized (this) {  
            return c;  
        }  
    }  
}
```

Still Wrong!

- Two threads in `run()`:

```
... // counter is a SynchronizedCounter instance
int c = counter.get();
c++; // c--;
counter.set(c);
```

- Thread A's result will still be lost if
 1. Thread A: Get c
 2. Thread B: Get c
 3. Thread A: Increment retrieved value; result is 1
 4. Thread B: Increment retrieved value; result is 1
 5. Thread A: Set result in c; c is now 1.
 6. Thread B: Set result in c; c is still 1.

Synchronization at Right Place

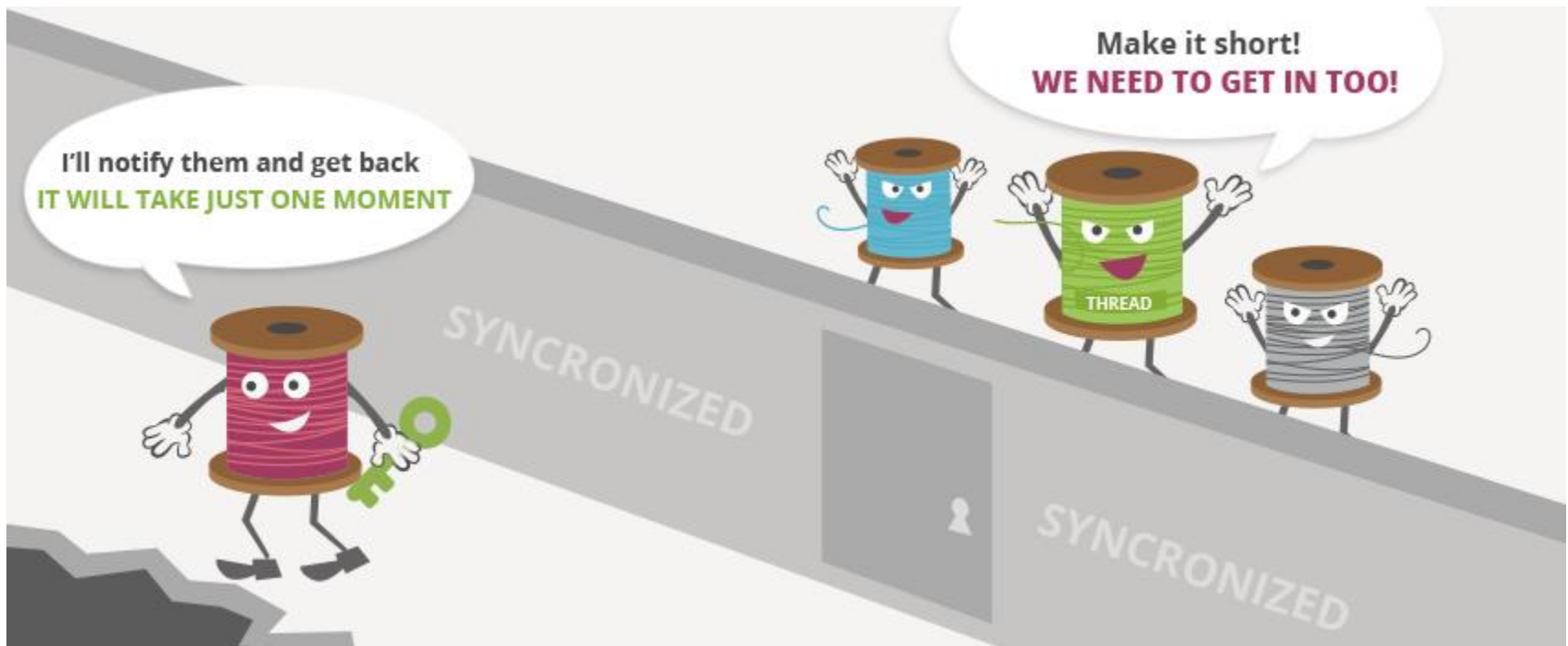
- Solution1: callers lock `counter` during the entire increment/decrement period:

```
synchronized(counter){  
    int c = counter.get();  
    c++; // or c--;  
    counter.set(c);  
}
```

- Solution2: callee provides atomic methods

```
public class SynchronizedCounter {  
    private int c = 0;  
    public void synchronized increment() {  
        c++;  
    }  
    public int get() {  
        return c;  
    }  
}
```

Synchronization



[Source](#)

Blocking and Waiting States

- Threads are **blocked** outside a critical section if someone is in
- Thread *A* in a critical section of `o` can stop and enter the **waiting** state by calling `o.wait()`
 - Gives up the lock, so some other blocking thread *B* can enter the critical section
 - If *B* calls `o.notifyAll()`, *A* competes for the lock again and resume

Wrap `wait()` in a Loop

- It's a good practice to wrap `wait()` in a loop to prevent bugs
- Queue length: 10

Threads A, B:

```
// enqueue
synchronized(queue) {
    while(queue.size() == 10) {
        queue.wait();
    }
    queue.add(...);
    queue.notifyAll();
}
```

Threads C, D:

```
// dequeue
synchronized(queue) {
    while(queue.size() == 0) {
        queue.wait();
    }
    ... = queue.remove();
    queue.notifyAll();
}
```

Assigned Reading

- [Java Concurrency Tutorial](#)