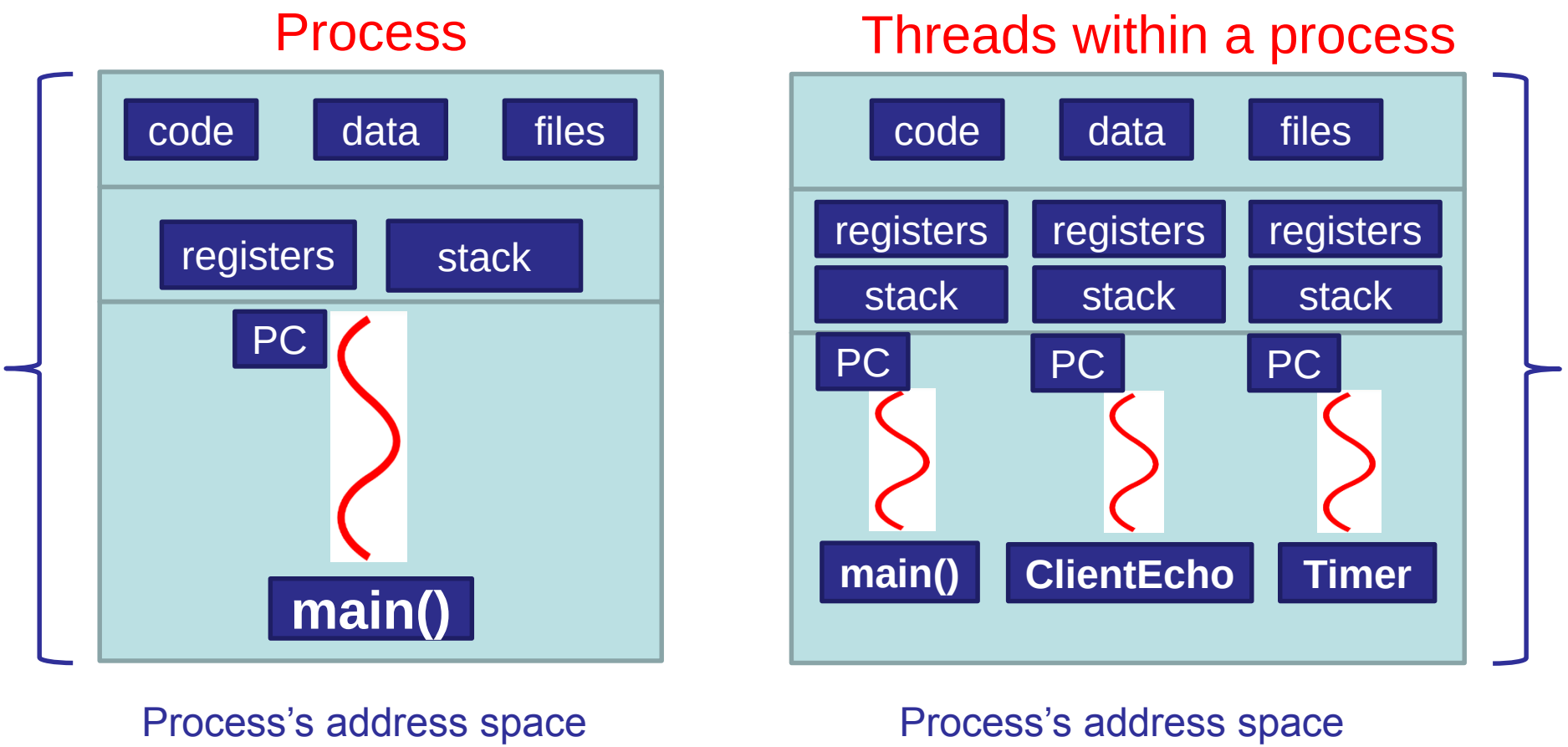
ECE 595
Threads Basics

What is a thread?



Process's address space

Process's address space

Why use threads?

- Ease of programming
  - Many interleaved tasks
  - Some tasks can ‘block’ (E.g. wait for user input)
  - Complex to program (have loop iterate over all tasks and achieve fair CPU share)
- Exploit CPU parallelism in multicores
  - OS schedules one thread per CPU

Spawn threads in Java

Method 1

```
Class ClientEchoHandler extends Thread {
    Socket client;

    ClientEchoHandler (Socket client) {
        this.client = client;
    }

    public void run() {
       BufferedReader In = new BufferedReader(
                                    new InputStreamReader
                                     (client.getInputStream()));
        PrintWriter Out = new PrintWriter
                                    (client.getOutputStream()));
        while(true) {
            String in_echo = In.readLine();
            Out.println("[echo] " + in_echo);
            System.out.println("Thread: " +
Thread.currentThread().getName());
        }
    }
}
```

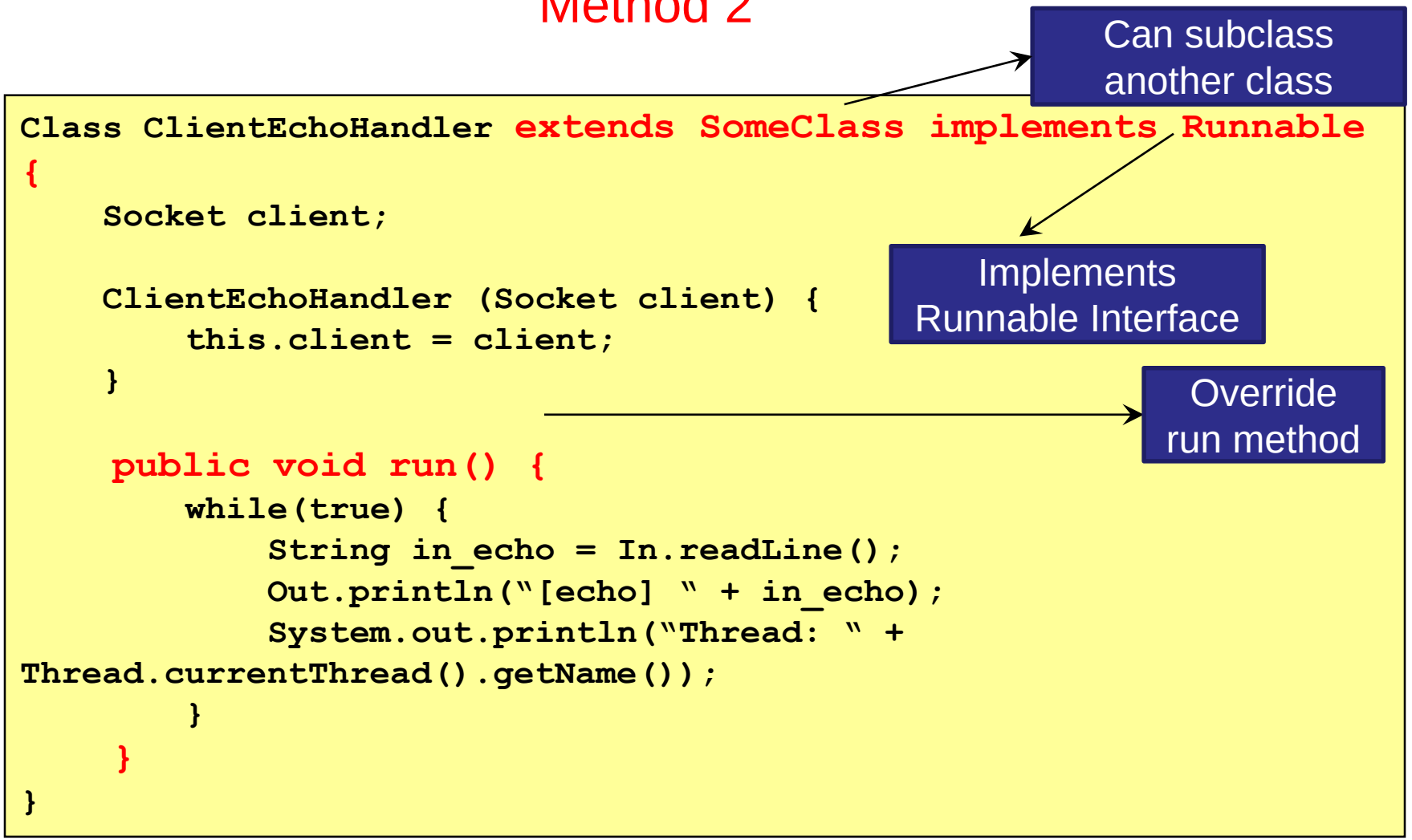
Extend base Thread class

Override run method

Print name of the thread – default Thread-#ID

Spawn threads in Java

Method 2




```
Class ClientEchoHandler extends SomeClass implements Runnable
{
    Socket client;

    ClientEchoHandler (Socket client) {
        this.client = client;
    }

    public void run() {
        while(true) {
            String in_echo = In.readLine();
            Out.println("[echo] " + in_echo);
            System.out.println("Thread: " +
Thread.currentThread().getName());
        }
    }
}
```

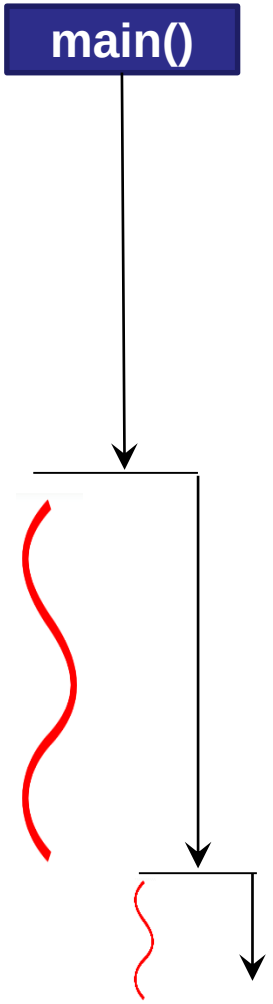
Starting Thread

```
public class EchoServer {
    public static void main(String[] args) {
        try {
            ServerSocket server = new
      ServerSocket(5000);
            Socket client = server.accept();
            ClientEchoHandler handler = new
      ClientEchoHandler(client);
     [handler.start();]
     [Thread tobj = new Thread(handler);
      tobj.start();]
    ……..
    ……..
         tobj.join();
        } catch (Exception e) {
            System.err.println("Socket
      exception:" + e);
        }
    }
}
```

Method1

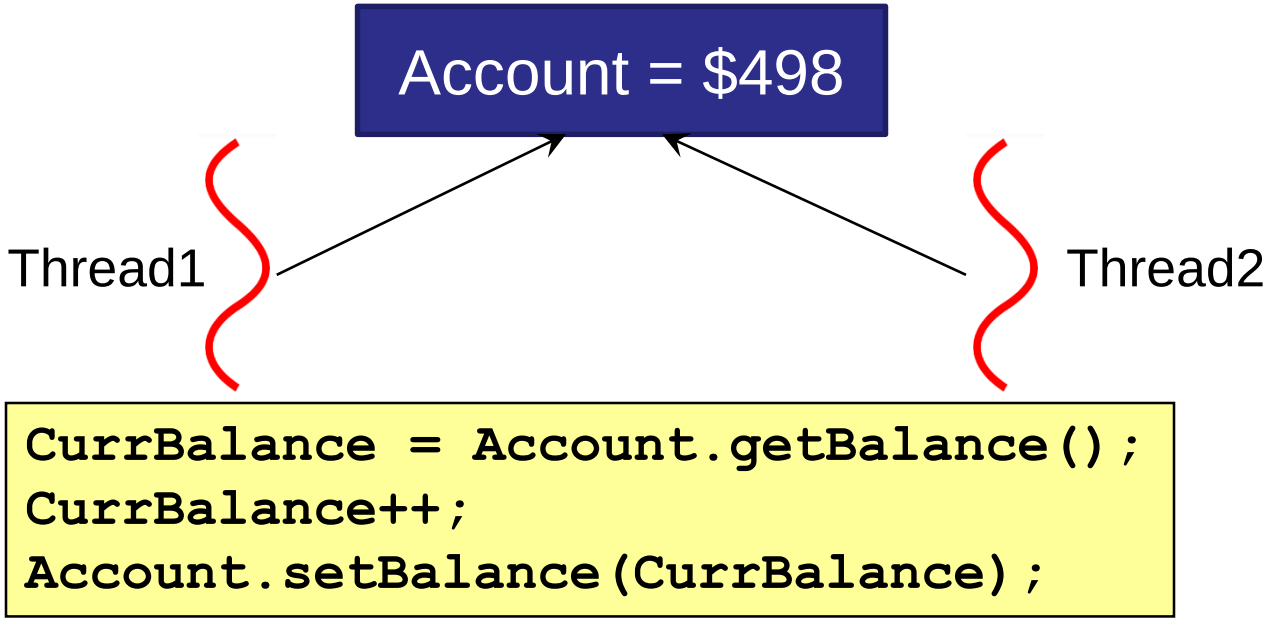
Method2

Parent waits until thread is done



Synchronization problem with threads

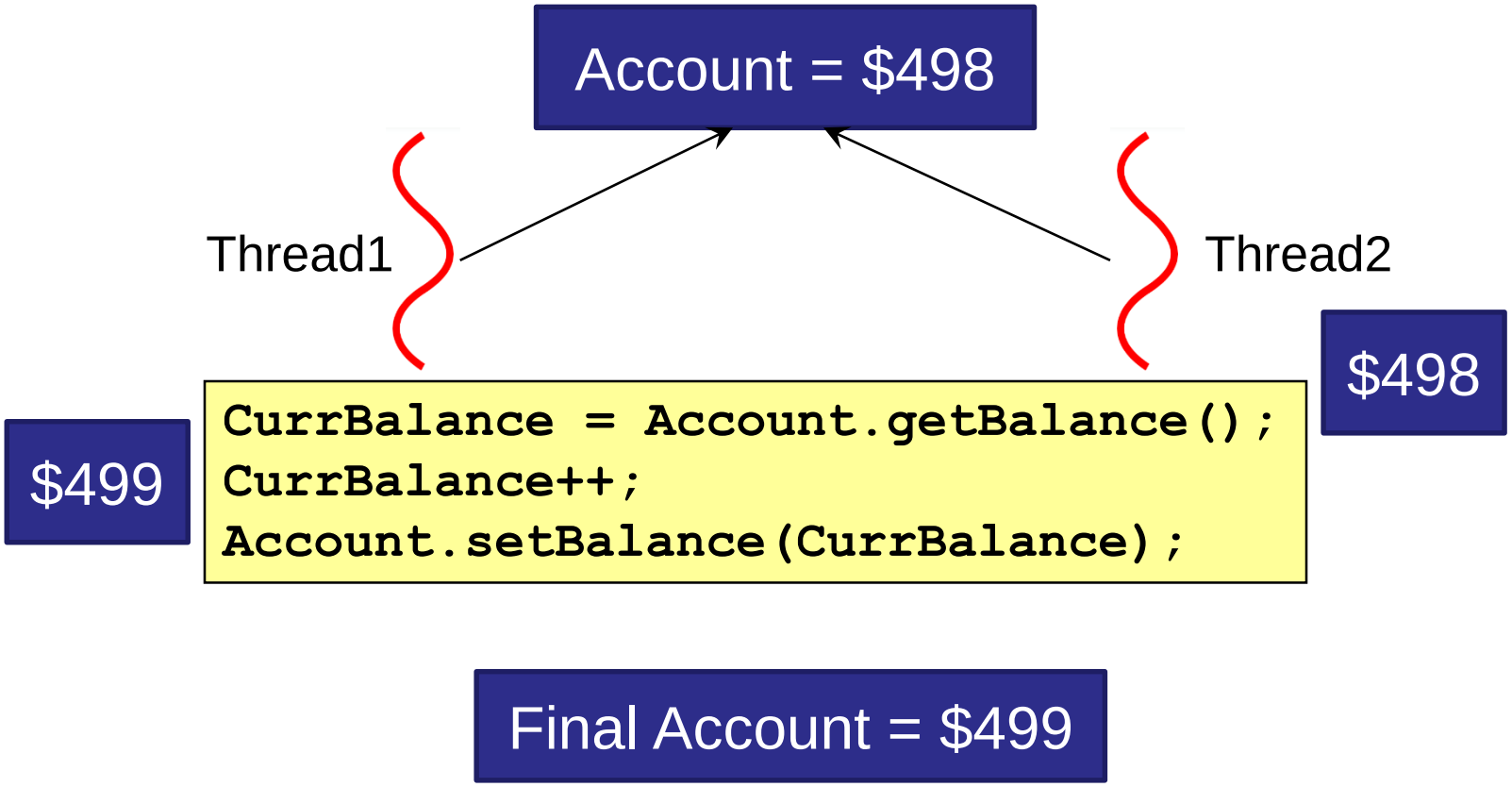
- Execution order and atomicity important



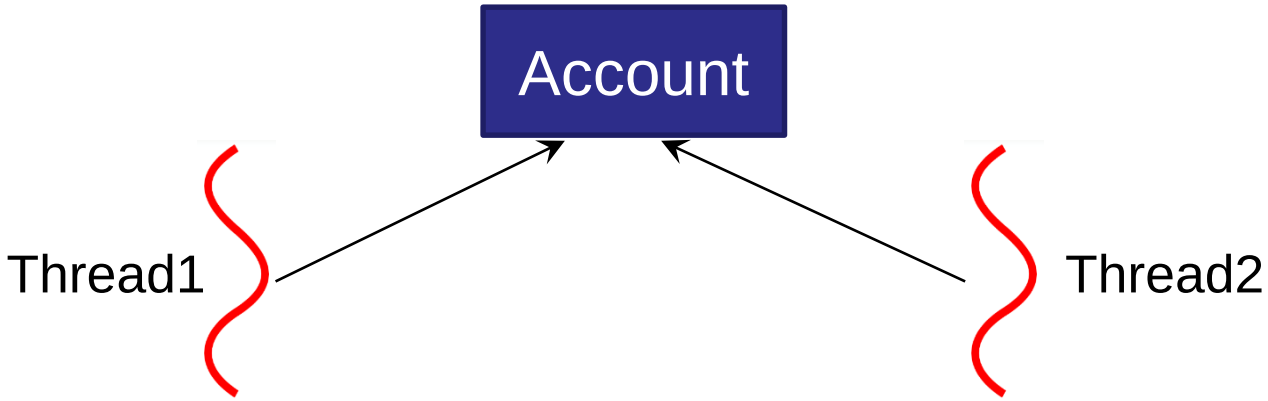
Sequential execution final output

Final Account = $500

Wrong final output – race condition



Synchronization with locks enforcing atomicity



acquire()

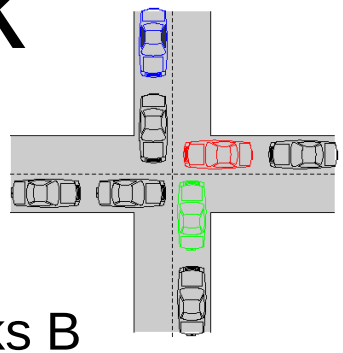
release()

```
synchronized(Account) {
    CurrBalance = Account.getBalance();
    CurrBalance++;
    Account.setBalance(CurrBalance);
}
```

critical section

synchronized method() { lock on the class}

Careless ordering with multiple locks leads to ‘deadlock’



Thread1
- Locks A
- Waits on B

Thread2
- Locks B
- Waits on A

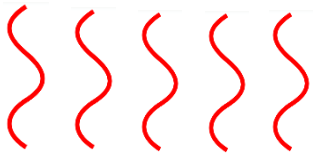
```
synchronized(A) {
   synchronized(B) {
       A.foo();
       B.bar();
   }
}
```

```
synchronized(B) {
   synchronized(A) {
       A.foo();
       B.bar();
   }
}
```

Solution : All threads maintain order while acquiring locks

Semaphores (Counting)

Allow max ‘5’ threads to execute critical section simultaneously



```
Semaphore sema = new Semaphore(5);

Runnable longRunningTask = () -> {
    boolean permit = false;
    try {
        permit = sema.tryAcquire(1, TimeUnit.SECONDS);
        if (permit) {
            System.out.println("Semaphore acquired");
            ……
        } else {
            System.out.println("Could not acquire semaphore");
        }
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    } finally {
        if (permit) {
            sema.release();
        }
    }
}
```

Other locks and synch. methods in Java – ReentrantLock(), wait(), notify(), sleep()