MET CS 622: Concurrency

Reza Rawassizadeh

Outline

- Thread Concepts
- Locks
  - Structured Lock
  - Unstructured Locks

Outline

- **Thread Concepts**
- Locks
  - Structured Lock
  - Unstructured Locks

Thread


The Coast News
New Venture Aims To Serve Lower Coast Area Weekly
'Natural Causes' Gersila Verdict
HEAT WAVE CLOSES FOREST
YOUR NEW PUBLISHERS . . .
Ferry Hearing Asked by Board
New Bus Service To Stillwater

Problem with Disk Operation

- Performing an operation inside memory is very fast.
- In contrast, accessing disk and performing I/O is slow process.
- If a process need something from disk, usually it should wait until that particular information gets available and then uses it.
- Considering slow disk access the performance of computers could be very poor.
- Operating systems resolve this challenge by introducing a **multi-thread feature**, which is called **concurrency**.

Definitions

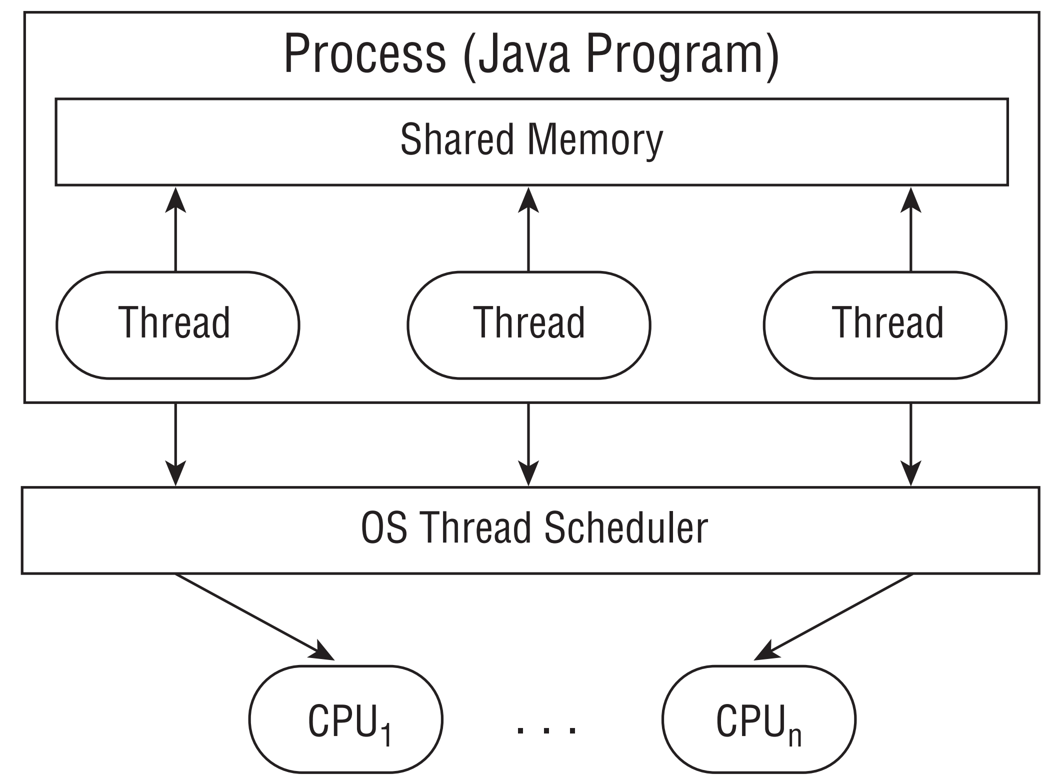
- **Thread** is the smallest unit of execution that can be scheduled by the operating system.
- A **task** is a unit of work that is performed by a thread.
- A **process** is a set of threads that execute in the same shared environment (shared memory space).
- Executing multiple threads or processes at a time is called **concurrency**.
- Most Java applications are multi-thread.

CPU and Thread

- A single core CPU can execute only one thread.
- A multi core CPU can execute more than one thread, but the number of threads could be larger than the number of CPU cores.
- Operating systems use something called **thread scheduler**, which decide what should be executed.

Thread Scheduler

- Operating systems use thread scheduler to determine which thread should be running on a CPU core.
- Thread schedulers can use a **round-robin algorithm** to determine resource to thread assignment.
- The round-robin algorithm focuses on *equal resource assignment* to each thread, in a circular fashion. To be honest, thread scheduling algorithms are far more complex than round-robin, but we need to be familiar with the concept of the round-robin algorithm.



Source: OCP_ Oracle Certified Professional Java SE 8 Programmer II Study Guide_ Exam 1Z0-809-Sybex

Thread Priority

- Java enables us to assign a priority to a thread.
- It supports three thread priorities:
  - `Thread.MIN_PRIORITY`
  - `Thread.NORM_PRIORITY`
  - `Thread.MAX_PRIORITY`

Runnable

- A functional interface is an interface that contains only **one abstract method**, but it can have any number of non-abstract methods.

- `java.lang.Runnable` is a **functional java interface** that is used to define the work that a thread should execute.

- Until now, we use “main” as an application execution entry point. However, for thread execution we need to call the `run()` method inside the `java.lang.Runnable` interface.

```
@FunctionalInterface
public interface Runnable {
    void run();
}
```

Lambda Expression

- We can also use lambda expression that implicitly implements the Runnable interface.

```
() -> System.out.println("Hello World")

() -> {int i=10; i++;}

() -> {return;}

() -> {}
```

A class that runs something inside a thread

```
public class ThreadExample implements Runnable {
   public void run(){
    // what is running inside the thread goes here
   }
}
```

Creating a Thread

There are two steps required to create a thread.

1. Defining the thread within the tasks it should perform.

2. Starting the thread.

There is no guarantee about the order of threads to be executed in Java.

Helloworld Thread

```java
package edu.bu.met622.threadtest;

public class SuperSimplePrint implements Runnable{

   @Override
   public void run() {
      for (int i=0; i<10; i++) {
         System.out.println("i is:"+i);
      }
   }
   public static void main(String[] args) {
      Thread t = new Thread(new SuperSimplePrint());
      t.start();
   }

}
```

Summary of Concurrency in Java

- Thread is a **class**
- Runnable is an **interface**
- The **Thread class implements Runnable**
- The argument passed to the thread constructor must be a Runnable.
- The threads spawned by a process run asynchronously

Now we make two threads and each one is printing a number (their task).

```java
package edu.bu.met622.threadtest;

public class FirstThread implements Runnable{

   @Override
   public void run() {
      for (int i =0; i<100 ; i++) {
         System.out.println("-from FirstThread: i:"+i);
      }
   }
}
```

```java
package edu.bu.met622.threadtest;

public class SecondThread implements Runnable{

   @Override
   public void run() {
      for (int j =0; j<100 ; j++) {
         System.out.println("-from SecondThread: j:"+j);
      }
   }

}
```

```java
package edu.bu.met622.threadtest;

import edu.bu.met622.threadtest.FirstThread;
import edu.bu.met622.threadtest.SecondThread;

public class TesttwoThread {
  public static void main(String[] args) {
    FirstThread firstT = new FirstThread() ;
    Thread a = new Thread(firstT);

    SecondThread secondT = new SecondThread();
    Thread b = new Thread(secondT);

    a.start();
    b.start();
  }
}
```

Difference between `run()` and `start()`

- The method `start()`, will starts a new thread and the JVM assigns it to a CPU core.

- The `run()`, will execute the content of a thread.

Some Common Job Interview Questions about Threads

- Explain the difference between extending the Thread class and implementing Runnable.

  A. If we need to define our own thread rules, upon which multiple tasks will rely, e.g. a **priority thread**, extending thread may be preferable.

  B. Since Java doesn't support multiple inheritance, **extending Thread** does not allow us to extend any other class, whereas implementing Runnable lets you extend another class.

  C. **Implementing Runnable** is often a better object-oriented design practice, because **it separates the task being performed from the Thread object that are performing it**.

  D. Implementing Runnable allows the class to be used by numerous Concurrency API classes.

Some Common Job Interview Questions about Threads

- Explain the difference between extending the Thread class and implementing Runnable.

   A. If we need to define our own thread rules, upon which multiple tasks will rely, e.g. a **priority thread**, extending thread may be preferable.

   B. Java does ... **tending Thread** does not a ... nplementing Runnable ...

   Many books recommend avoid extending Thread class, unless you must doing it.

   C. **Implementing Runnable** is often a better object-oriented design practice, because **it separates the task being performed from the Thread object that are performing it**.

   D. Implementing Runnable allows the class to be used by numerous Concurrency API classes.

Thread Join

experiment this code with/without join

The join method allows the calling thread to wait until another thread gets its task done.

```
package edu.bu.met622.threadtest;

//Illustrating join()
public class TestThreadJoin {
      public static void main(String[] args) throws InterruptedException {
            Thread t1 = new Thread() {
                  public void run() {
                        for (int j=0 ; j<100; j++) {
                              System.out.println("t111111");
                        }
                  }
            };
            Thread t2 = new Thread() {
                  public void run() {
                        for (int j=0 ; j<100; j++) {
                              System.out.println("t222222");
                        }
                  }
            };
            Thread t3 = new Thread() {
                  public void run() {
                        for (int k=0 ; k<100; k++) {
                              System.out.println("t333333");
                        }
                  }
            };

            // set the counter value to 10 to see its impact
            t1.start();
            t1.join();
            t2.start();
            // t2.join();
            // t3.join();
            t3.start();


            // increase the counter value to 100 to see its impact
            //t1.start();
            //t2.start();
            //t3.start();
            //t2.sleep(1000);

//          Lets comment and uncomment t2.join and see the differences.

            System.out.println("End");
      }
}
```

Thread Sleep

- As it has been explained a thread is running out of our control after it has been started.
- Sometimes it is useful if we can pause thread execution temporarily.
- `Thread.sleep(xxx)`, causes a thread to pause its execution for `xxx` milliseconds.
- `sleep()` is a **static** method in Thread.
- The `sleep()` method throws a checked exception, namely `InterruptedException.`

More example

```
public class TestThreadJoin {
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread() {
            public void run() {
                for (int j=0 ; j<100; j++) {
                    System.out.println("taaaaaa1");
                }
            }
        };
        Thread t2 = new Thread() {
            public void run() {
                for (int j=0 ; j<100; j++) {
                    System.out.println("t222222");
                }
            }
        };
        Thread t3 = new Thread() {
            public void run() {
                for (int k=0 ; k<100; k++) {
                    System.out.println("t333333");
                }
            }
        };
        t1.start();
        t1.sleep(1000);
        t2.start();
        t3.start();
//      Lets comment and uncomment t2.join and see the differences.
//      t2.join();
        System.out.println("End");
    }
}
```

Thread Sleep

experiment with/without sleep

```
package edu.bu.met622.threadtest;

public class SleepTest1 {
   public static void main(String args[]) throws InterruptedException {
      String importantInfo[] = { “msg 1", "msg 2", "msg 3”, "msg 4” };

      for (int i = 0; i < importantInfo.length; i++) {
         // Pause for 1 seconds
         Thread.sleep(2000);
         // Print a message
         System.out.println(importantInfo[i]);
      }
   }
}
```

Outline

- Thread Concepts
- **Locks**
  - Structured Lock
  - Unstructured Locks

Race Condition Problem

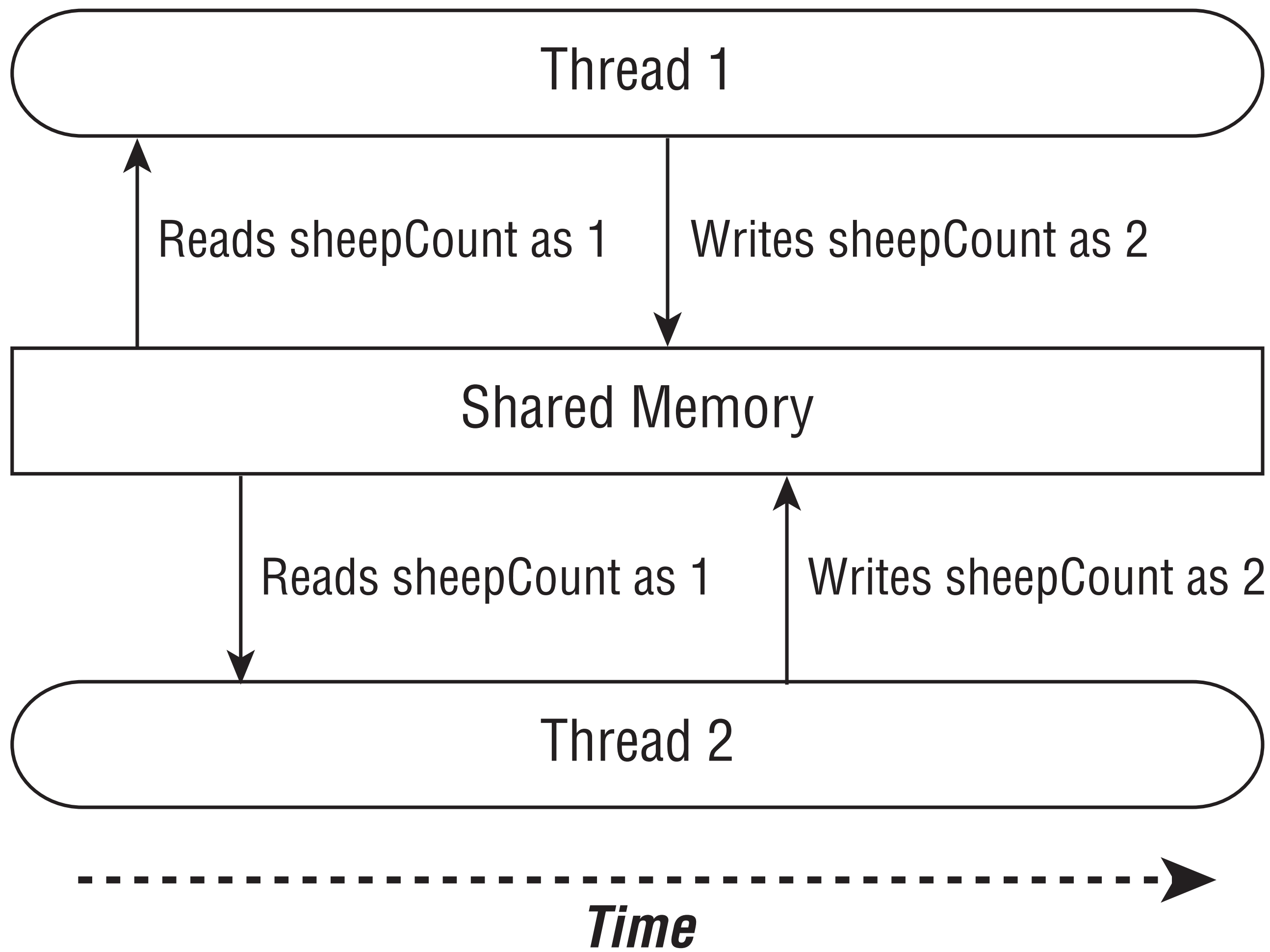
- Suppose we have a method that count an object inside our limited memory, e.g. `increment(val)` as follows:

```
void increment(val){
   val = val+1
   }
```

- Now imagine we have two threads, `T1` and `T2`, and both are calling `increment(val)`

- They are not aware about each other, but they do the same thing. When `T1` calls `increment(val)`and increases `val` from zero to one, then `T2` calls val to increments it to 1, but mistakingly it will increased to 2, because `T1` did increase to 1.

Another Example
Sheep Counting



Source: *OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide: Exam 1Z0-809*. John Wiley & Sons.

Outline

- Thread Concepts
- Locks
  - **Structured Lock**
  - Unstructured Locks

Structured Lock

- To avoid such a confusion we can use the `synchronize` keyword, which imposes a **lock** on the `increment` while a thread is working with it. Or we can use atomic classes, which will be described later.

```
synchronized void increment(val){
    val = val+1
}
```

- A content of the synchronized method is called **mutually exclusive**, because one thread at a time can work with it. As soon as the thread is starting to work with this method, there will be **lock** on this method and no other method can use it. When a thread is done, the lock will be released and other threads can use it.

Locking an Object with synchronized

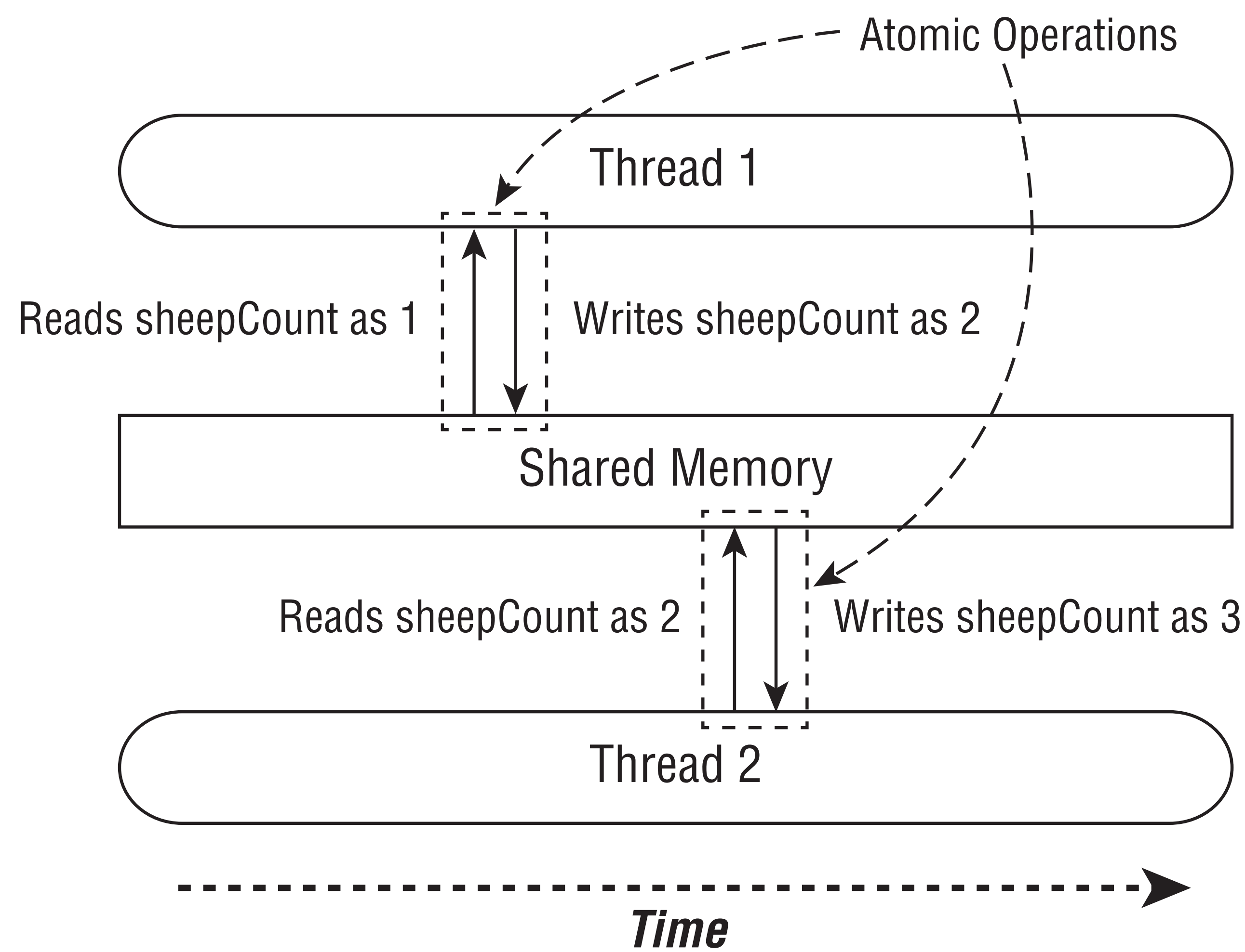
- Some time locking the entire method might negatively impact the other tasks of the method. We can also lock a an object in a block of code.

```
BufferVal buf = new BufferVal{
   …
   synchronized(buf){
     val = val+1;
     …
    }
   …
   }
```

Atomic

- Atomic is a property of operation, which **disables any other thread interference** on that operation.

It is more restricted than Synchronized lock. There are limited number of Atomic classes and Methods and we need to keep them in mind.



Atomic classes and Methods

- Atomic Classes: `AtomicBoolean, AtomicInteger, AtomicIntegerArray, AtomicLong, AtomicLongArray, AtomicReference, AtomicReferenceArray`

- Atomic Methods: `get(), set(), getAndSet(), incrementAndGet(), getAndIncrement(), decrementAndGet(), getAndDecrement()`

AtomicInteger Example

```
private AtomicInteger sheepCount = new AtomicInteger(0);

private void incrementAndReport() {
    System.out.print(sheepCount.incrementAndGet()+" ");
}
```

What are the disadvantage of using Synchronized for thread?

Synchronization Disadvantage

- The objective of multi-threat programming is doing task multiple tasks in parallel. By enforcing a lock we disable the multi tasking features, which might increase response time.
- Synchronization protects data integrity, but its costs will be on response time.
- Being able to identify the **performance bottleneck of a system**, especially in multithread environment is very valuable capability.

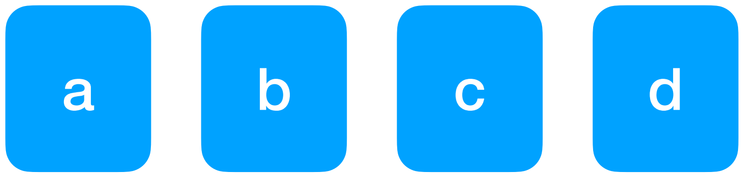
Outline

- Thread Concepts
- Locks
  - Structured Lock
  - **Unstructured Locks**

Unstructured Lock

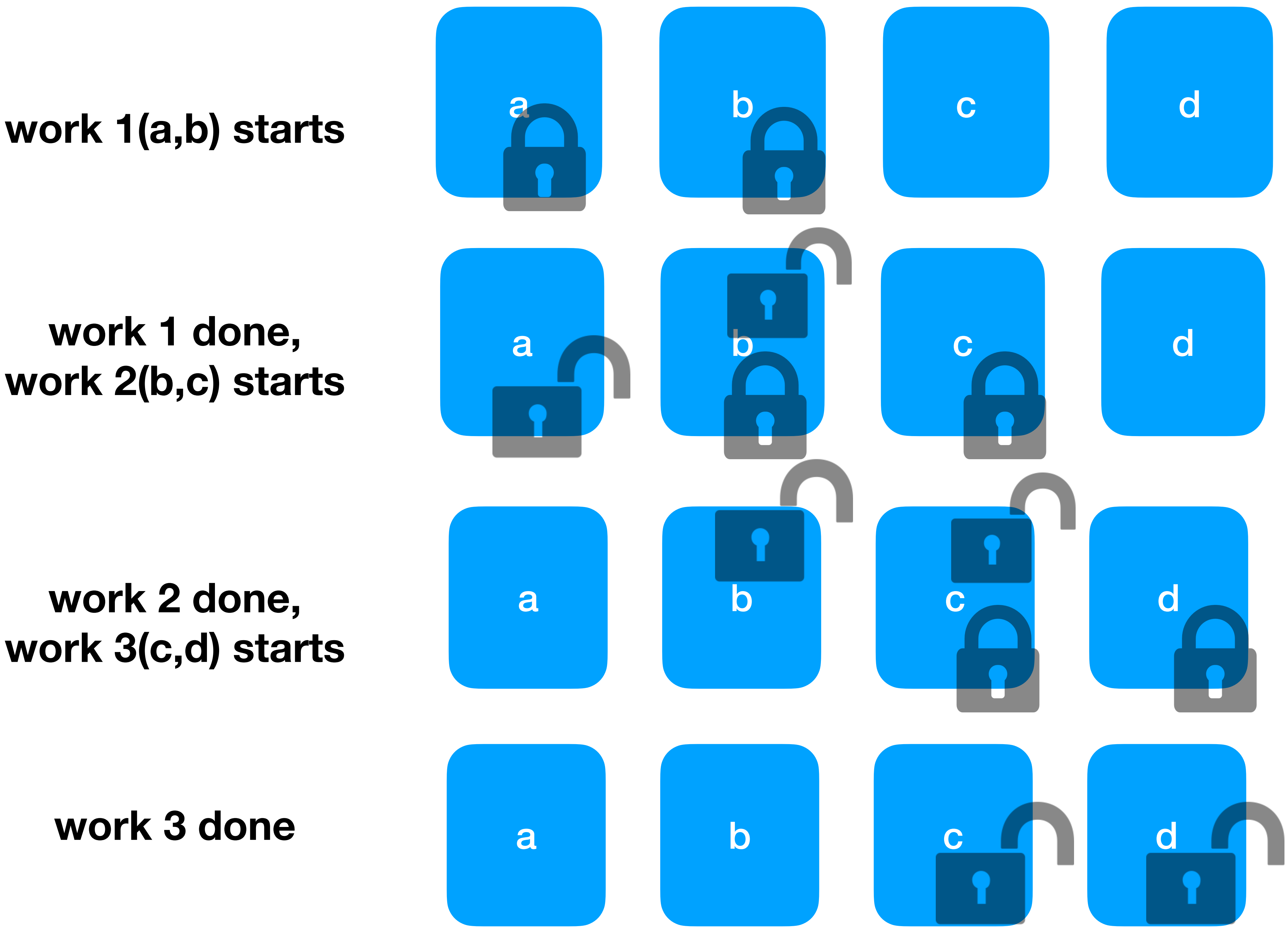
Assume we have a four objects (a,b,c,d) and we have three works that should to be executed in a sequence, we need to do with these objects (w1, w2, w3).

Each of these works require two objects: w1(a,b), w2(b,c) , w3(c,d)

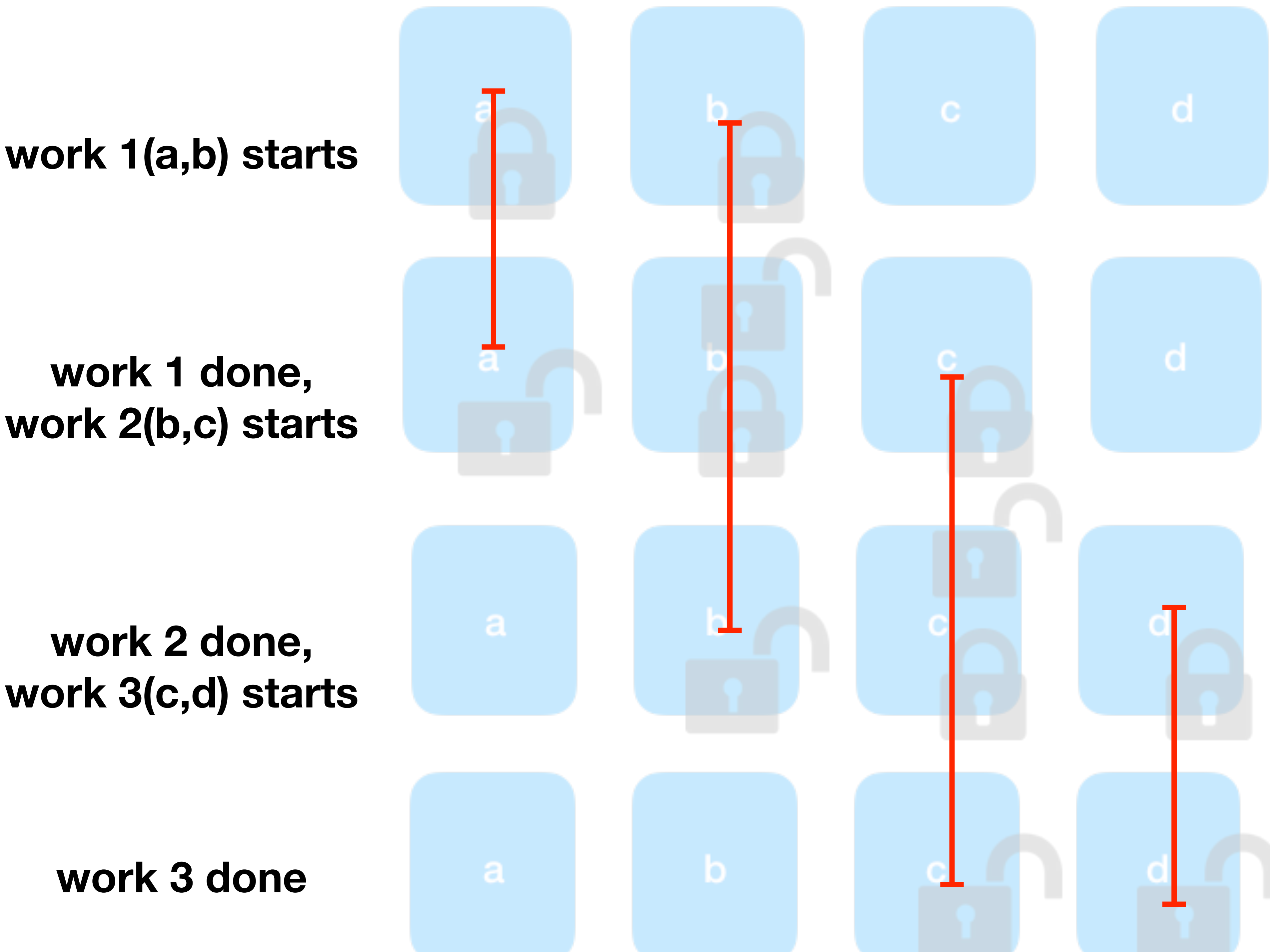


work 1(a,b)
work 2(b,c)
work 3(c,d)

Unstructured Lock



Locks are nested inside each other



How to Resolve Nested Locks

- There are two solutions designed to handle nested locks, using a **ReentrantLock** (**Interface Lock** or **Try Lock**) and using **Read/Write** Lock.
- ReentrantLock will hold the lock until the thread is done.
- Read/Write Lock <u>separates read and write operation</u> and implement different locks for them

ReentrantLock

- **`Lock`** implementations provide more extensive locking operations than can be obtained using **`synchronized`** methods and statements. They allow more flexible structuring.

```
Lock lock = new ...;
    lock.lock();
    try{
          // use the resource protected by this lock
    }
    finally {
      lock.unlock();
    }
```

Try Lock Example (1/3)

```
package edu.bu.met622.threadtest.trylock;

public class MyThread extends Thread{

    PrintThread printT;
    MyThread(String name, PrintThread pt){
        super(name);
        this.printT = pt;

    }
    @Override
    public void run() {
        System.out.printf(
            "%s starts printing a document\n”,
Thread.currentThread().getName());
        printT.print();
    }

}
```

Try Lock Example (2/3)

```
package edu.bu.met622.threadtest.trylock;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class PrintThread {
      private final Lock queueLock = new ReentrantLock();

      public void print() {
         queueLock.lock();

         try {
            Long duration = (long) (Math.random() * 1000);
             // Play with this duration to show lock changes
            System.out.println(Thread.currentThread().getName() +
                                   " Time Taken " + (duration / 1000) + " secs.");
            Thread.sleep(duration);
         } catch (InterruptedException e) {
            e.printStackTrace();
         } finally {
            System.out.printf(
               "%s printed the document successfully.\n",
Thread.currentThread().getName());
            queueLock.unlock();
         }
      }
}
```

Try Lock Example (3/3)

```java
package edu.bu.met622.threadtest.trylock;

public class TestThread {
    public static void main(String args[]) {
        PrintThread PD = new PrintThread();

        MyThread t1 = new MyThread("Thread - 1 ", PD);
        MyThread t2 = new MyThread("Thread - 2 ", PD);
        MyThread t3 = new MyThread("Thread - 3 ", PD);
        MyThread t4 = new MyThread("Thread - 4 ", PD);

        t1.start();
        t2.start();
        t3.start();
        t4.start();
    }
}
```

Source: https://www.tutorialspoint.com/java_concurrency/concurrency_lock.htm

Read/Write Lock

- It is a solution to mitigate nested locks.
- For example, assume we have an application that both read and write a set of objects. The read threads occurs more often than write, but some writes threads are also there.
- We can define two locks:
  - **Read Lock:** It will be unlocked if no thread is reading or writing.
  - **Write Lock:** It will be unlocked if no thread is writing and no thread has requested write access.
- ***More than one thread can hold a Read Lock, but only one thread can holds the Write Lock***

```java
public class ReadWriteLock{

  private int readers       = 0;
  private int writers       = 0;
  private int writeRequests = 0;

  public synchronized void lockRead() throws InterruptedException{
    while(writers > 0 || writeRequests > 0){
      wait();
    }
    readers++;
  }

  public synchronized void unlockRead(){
    readers--;
    notifyAll();
  }

  public synchronized void lockWrite() throws InterruptedException{
    writeRequests++;

    while(readers > 0 || writers > 0){
      wait();
    }
    writeRequests--;
    writers++;
  }

  public synchronized void unlockWrite() throws InterruptedException{
    writers--;
    notifyAll();
  }
}
```

source: http://tutorials.jenkov.com/java-concurrency/read-write-locks.html

Semantic Errors in Concurrency (Liveness problems)

- DeadLock
- LiveLock
- Starvation

DeadLock

DeadLock

Thread 1:

```
synch(A) {
   synch(B) {
    }
 }
```

Thread 2:

```
synch(B) {
   synch(A) {
    }
 }
```

Thread 1 starts and acquires a lock on A, then Thread 2 starts and acquires a lock on B. Now thread 1 is willing to acquire the lock on B, but thread 2 is holding it. Also thread B is waiting for thread A to release the lock, but A is waiting for B.

LiveLock

Threads are not blocked, but they are in a mode, which do not make any progress toward finishing their task.

Thread 1:

```
do {
    synch(x) {
       x=x+1
     }
  } while (x<2)
```

Thread 2:

```
do {
    synch(x) {
       x=x-1
     }
  } while (x>-2)
```

LiveLock

Thread 1:

```
do {
    synch(x) {
        x=x+1
    }
} while (x<3)
```

Thread 2:

```
do {
    synch(x) {
        x=x-1
    }
} while (x>-3)
```

x=0
T1: x=1
T2: x=0
T1: x=1
T1: x=2
T2: x=1
T2: x=0
T2: x=-1
T1: x=0
...

Starvation

- Assume we have many threads, e.g. 500 threads (T1, … T500) and each are designed to do a small task, which might be similar or dissimilar tasks.
- We start all of them together, and they continuously get called and execute their tasks.
- Since we have too many threads, some threads might never get called, e.g. T1,….T498, called but the last two ones never get called, T499 and T500.
- Then we can say T499 and T500 are starved.

Starvation

- Assume we have many threads, e.g. 500 threads (T1, … T500) and each are designed to do a small task, which might be similar or dissimilar tasks.

- We s … sly get calle…

This usually happen in read/write locks. We do lots of read threads and write threads are getting starved.

- Since … might never get called, e.g. T1,….T498, called but the last two ones never get called, T499 and T500.

- Then we can say T400 and T500 are starved.

ExecutorService & Thread Pools

- Previous Java thread classes are complex to make them scalable. Because thread creation and management will be handled inside other codes.

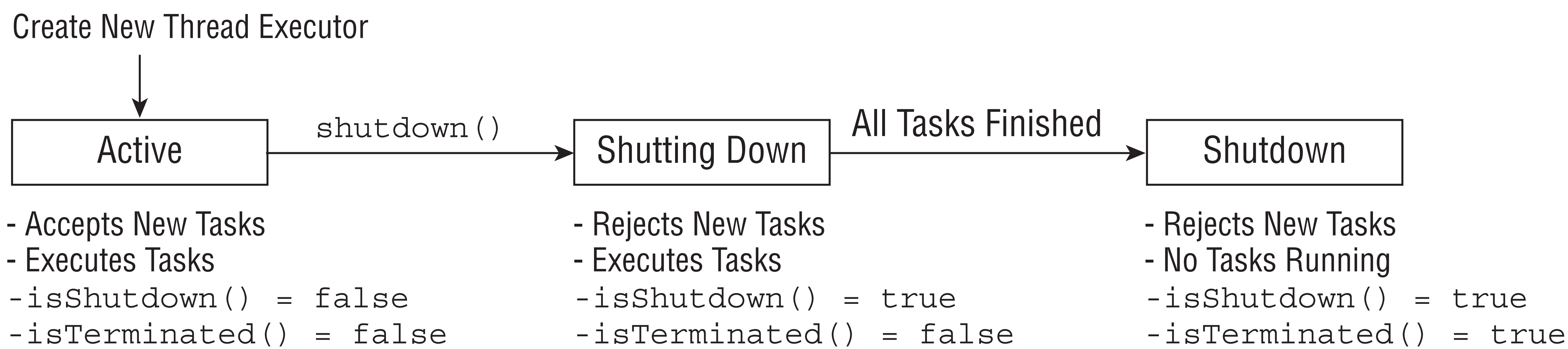
- Java tries to mitigate this challenge by introducing the **Executor Framework**. It helps us in thread creation, management, and task assignment to the thread pool:

1. **`Executor`:** It is an interface, with an **`execute()`** method to launch a task specified by a **`Runnable`** object.

2. **`ExecutorService`:** It implements the **`Executor`** interface and adds functionality to manage the life cycle of the **task**.
   It also includes **`submit()`** method which is a newer version of **`execute()`**. It returns **`Callable`** object. Callable objects are similar to Runnable but they have output as well.

3. **`ScheduledExecutorService`:** A sub-interface of `ExecutorService`, but it has the functionality to the execution of tasks as well.

Executor LifeCycle



Execute vs Submit

- The `execute()` takes a Runnable object and completes the task. It **doesn't return anything about the success or failure of task completion** (fire-and-forget).

- The `submit()` is a newer method which **returns the future object, which is the result of thread execution**. This object can assist us identifying the result of thread execution.

`void execute(Runnable command):` Executes a Runnable task at some point in the future.

`<T> Future<T> submit(Callable<T> task):` Executes a Callable task at some point in the future and returns a Future object which is representing the pending results of the task.

Future Object

The Future class includes methods that are useful in determining the state of a task.

`boolean isDone():` Returns true if the task was compiled.

`boolean isCancelled():` Returns true if the task is cancelled.

`boolean cancel():` Attempt to cancel the execution of a task.

`V get():` Retrieves the result of task.

`V get(long timeout, TimeUnit unit):` Retrieves the result of a task, waiting the specified amount of time. If the result is not ready by the time the timeout is reached, a checked TimeoutException will be thrown.

Executor Example (1)

```
package edu.bu.met622.threadtest.executor2eg;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class ExecutorExample {
      public static void main(String[] args) {
            System.out.println("Inside : " + Thread.currentThread().getName());

            // Callable (returns future object)
//          System.out.println("Creating a Runnable...");
            Runnable runnable = () -> {
//              System.out.println("Inside : " + Thread.currentThread().getName());
            };

            Callable callable = Executors.callable(runnable);
            System.out.println("---Callable output:"+ callable.getClass().getSimpleName());
            System.out.println("---Callable output2:"+ callable.getClass().getTypeName());

            // Executor (does not return future object)
//          System.out.println("Creating Executor Service...");
            ExecutorService executorService = Executors.newSingleThreadExecutor();

//          System.out.println("Submit the task specified by the runnable to the executor service.");
            Future a = executorService.submit(runnable);
            System.out.println(" ---> a is done: "+a.isDone());
            executorService.shutdown();
        }
}
```

Executor Example (2-1)

```
package edu.bu.met622.threadtest.executoreg;

class Counter {
    private int c = 0;
    public void increment() {
        System.out.println("Thread Id: " +
                           Thread.currentThread().getId());
        c = c + 1;
    }
    public int getvalue() {return c;}
}
```

Executor Example (2-2)

```
package edu.bu.met622.threadtest.executoreg;

class First implements Runnable {
	private int i;
	private Counter ctr;
	public First(int i, Counter c) {
	 this.i = i;
	    this.ctr = c;
	}
	public void changeCounter(Counter cntr) {
	 if (i > 0) {
			synchronized(cntr) {cntr.increment();}
	 }
	}
	public void run() {
	   changeCounter(ctr);
	}
}
```

Executor Example (2-3)

```
package edu.bu.met622.threadtest.executoreg;

import java.util.concurrent.*;
public class ThreadPoolExecuteService
{
      public static void main(String[] args) throws InterruptedException
      {
       Counter sharedcounter = new Counter();
       ExecutorService ex = Executors.newCachedThreadPool();
       ex.execute(new First(1, sharedcounter));
       ex.execute(new First(2, sharedcounter));
       ex.execute(new First(3, sharedcounter));

       ex.shutdown();

//       ex.awaitTermination(50, TimeUnit.MILLISECONDS);
      while (!ex.isTerminated())

       System.out.println("Final count = " + sharedcounter.getvalue());
      }
}
```

References

https://www.coursera.org/instructor/vivek-sarkar

https://www.callicoder.com/java-executor-service-and-thread-pool-tutorial/