Introduction to Software Design & Concurrency

Lecture Notes

Software Design (SE324)

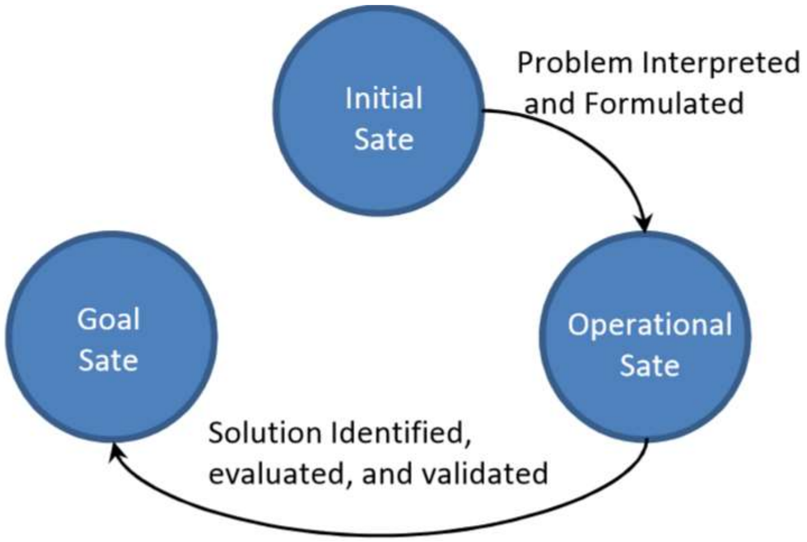
(Prepared by Dr. Khaldoon Al-Zoubi)

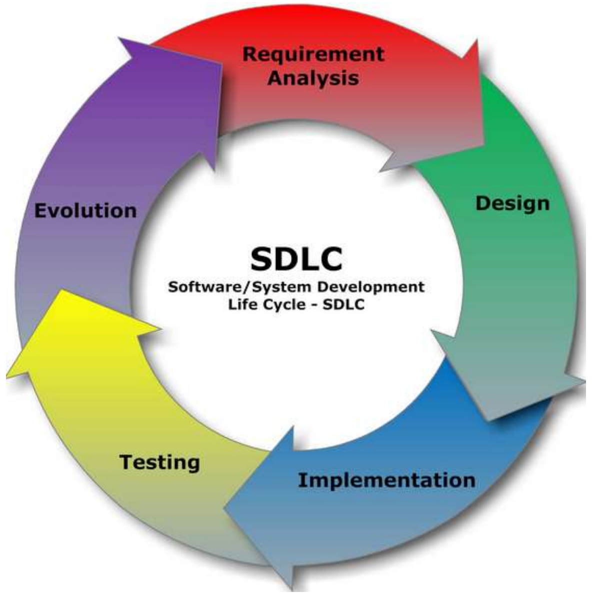
Engineering Problem Solving States

- Initial State
  - Problems are formulated and interpreted
  - Problems can be classified as:
    - *Well-defined Problems (*Clear and understood)
    - *Ill-defined problems (ambiguous with undefined goals)*
    - *Wicked Problem*
      - *No one "correct" solution*
      - Tradeoffs in design
      - one that could be clearly defined only by solving it, or by solving part of it
- Operational State
  - Engineers generate alternative solutions
  - Engineers select a possible solution from the alternatives
  - Type of thinking
    - Convergent thinking (seek single solutions -> math)
    - Divergent thinking (seek alternative solutions)
- Goal State
  - Final solution found = the end of the process

– *How these states match software process?*

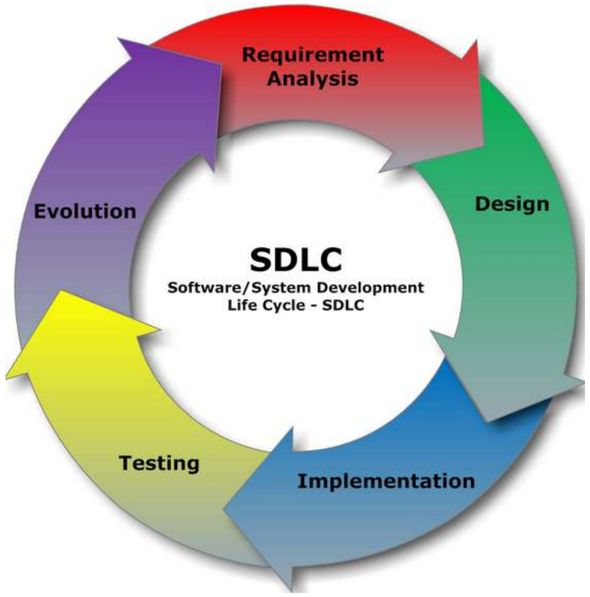
– *What is Engineering?*


Initial Sate
Problem Interpreted and Formulated
Operational Sate
Solution Identified, evaluated, and validated
Goal Sate



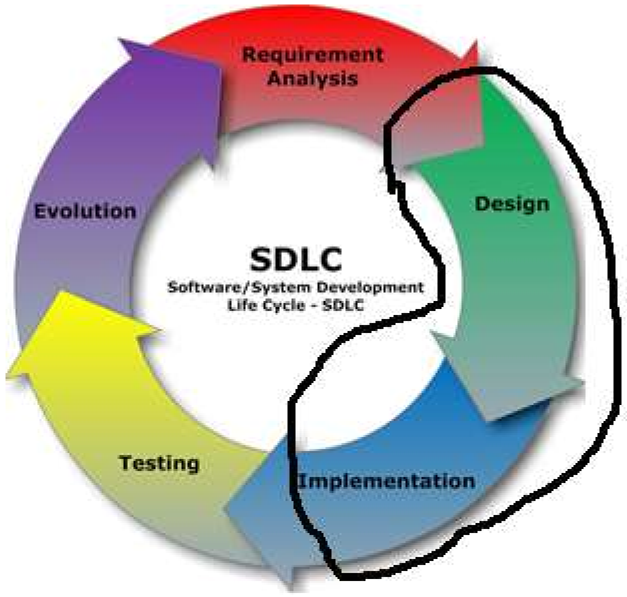
Software Engineering Process

- What is a process in general?
  - a series of actions or steps taken in order to achieve a particular end.
- What is a software Development Process?
  - A structured set of activities required to develop a software system
    - Activities to solve a problem in software
  - All software processes involve same activities
- Software Process Model
  - The way you advance from a stage to another
    - Waterfall, Incremental, Agile, etc



Software Design

- Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.
  - SDLC → to produce software product
- Software Design → How to build software Systems
  - The **process** for <u>turning</u> a specification into operational software with acceptable quality
  - The **process** of <u>implementing</u> software solutions to one or more sets of problems with acceptable quality
  - The **process** of <u>How to build software</u> to meet its intended functional and non-functional requirements
    - Software Modelling → representations of the real thing
    - Programming → to build the real thing



Why Software Design matters?

- Because it always takes more time to write something from scratch then changing something that is already there.
  - With good design the time you spend to change (and test) something is also minimized
    - Maintainable software
      - Can implement new requirements
      - Can change existing implementation
        - » Because requirement keep changing
    - Unit Test Programs also need to be changed
      - They also need to have good design
- What is a bad code?

Why Software Design matters?

- What is a bad designed code?
  - If you change something
    - you break something else that is not related to the change
  - if you change something
    - you need to change something else
- ***How Software designers solve problems?***
  - Dividing the problem
  - Isolating parts (each part has one goal/responsibility)
  - Reducing dependencies between different related elements

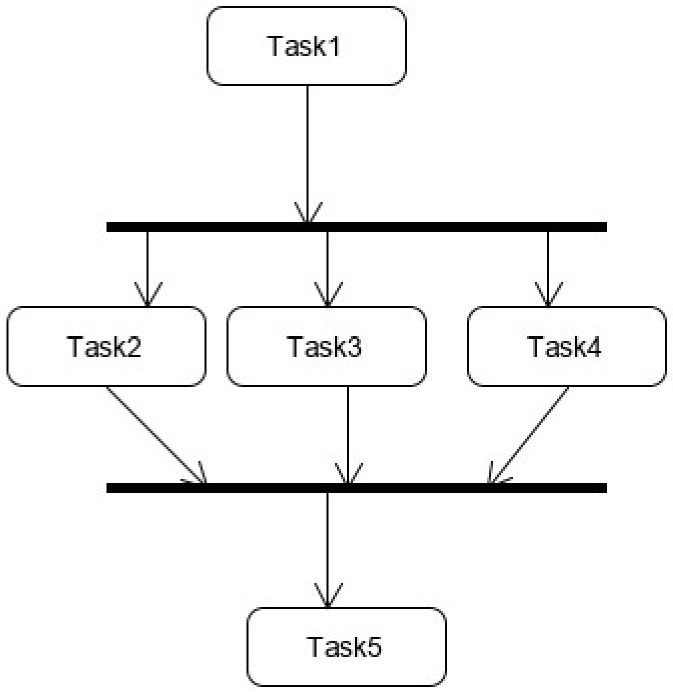
Cohesion & Coupling

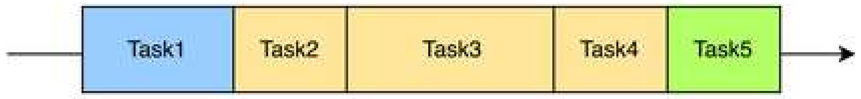
- **Cohesion** refers to the degree to which the elements of an entity belong together
- **Coupling** refers to degree of interdependence between software entities.
- **Good Design =**
  - **High Cohesion**
    - High cohesion is when you have a class that does a well defined job. Low cohesion is when a class does a lot of jobs that don't have much in common.
  - **Low/Loosely Coupling**
    - The goal of a loose coupling architecture is to reduce the risk that a change made within one element will create unanticipated changes within other elements.
    - Examples
      - physical connections via mediator
      - Interfaces, Encapsulation, etc.
      - Asynchronous message communication

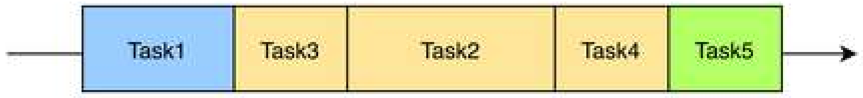
Concurrency

Concurrency

- <u>In theory</u>: concurrency refers to the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome.
  - If you have tasks having inputs and outputs, and you want to **<u>schedule</u>** them so that they produce correct results, you are solving a concurrency problem.
- The figure shows a data flow with input and output dependencies.
  - Here tasks 2, 3, 4 need to run after 1, and before 5.
    - There is no specific order between them
    - so we have multiple alternatives for running it sequentially. Showing only two of them below

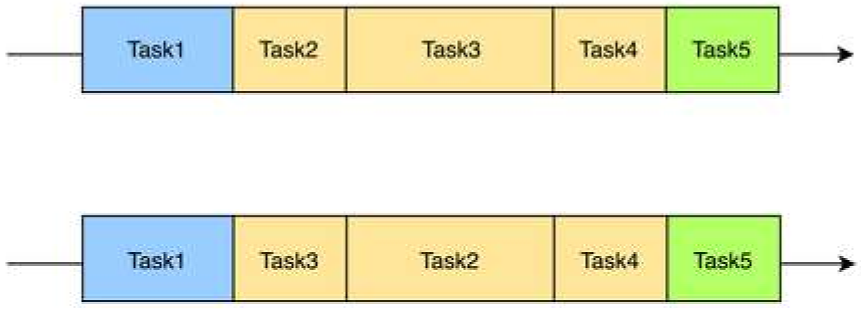


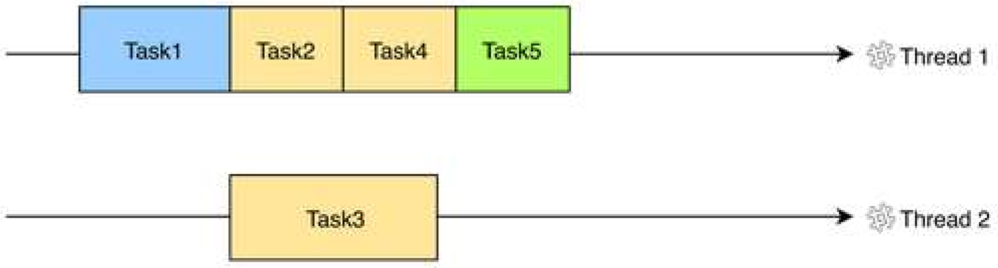




Concurrency

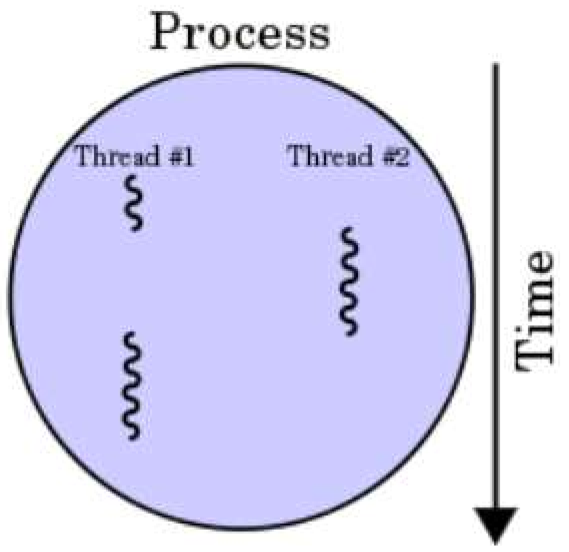
- Alternatively, these tasks can run in <u>parallel</u>,
  - e.g. on another processor core, another processor, or an entirely separate computer.
  - Parallel = tasks run at the same time on multi-core processors
    - Special case of concurrency
- On these diagrams, thread means a computation carried out on dedicated processor core
  - as they are not necessarily parallel.
  - How else could you run a multithreaded web server with dedicated threads for hundreds of connections?
    - Each connection handled by a single thread
- running concurrent tasks in <u>parallel</u> can reduce the overall computation time → speed it up
- How about concurrency on single processor core?
  - Reduce latency responsiveness





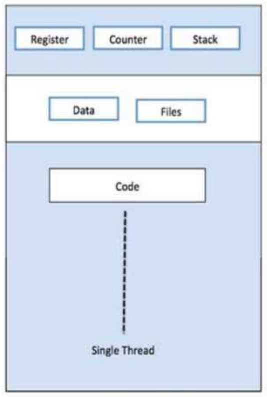
Process & Threads

- A process has a self-contained execution environment.
  - A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.
  - Communicate with each other via operating system
    - *Inter Process Communication* (IPC)
- A thread is a single execution sequence that represents a separately schedulable task
  - Single execution sequence: familiar programming model
  - Separately schedulable: OS can run or suspend a thread at any time

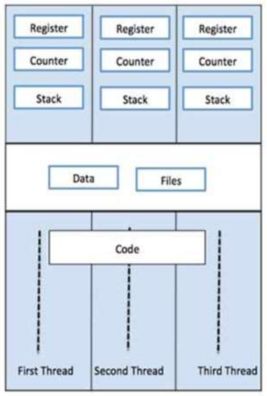


Process & Threads

- What is stack (within memory context)?
  - special region of your computer's memory that stores temporary variables created by each function
    - "LIFO" (last in, first out) data structure
    - Local variables are allocated and freed automatically
      - Managed by CPU
    - stack variables only exist while the function that created them, is running
    - Each thread gets a stack (not shared with others)
- What is heap?
  - The heap is a region of your computer's memory that is not managed automatically for you
    - To allocate memory on the heap, you must use [malloc() in C, new() in C++, etc.]
    - Once you have allocated memory on the heap, you are responsible for deleting them [free() in C, delete in C++, etc.]
      - If you fail to do this, your program will have what is known as a **memory leak**.



Single Process P with single thread

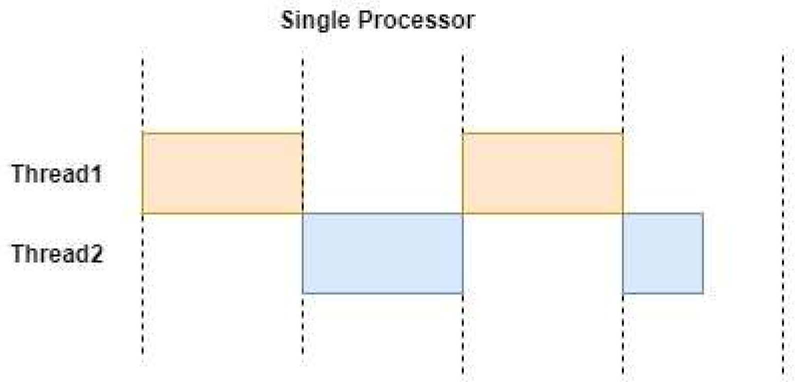


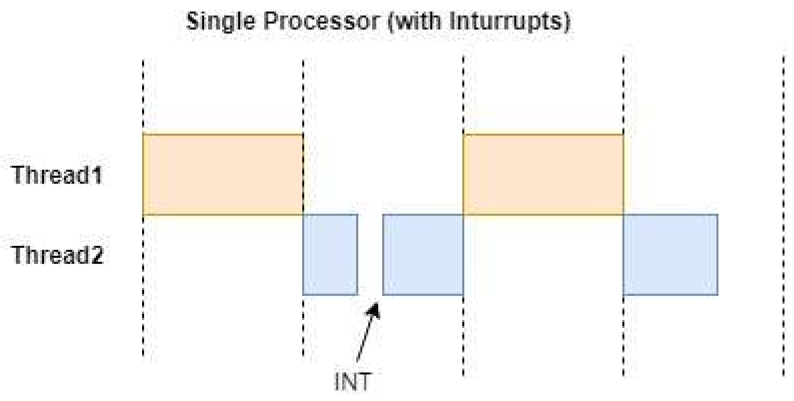
Single Process P with three threads

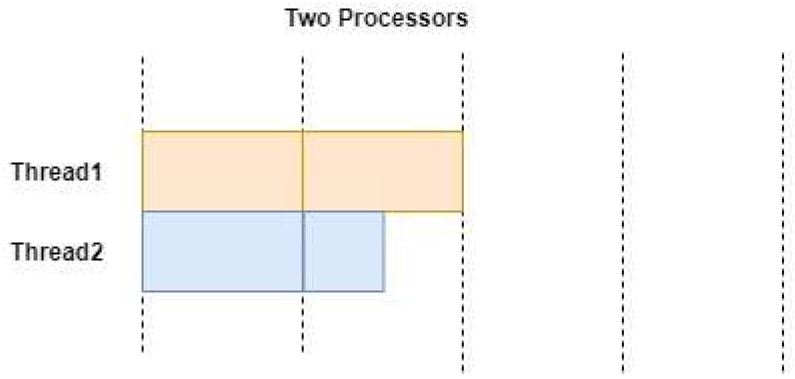
Thread Context switching

- OS calls routine *Switch(oldThread, nextThread)* , which performs:
    1. Save all registers in *old Thread*'s *context block*
    2. Save Program Counter (PC) in *old Thread*'s *context block*
    3. load next thread values into the registers from the context block of the next thread
    4. Restore next thread saved PC. At this point, next thread is executing (Context routine is done)
- Typical Reasons:
    - Time slice expired
        - ***OS Time slice*** *is the period of time for which a thread is allowed to run in a multitasking system*
    - Timer or I/O interrupt
    - Higher priority thread needs to run

Possible Executions







What could go wrong in concurrency?

- When threads concurrently read/write shared memory, program behavior is undefined
  - Two threads write to the same variable; which one should win?
- Thread schedule is non-deterministic
  - Behavior changes when re-run program
- Multi-word operations are not atomic
- To solve any of these, you need **synchronization**.
  - You want program behavior to be a specific function of input
    - not of the sequence of who went first.
  - You want the behavior to be deterministic
    - not to vary from run to run

Synchronization

Synchronization refers to the coordination of simultaneous threads or processes to complete a task with correct runtime order and no unexpected race conditions

Race condition: output of a concurrent program depends on the order of operations between threads

Mutual exclusion: only one thread does a particular thing at a time

- **Critical section:** piece of code that only one thread can execute at once

Lock (mutex = mutual exclusion): prevent someone from doing something

- Lock before entering critical section, before accessing shared data
- Unlock when leaving, after done accessing shared data
- Wait if locked (all synchronization involves waiting!)
- only one thread

Semaphores

- Use a semaphore when you (thread) want to sleep till some other thread tells you to wake up
- Allow one or more threads to enter

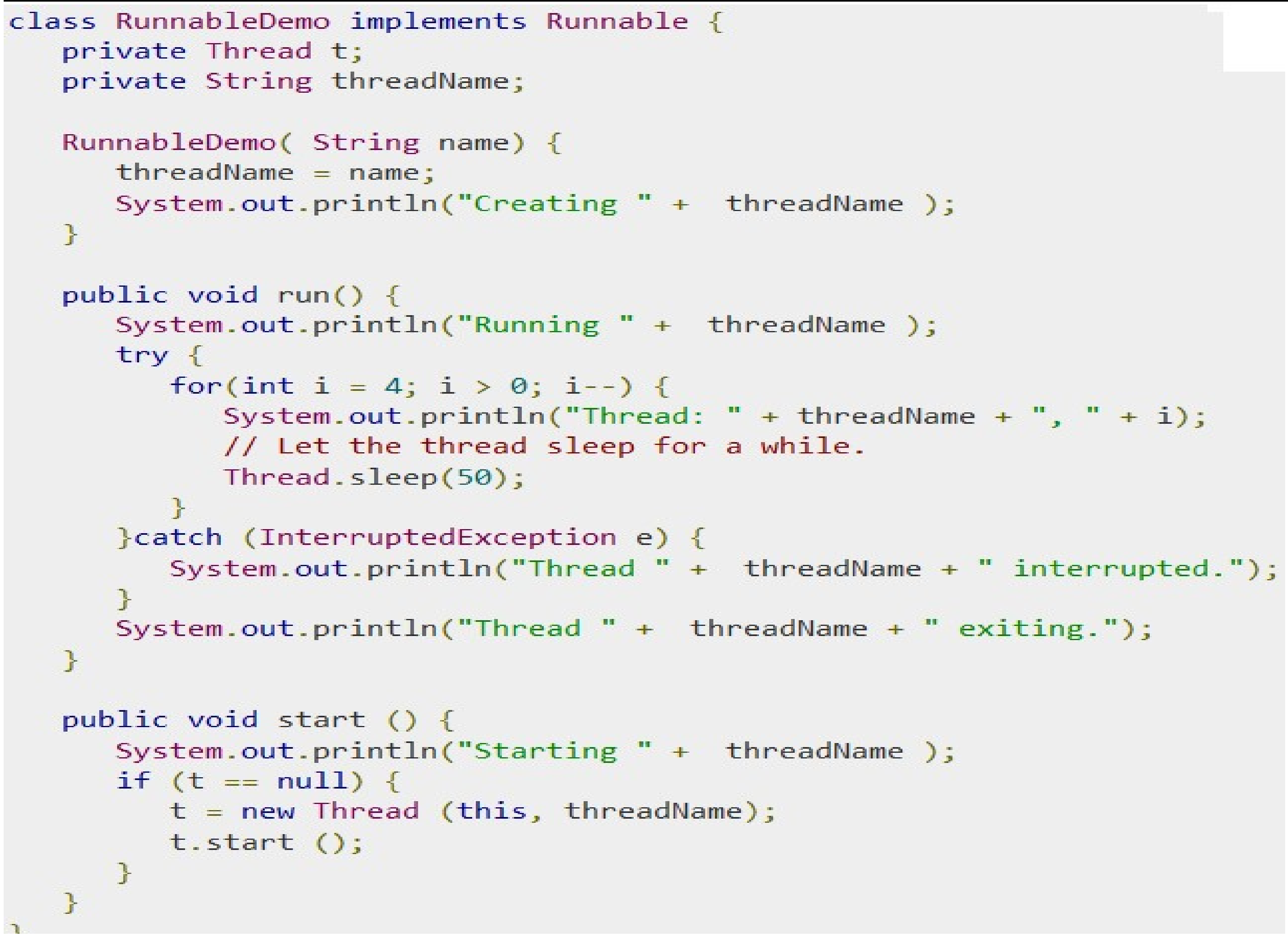
Synchronization

- A program with lots of concurrent threads can still have poor performance on a multiprocessor
  - Threads creation overhead & Context switching
  - Unnecessary synchronization (no need for protection)
    - Big critical sections
- Starvation: thread waits indefinitely to enter the critical section
- Deadlock: circular waiting for resources

Java Threads Creation

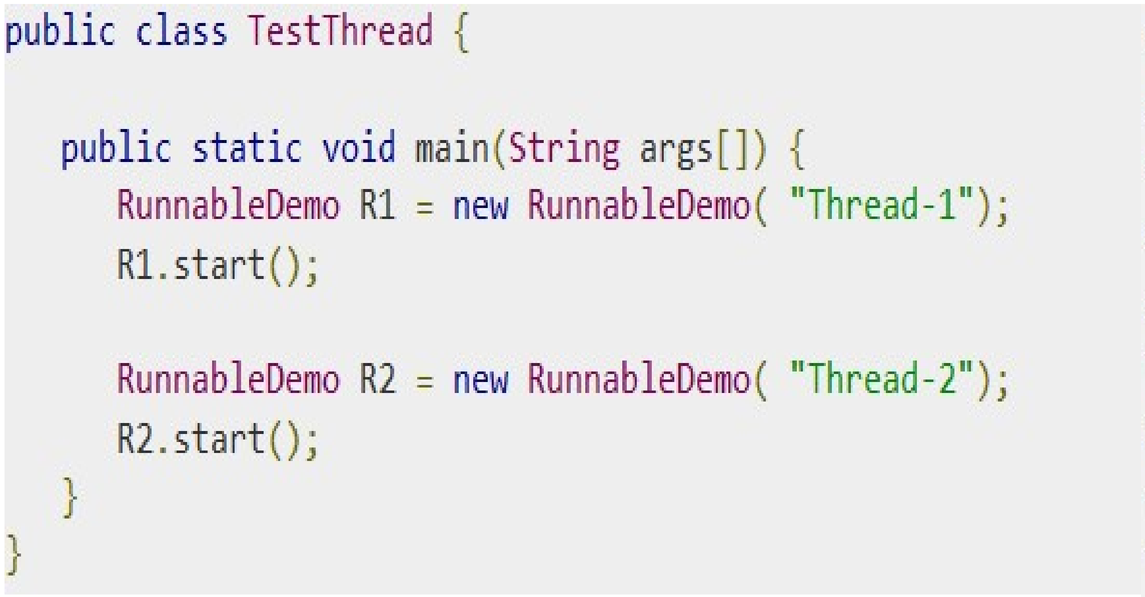
- **Option 1:**
  - If your class is intended to be executed as a thread then you can achieve this by implementing a **Runnable** interface
  - you need to implement a **run()** method provided by a Runnable interface.
    - run() provides an entry point for the thread and you will put your complete business logic inside this method
  - you will instantiate a **Thread** object using the following constructor : *Thread(Runnable threadObj, String threadName);*
  - Once a Thread object is created, you can start it by calling start() method, which executes a call to run( ) method

Java Threads
(runnable interface Example)

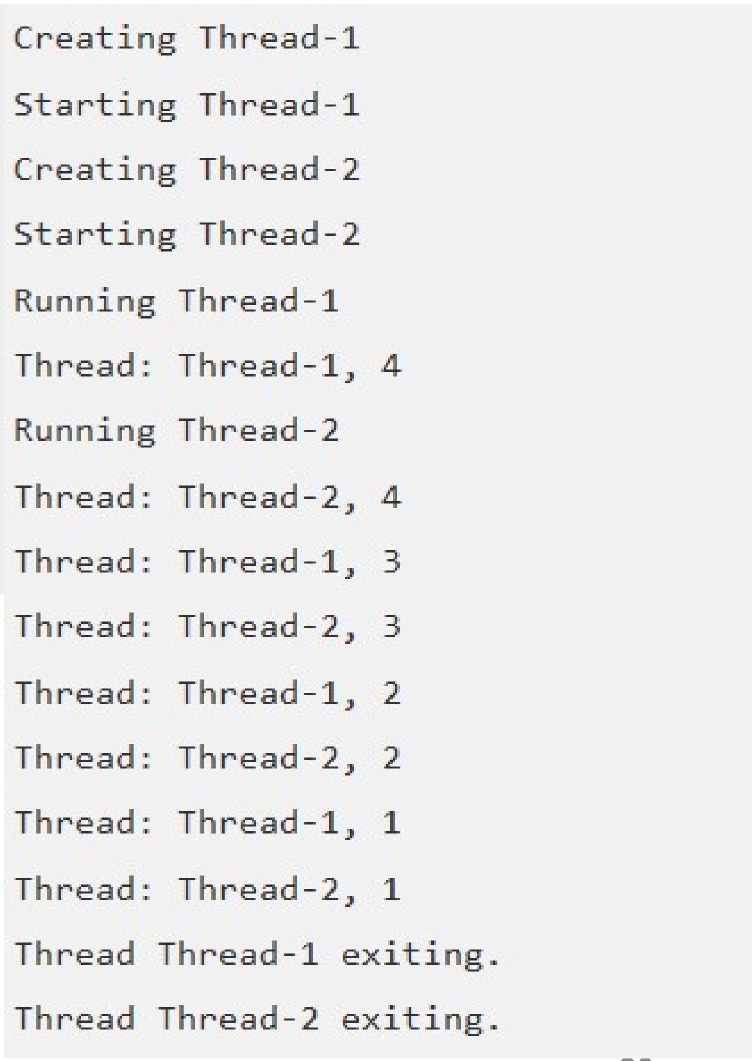


```
class RunnableDemo implements Runnable {
   private Thread t;
   private String threadName;

   RunnableDemo( String name) {
      threadName = name;
      System.out.println("Creating " +  threadName );
   }

   public void run() {
      System.out.println("Running " +  threadName );
      try {
         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", " + i);
            // Let the thread sleep for a while.
            Thread.sleep(50);
         }
      }catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + " interrupted.");
      }
      System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start () {
      System.out.println("Starting " +  threadName );
      if (t == null) {
         t = new Thread (this, threadName);
         t.start ();
      }
   }
}
```

Java Threads (runnable interface Example Cont.)



```java
public class TestThread {

   public static void main(String args[]) {
      RunnableDemo R1 = new RunnableDemo( "Thread-1");
      R1.start();

      RunnableDemo R2 = new RunnableDemo( "Thread-2");
      R2.start();
   }
}
```



```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Java Threads Creation

- **Option 2:**
  - The second way to create a thread is to create a new class that extends **Thread** class
  - you need to implement a **run()** method available in Thread class.
    - run() provides an entry point for the thread and you will put your complete business logic inside this method
  - Once a Thread object is created, you can start it by calling **start()** method, which executes a call to **run()** method

Java Threads
(Thread Class)

```java
class ThreadDemo
{
   public static void main (String [] args)
   {
      MyThread mt = new MyThread ();
      mt.start ();
      for (int i = 0; i < 50; i++)
           System.out.println ("i = " + i + ", i * i = " + i * i);
   }
}
class MyThread extends Thread
{
   public void run ()
   {
      for (int count = 1, row = 1; row < 20; row++, count++)
      {
           for (int i = 0; i < count; i++)
                System.out.print ('*');
           System.out.print ('\n');
      }
   }
}
```

Synchronization in Java

- Any object can serve as a lock
  - Separate object: `Object myLock = new Object();`
  - Current instance: the this object
- Enclose lines of code in a *synchronized* block

```
synchronized(myLock) {
        // code here
}
```

- More than one thread could try to execute this code, but one acquires the lock and the others “block” or wait until the first thread releases the lock
- Common situation: all the code in a method is a critical section
  - add synchronized keyword to method signature.
  - E.g. `public synchronized void update(…) {`

-
```
synchronized(this) {
        // code here
}
```

Synchronization in Java

- Any object can serve as a lock
  - Separate object: **static** `Object myLock = new Object();`
  - Current instance: the this object
- Enclose lines of code in a *synchronized* block

```
synchronized(myLock) {
     // code here
}
```

- More than one thread could try to execute this code, but one acquires the lock and the others “block” or wait until the first thread releases the lock
- Common situation: all the code in a method is a critical section
  - add synchronized keyword to method signature.
  - E.g. `public synchronized static void update(…) {`
- `synchronized (DemoClass.class) {`

```
         //other thread safe code
}
```

Example

```
public class BankAccount {

    private double balance;

    public void deposit(double amount) {
        balance += amount;
    }

    public void withdraw(double amount) {
        balance -= amount;
    }
}
```

Is this a thread safe?

Example

```
public class BankAccount {

    private double balance;

    public synchronized void deposit(double amount) {
        balance += amount;
    }

    public void withdraw(double amount) {
        balance -= amount;
    }
}
```

Is this a thread safe?

Example

```
public class BankAccount {

    private double balance;

    public synchronized void deposit(double amount) {
        balance += amount;
    }

    public synchronized void withdraw(double amount) {
        balance -= amount;
    }
}
```

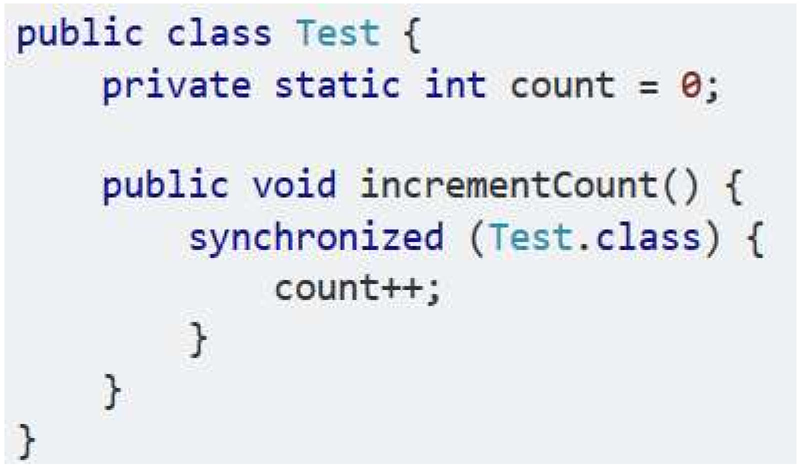
Is this a thread safe?

Example

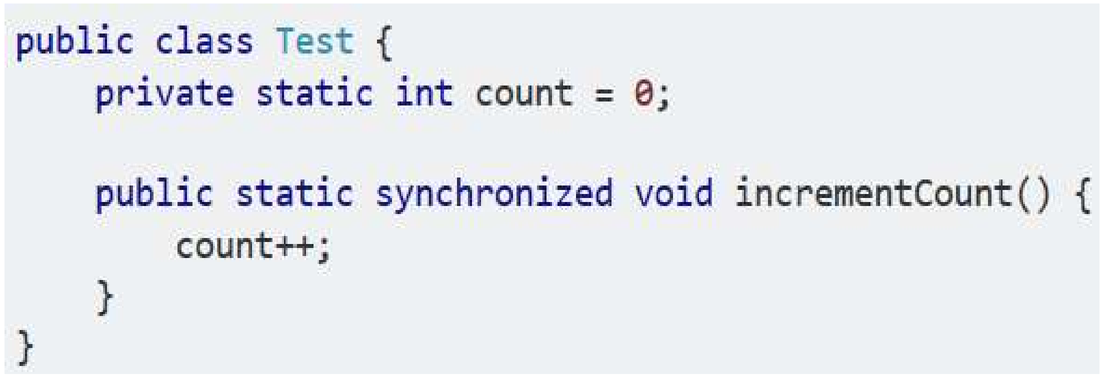
```
public void method1() {
    do something ...
    synchronized(this) {
        a ++;
    }
    ................
}


public void method2() {
    do something ...
    synchronized(this) {
        b ++;
    }
    ................
}
```

```
  class Test {
        private Object lockA = new Object();
        private Object lockB = new Object();

public void method1() {
    do something ...
    synchronized(lockA) {
        a ++;
    }
    ................
}


public void method2() {
    do something ...
    synchronized(lockB) {
        b ++;
    }
    ................
 }
```

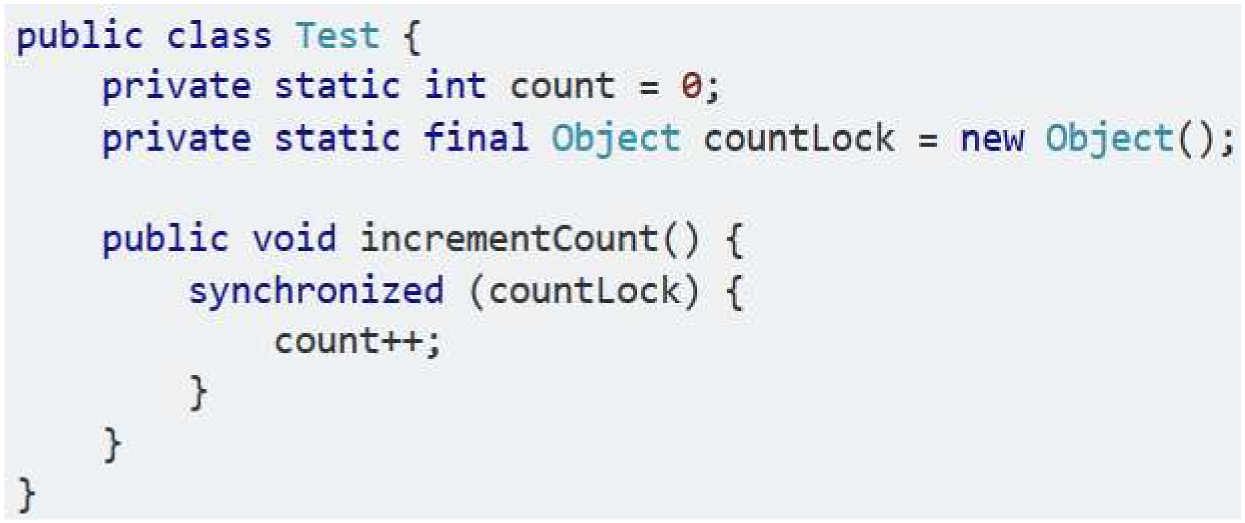
Example



```java
public class Test {
    private static int count = 0;

    public void incrementCount() {
        synchronized (Test.class) {
            count++;
        }
    }
}
```

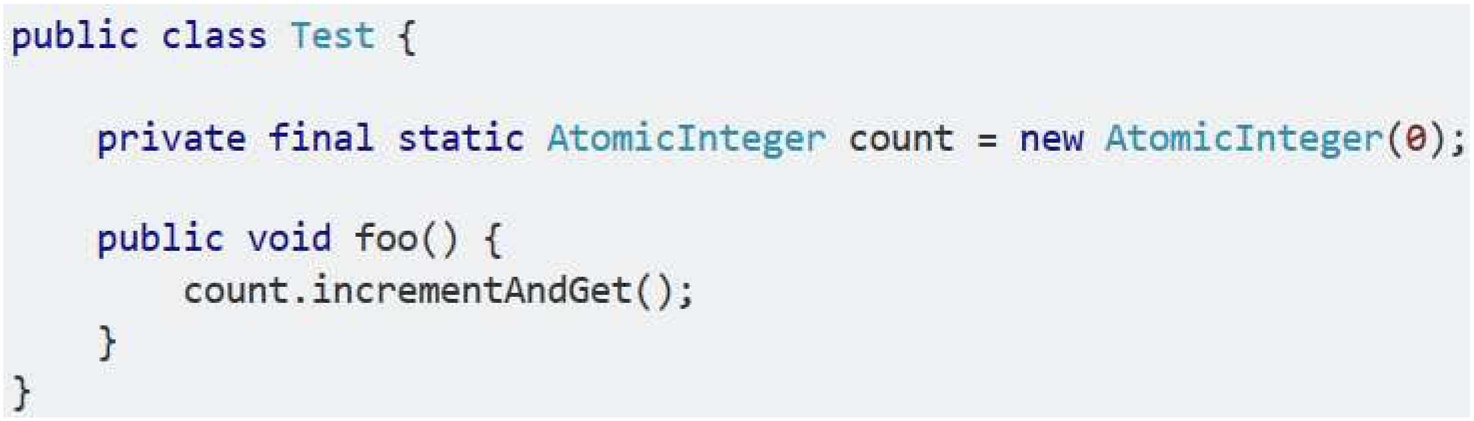


```java
public class Test {
    private static int count = 0;

    public static synchronized void incrementCount() {
        count++;
    }
}
```



```java
public class Test {
    private static int count = 0;
    private static final Object countLock = new Object();

    public void incrementCount() {
        synchronized (countLock) {
            count++;
        }
    }
}
```

Example



```
public class Test {

    private final static AtomicInteger count = new AtomicInteger(0);

    public void foo() {
        count.incrementAndGet();
    }
}
```

Example

Synchronized Collection Methods of Collections class
Collections.synchronizedCollection(Collection<T> c)
Collections.synchronizedList(List<T> list)
Collections.synchronizedMap(Map<K,V> m)
Collections.synchronizedSet(Set<T> s)
Collections.synchronizedSortedMap(SortedMap<K,V> m)
Collections.synchronizedSortedSet(SortedSet<T> s)

```
List<String> syncList = Collections.synchronizedList(new ArrayList<String>());

syncList.add("one");
syncList.add("two");
syncList.add("three");
```