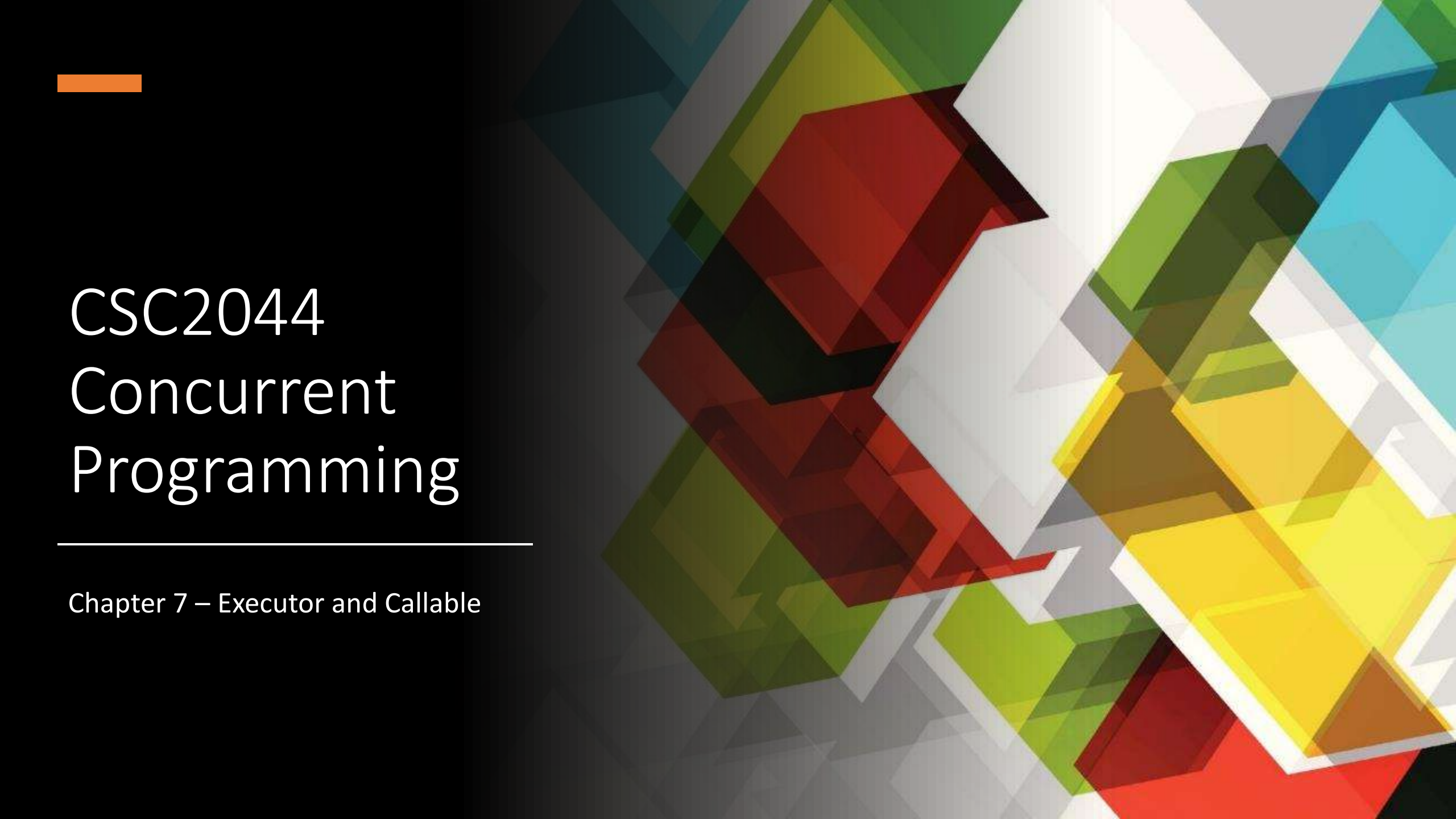


CSC2044 Concurrent Programming

Chapter 7 – Executor and Callable

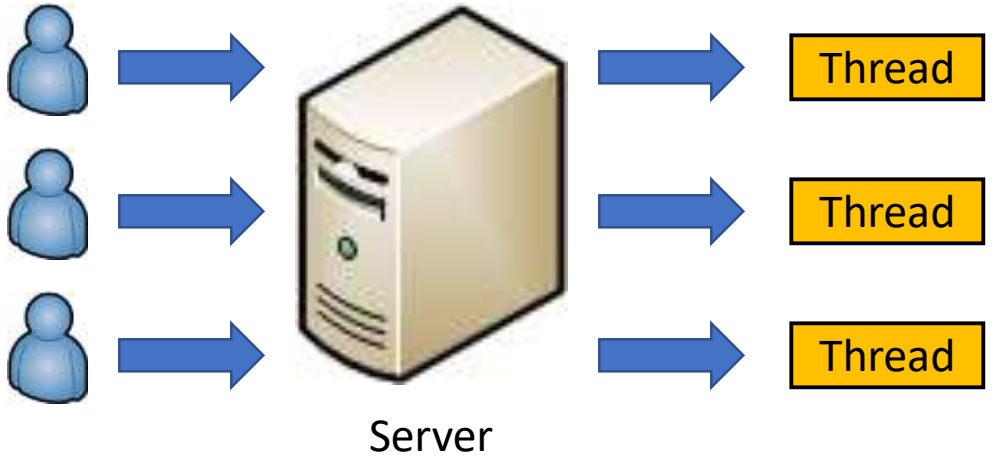
Executor

Executing Tasks using Threads

- So far, we have been trying to create `Runnable` object that contains what we want a thread to do for us (the task).
- We then create a thread based on the `Runnable` object and invoke the start() method
- Not only that we have to define what we want a thread to do for us, but also to create a thread based on it
- In other words, there is a close connection between the task and a thread, that we always think of them as a one and the same

Task Execution

- Explicitly create a thread for each user's request (task).



- What will happen if the number of users continue to grow without limit?

Executing Tasks using Threads

- Explicitly create a thread for each user's request (task).

```
class BookSeat implements Runnable {
     public void run() {
          // Assign task(s) to BookSeat:
     }
}

public class Booking {
     public static void main (String[] args) {
          Thread tA = new Thread(new BookSeat());
          Thread tB = new Thread(new BookSeat());
          Thread tC = new Thread(new BookSeat());

          tA.start();
          tB.start();
          tC.start();
}
```

Executing Tasks using Threads

- This approach is not efficient because, it still takes time to create a new thread (even though it is faster than creating a new process).
- Moreover, it can be cumbersome if we would like to adjust the number of threads based on the number of processing cores available in a system.
- Imagine if we would like to run the same application on two different systems, one with 2 processing cores and one with 16 processing cores.
- Are we going to write an application with a lot of if-else statements?

Thread Overhead and Resource Consumption

- Thread creation and teardown are not free.
  - It takes time (requires some processing by the JVM and OS).
  - Introduces some latency into the process.
  - The overhead varies across platforms.
  - If requests are frequent, the overhead can be significant.
- Active threads consume system resources, especially memory.
- Large number of threads competing for scarce CPU cycles can also impose other performance costs (for example, context switching).
- Out of memory error when the number of threads exceeded the limit

Context Switch

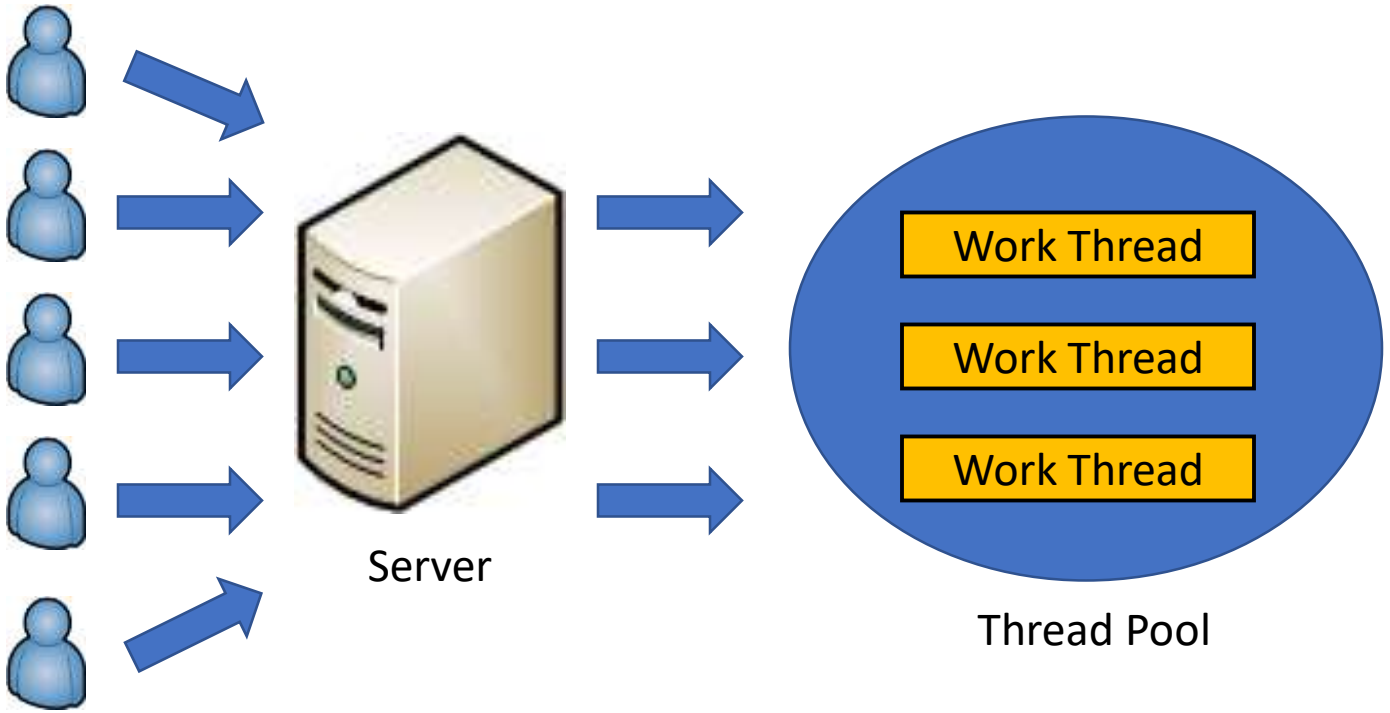
- Save the states associated with the old thread and reload the states associated with the new thread.
- So that the new thread can continue to execute (resume) from the point where it previously stopped.
- A processing core is not doing anything useful during this period of time (reduce performance).

Executor

- Separate the task from a thread
  - a thread is still needed to run a task
  - Number of thread does not have to be the same as number of task
- Create a pool of threads
  - also known as a thread pool with reusable threads
  - the same thread can be used to run different tasks
- Use other advanced features to control how the tasks should be executed
  - for example, run the same task multiple time at certain interval

Executor

- Pre-create a thread pool with limited number of threads.



Executor

- Separating (decoupling) the task from a thread gives us more flexibility when setting up the execution policy to match a specific system (based on hardware).
- The execution policy is used to specify “what, where, when, and how” the tasks should be executed.
- For example:
  1) How many tasks can be executed concurrently?
  2) How many tasks can be queued pending execution?
  3) What actions should be taken before or after executing a task?
  4) How often the tasks should be executed?
  5) …

Executor

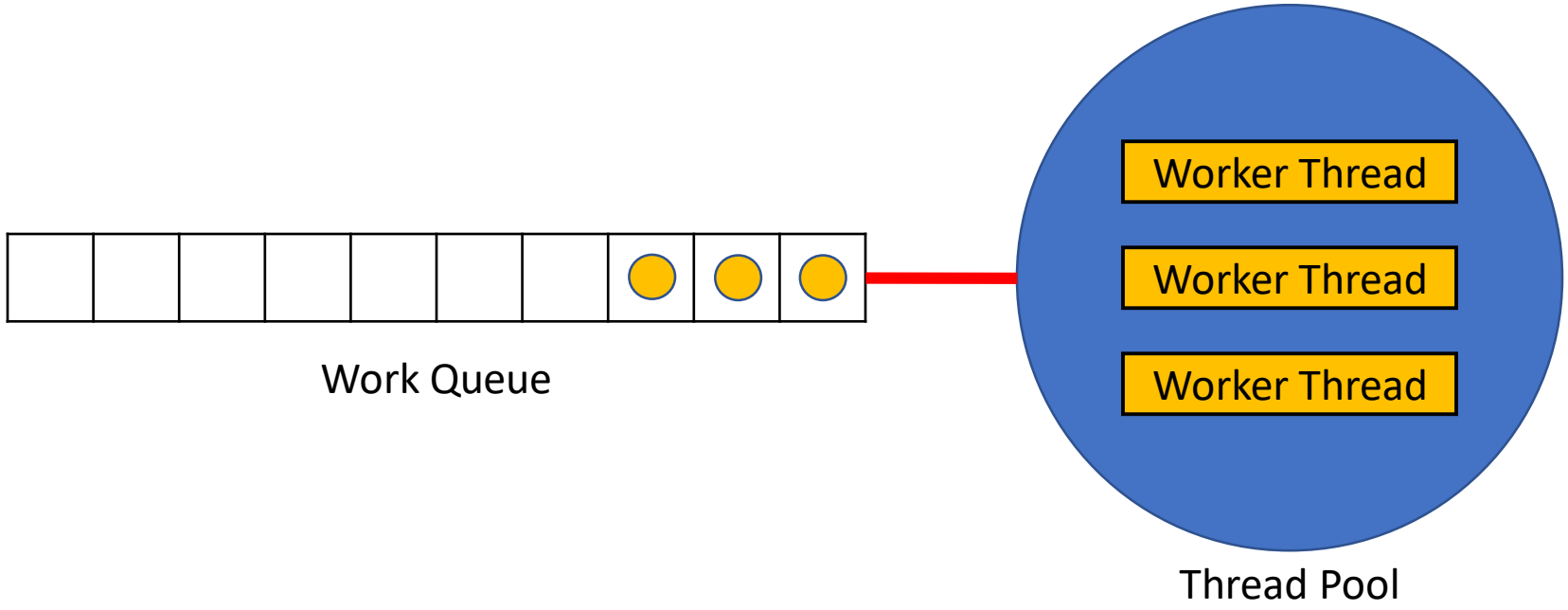
- Before we can start to use the executor, we need to first create a thread pool
- The `Executors` class provides several static (factory) methods for us to create thread pools with different characteristics, and all these methods will return an `ExecutorService` object that can be used to access the thread pool.

```
ExecutorService executor = Executors.newFixedThreadPool(2);
```

- The code fragment shown above will create a thread pool with 2 threads.
- Because the thread pool plays an important role when we used the executor, we will first go though some basics related to thread pool.

Thread Pool

- A pool with certain number of pre-created and reusable threads.

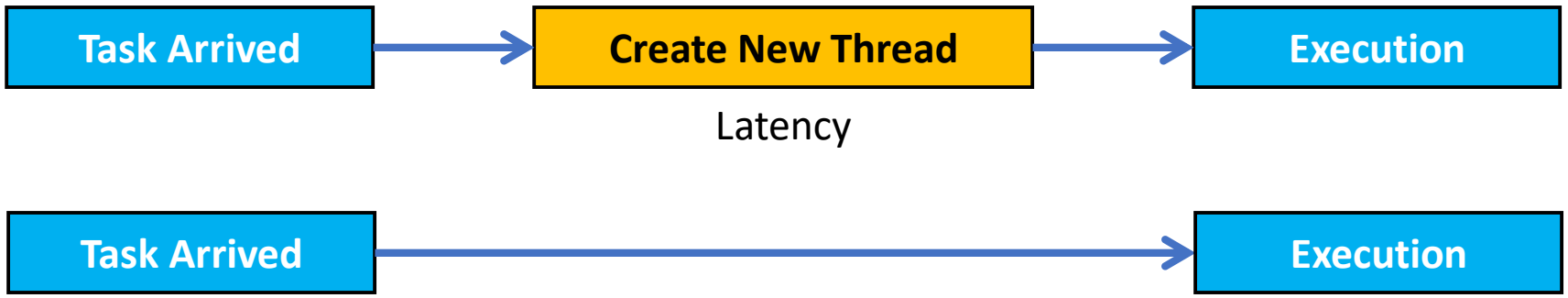


Thread Pool

- It is tightly bound to a work queue holding tasks waiting to be executed.
- The worker threads simply request a task from the work queue, execute it, and go back to waiting for another task.
- Reusing existing thread instead of creating a new one.
- Prevent from being overloaded by creating excessive number of threads.
- With some proper tuning, the thread pool can have enough threads to keep the processing cores busy without worrying about creating too many threads.

Thread Pool

- Using pre-created threads can help to reduce latency (or improve responsiveness), because there is no need to wait for thread creation when a task arrived.



Thread Pool

- The `Executors` class provides several static methods for thread pool creation:
  1) newSingleThreadExecutor
     - A pool with only one thread. All submitted tasks will be executed sequentially.
  2) newFixedThreadPool(int N)
     - A fixed-size pool with N threads. A thread can be reused to execute another task after the current task has finished.
  3) newCachedThreadPool
     - A variable-size pool that grows as needed. Threads available can be reused for new tasks, but a thread will be removed from the pool when idle for 60s.
  4) newScheduledThreadPool(int N)
     - Similar to newFixedThreadPool(int N), but mainly for executing tasks after certain delay or periodically.

Thread Pool

- In general, thread pool is best for executing tasks that are homogenous:
  - similar in term of workload
  - independent.
- If the tasks are somewhat dependent on each other:
  - then we must be careful when sizing the thread pool
  - determine the number of threads the thread pool should maintain

Sizing Thread Pool

- Using threads in a thread pool to execute tasks that are dependent on each other may lead to deadlock.
- Assume that we have prepared two tasks, namely A and B, where task A will need some inputs from task B in the middle of execution.
- Let say both tasks are submitted to a thread pool with only a single thread.
- If task A is first submitted and executed, then the thread will be blocked when the task needs the inputs from task B.
- At this point, we can say that the application has run into a deadlock, because the thread will be blocked (waiting) forever, and there is no another thread that can be used to run task B.

Sizing Thread Pool

- The ideal size (number of threads in a thread pool) usually depends on several factors.
- The general rule is to avoid extremes of “too big” and “too small”.
- If the thread pool is too big:
  - Many threads will have to compete for scarce CPU cycles (as well as memory, can lead to higher memory usage).
- If the thread pool is too small:
  - Unable to utilize unused processing core(s) despite the tasks are queueing (no thread to fetch the task for execution).

Sizing Thread Pool

- To size a thread pool properly, we need to understand the resources available, as well as the nature of our tasks.
  - How many processing cores does the system have?
  - How long a task would take to run?
  - …
- If we have tasks with different nature (for example, some take longer time to run, some take shorter time to run), we can consider to use multiple thread pools so that each thread pool can be tuned accordingly.
- Mixing the two could lead to increase in waiting time (for example, shorter tasks will have to wait longer when they are queuing behind a long task).

Sizing Thread Pool

- For tasks that are CPU-bound, a system with N_{PC} processing cores usually achieves optimum utilization with a thread pool of N_{PC} + 1 threads. This can be done by calling the `availableProcessors()` method from the `Runtime` class.

```
int N_PC = Runtime.getRuntime().availableProcessors();
```

- The extra thread is used to prevent CPU cycles from going unused.
- Alternatively, we can also tune the number of threads by running the application using thread pools with different sizes and observe the utilization.

Executor

- Four general steps when using the executor:
  1) Define and create the tasks.
  2) Create a thread pool by using one of the static methods.
  3) Submit tasks to thread pool (or the executor).
  4) Terminate the threads in the thread pool (or shutdown the executor).

Delegate Tasks to the Threads in a Thread Pool

- After a thread pool is created, we can call the submit() method to submit a task to the thread pool.
- One of the threads in the thread pool will be selected to run the task.
- But if none of the threads is available, the task will need to wait.
- A `Future` object will be returned for us to check the status of the task by calling the `get()` method.
- It will return **null** if the task has finished successfully.

```
ExecutorService executor = Executors.newFixedThreadPool(2);
Future future = executor.submit(new RunnableTask());
future.get();
```

Delegate Tasks to the Threads in a Thread Pool

- There are several possible outcomes when calling the `get()` method:
  - It will block if the task is still halfway running.
  - It will throw `ExecutionException` if the task completed by throwing an exception.
  - It will throw `CancellationException` if the task is cancelled.
  - It will throw `InterruptedException` if the thread executing the task is interrupted.

Terminate Threads in a Thread Pool

- Threads inside a thread pool will continue to stay even though the main thread (the one that runs the main() method) has terminated. This can prevent the main application from stopping correctly.
- Call the shutdown() method to terminate the threads in a thread pool. However, it will still allow the existing tasks to complete before shutting down. Just that it will no longer accept any new tasks.
- If the threads have to be terminated immediately, the shutdownNow() method can be used. But there is no guarantee on stopping the tasks that are halfway running (best effort basis).

```
ExecutorService executor = Executors.newFixedThreadPool(2);
Future future = executor.submit(new RunnableTask());
future.get();
executor.shutdown();
```

Example

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class PrintMyName implements Runnable {
public void run() {
      System.out.println(Thread.currentThread().getName());
      try {
         Thread.sleep(1000);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
}

public class ExecutorExample {
   public static void main(String[] args) {
      ExecutorService executor = Executors.newFixedThreadPool(2);     // Create a thread pool with 2 threads.
      for(int i = 0; i < 10; i++) {
         executor.submit(new PrintMyName());                          // Submit 10 tasks to the thread pool.
      }
      executor.shutdown();                                            // Terminate threads in the thread pool.
   }
}
```

Sample Output:

```
pool-1-thread-2
pool-1-thread-1
pool-1-thread-2
pool-1-thread-1
pool-1-thread-1
pool-1-thread-2
pool-1-thread-1
pool-1-thread-2
pool-1-thread-2
pool-1-thread-1
```

Sample Output:

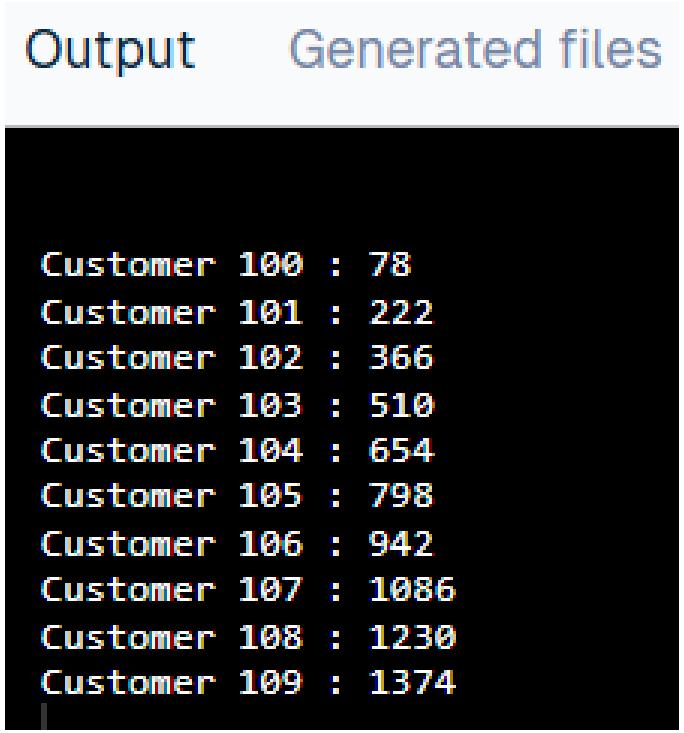
```
pool-1-thread-2
pool-1-thread-1
pool-1-thread-1
pool-1-thread-2
pool-1-thread-2
pool-1-thread-1
pool-1-thread-2
pool-1-thread-1
pool-1-thread-2
pool-1-thread-1
```

Exercise Code 1

Customer ID	Expenses												Total
	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	
100	1	2	3	4	5	6	7	8	9	10	11	12	
101	13	14	15	16	17	18	19	20	21	22	23	24	
102	25	26	27	28	29	30	31	32	33	34	35	36	
103	37	38	39	40	41	42	43	44	45	46	47	48	
104	49	50	51	52	53	54	55	56	57	58	59	60	
105	61	62	63	64	65	66	67	68	69	70	71	72	
106	73	74	75	76	77	78	79	80	81	82	83	84	
107	85	86	87	88	89	90	91	92	93	94	95	96	
108	97	98	99	100	101	102	103	104	105	106	107	108	
109	109	110	111	112	113	114	115	116	117	118	119	120	

Exercise Code 1

- We will create a toy data set that will be stored into a two-dimensional array
- We want to calculate total expenses for each customer
- The total expenses will be written into the last column of the array
- Create a thread pool with 5 threads
- Use the template ExerciseCode1.java and fill in the blanks



```java
import java.util.concurrent.*;

class SumRow implements Runnable {
    private int[][] customerData;
    private int rowNumber;
    private int total = 0;

    SumRow(int[][] customerData, int rowNumber) {
        this.customerData = customerData;
        this.rowNumber = rowNumber;
    }

    @Override
    public void run() {
        // j - column index
        // j:0 -> Customer ID
        // j:13 -> Total Expenses
        // j loop from 1 to 12 (Jan to Dec)
        for(int j = 1; j < 13; j++) {
            total = total + customerData[rowNumber][j];
        }
        customerData[rowNumber][13] = total;
    }
}

public class Ex1 {
    public static void main(String[] args) {
        // Create an array with 10 rows and 14 columns.
        int[][] customerData = new int[10][14];
        int customerStartID = 100;
        int expenses = 1;
        // i - row index, j - column index
        // Just to fill the array with some values for calculation purpose.
        for(int i = 0; i < 10; i++) {
            customerData[i][0] = customerStartID;
            customerStartID++;
            for(int j = 1; j < 13; j++) {
                customerData[i][j] = expenses;
                expenses++;
            }
        }
```

```
43          // Create a thread pool with 5 threads.
44          Executor_________ executor = Executors.__________(__);
45
46          // Submit 10 tasks to the executor.
47          // Each task is going to calculate the total expenses of one customer.
48          for(int i = 0; i < 10; i++) {
49              executor._______((new ________(customerData,__)));
50          }
51
52          executor.shutdown();
53
54          // Print out the last column (total expenses) to check the final answer.
55          for(int i = 0; i < 10; i++) {
56              System.out.println("Customer " + customerData[i][__] + " : " + customerData[i][__]);
57          }
58
59      }
60  }
```

Callable

Limitation of Runnable

- So far, we have been trying to create `Runnable` object and pass it to the `Thread` class constructor to create a thread.
- However, you might have also realized that we never try to return any value from the `Runnable` object we created.
- The fact is that we are unable to do so because the `run()` method cannot return a value.
- But we can still work around with this by writing the value to a shared structure (for example, a buffer / array).

Callable and Future

- Many tasks are effectively deferred computations.
  - Executing database query.
  - Computing complicated function.

- `Callable` will return a value and anticipates that they might throw an exception.

- Can be used only with the Executor framework.

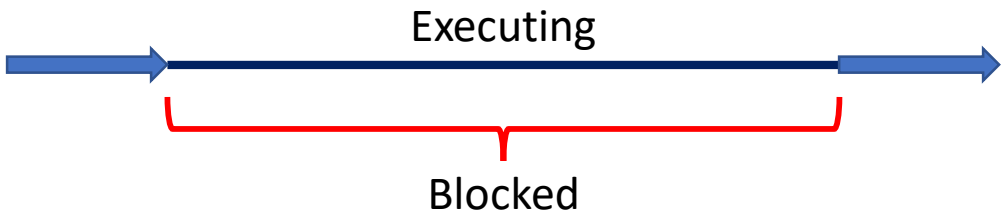
Callable and Future

- Return result can be obtained via the `Future` object.

- Methods Provided:
  - Retrieve the return result.
    - get()
  - Test whether the task has completed or been cancelled.
    - isDone()
    - isCancelled()
  - Cancel the task.
    - cancel(boolean mayInterruptIfRunning)

Callable and Future

- The behavior of `get` varies.
  - It return or throws exception immediately if the task has already completed.
  - Otherwise it will block until the task completes.



  - If the task completes by throwing an exception, `get` throws `ExecutionException`.
  - If it was cancelled, `get` throws `CancellationException`.
  - If the current thread was interrupted while waiting, `get` throws `InterruptedException`.

Callable

- We define the task that a thread should run in the `call()` method (instead of the `run()` method).
- As shown below is an example of creating a `Callable` task.

```
class SumValue implements Callable<Integer> {
	public Integer call() throws Exception {
		int total = 0;
		for (int i = 0; i < 10; i++) {
			total = total + i;
		}
		return total;
	}
}
```

Callable

- After defining the task, we can create the `Callable` objects and submit them to the executor by calling the `submit()` method.

```
public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);
        SumValue sumValueA = new SumValue();
        SumValue sumValueB = new SumValue();
        Future<Integer> futureA = executor.submit(sumValueA);
        Future<Integer> futureB = executor.submit(sumValueB);
        int totalA = 0;
        int totalB = 0;
        try {
                totalA = futureA.get();
                totalB = futureB.get();
        } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
        }
        System.out.println("The total sum is: " + (totalA + totalB));
        executor.shutdown();
}
```

Exercise Code 2

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Exercise Code 2

- We will create another two-dimensional array such as shown in the previous slide
- We want to find the largest value by using Callable interface
- We will create 4 threads in a thread pool
- We will divide the array into four quadrants
  - topLeft
  - topRight
  - bottomLeft
  - bottomRight
- Largest value at each quadrant will be compared with each other
- Use the template ExerciseCode2 and fill in the blanks

```java
import java.util.concurrent.*;

class FindLargestValue implements Callable<Integer> {
    private int[][] intArray;
    private int startRow, startCol, length;
    private int localLargestValue = -1;

    FindLargestValue(int[][] intArray, int startRow, int startCol, int length) {
        this.intArray = intArray;
        this.startRow = startRow;
        this.startCol = startCol;
        this.length = length;
    }

    public Integer call() throws Exception {
        // i - row index, j - column index
        // Assume we are finding the largest value in the top left quadrant.
        // startRow = 0, startCol = 0, length = 4

        // Assume we are finding the largest value in the top right quadrant.
        // startRow = 0, startCol = 4, length = 4
        for(int i = startRow; i < (startRow+length); i++) {
            for(int j = startCol; j < (startCol+length); j++) {
                if (intArray[i][j] > localLargestValue) {
                    localLargestValue = intArray[i][j];
                }
            }
        }
        return localLargestValue;
    }
}

public class Ex2 {
    public static void main(String[] args) {
        // Create an array with 8 rows and 8 columns.
        int[][] intArray = new int[8][8];
        int countValue = 1;
        for(int i = 0; i < 8; i++) {
            for(int j = 0; j < 8; j++) {
                intArray[i][j] = countValue;
                countValue++;
            }
        }
```

```
        // Create a thread pool with 4 threads.
        Executor________ executor = Executors.________(__);

        // Create 4 callable tasks (each for a quadrant).
        FindLargestValue topLeftQuadrant = new FindLargestValue(intArray,__,__,4);
        FindLargestValue topRightQuadrant = new FindLargestValue(intArray,0,__,__);
        FindLargestValue bottomLeftQuadrant = new FindLargestValue(intArray,__,0,__);
        FindLargestValue bottomRightQuadrant = new FindLargestValue(intArray,4,__,4);

        // Submit the tasks to the executor.
        Future<Integer> futureOfTopLeftQuadrant = executor.____(topLeftQuadrant);
        Future<Integer> futureOfTopRightQuadrant = executor._____(topRightQuadrant);
        Future<Integer> futureOfBottomLeftQuadrant = executor._____(bottomLeftQuadrant);
        Future<Integer> futureOfBottomRightQuadrant = executor._____(bottomRightQuadrant);

        int topLeftLargest = 0;
        int topRightLargest = 0;
        int bottomLeftLargest = 0;
        int bottomRightLargest = 0;

        // Retrieve the largest value from each thread.
        _____ {
            topLeftLargest = futureOfTopLeftQuadrant.___();
            topRightLargest = futureOfTopRightQuadrant.____();
            bottomLeftLargest = futureOfBottomLeftQuadrant.____();
            bottomRightLargest = futureOfBottomRightQuadrant.____();
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }

        // Compare the four return values.
        if (topLeftLargest >= topRightLargest && topLeftLargest >= bottomLeftLargest && topLeftLargest >= bottomRightLargest) {
            System.out.println("Largest Value = " + topLeftLargest);
        } else if (topRightLargest >= topLeftLargest && topRightLargest >= bottomLeftLargest && topRightLargest >= bottomRightLargest) {
            System.out.println("Largest Value = " + topRightLargest);
        } else if (bottomLeftLargest >= topLeftLargest && bottomLeftLargest >= topRightLargest && bottomLeftLargest >= bottomRightLargest) {
            System.out.println("Largest Value = " + bottomLeftLargest);
        } else {
            System.out.println("Largest Value = " + bottomRightLargest);
        }

        executor.shutdown();
    }
```

Summary

- Using an executor gives us more flexibility in setting up the execution policy to match specific system (based on hardware).
- Sizing the thread pool is not a simple task. In general, we should avoid extremes of “too big” and “too small”.
- You can switch to the `Callable` interface when it is necessary to use the `call()` method that could return a value.