FUNDAMENTAL PROGRAMMING TECHNIQUES

ASSIGNMENT 2 – SUPPORT PRESENTATION

Outline

- Threads
- Scheduled Execution
- Thread Safety
- Assignment 2 Implementation using Threads
- Java Concurrency – Advanced Concepts

Threads

In **concurrent programming**, there are two basic units of execution: ***processes*** and ***threads***.

On a computer run many active threads and processes, even if there is only a **single-core processor**. This is achieved through the OS feature of ***time-slicing****.*

In Java, concurrent programming is concerned with ***threads****.*

Threads vs Processes

Process

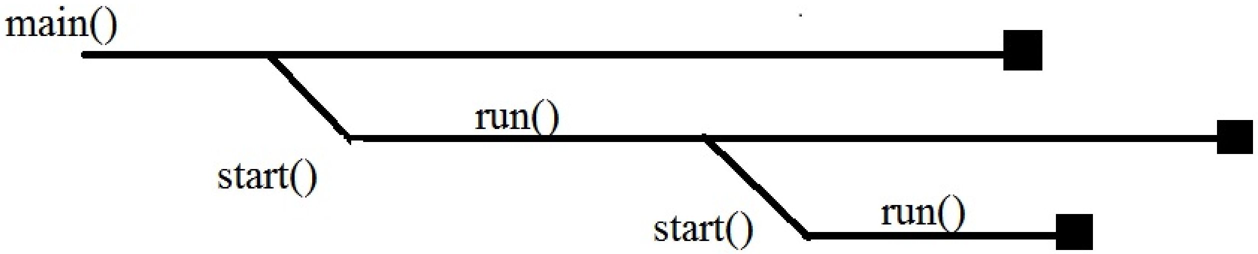
- Synonymous with program or application
- Self-contained execution environment (own memory space)
- Inter Process Communication (IPC) resources, such as pipes and sockets
- Most implementations of the Java VM run as a single process.

Thread

- Lightweight processes
- Exists within a process (each process has at least one thread)
- Share resources (memory and files) => communication problems
- Multithreading is a feature supported by the Java platform.

Threads in Java

"Java threads are objects like any other Java objects. Threads are instances of class **java**.lang.**Thread,** or instances of subclasses of this class. In addition to being objects, java threads can also execute code."



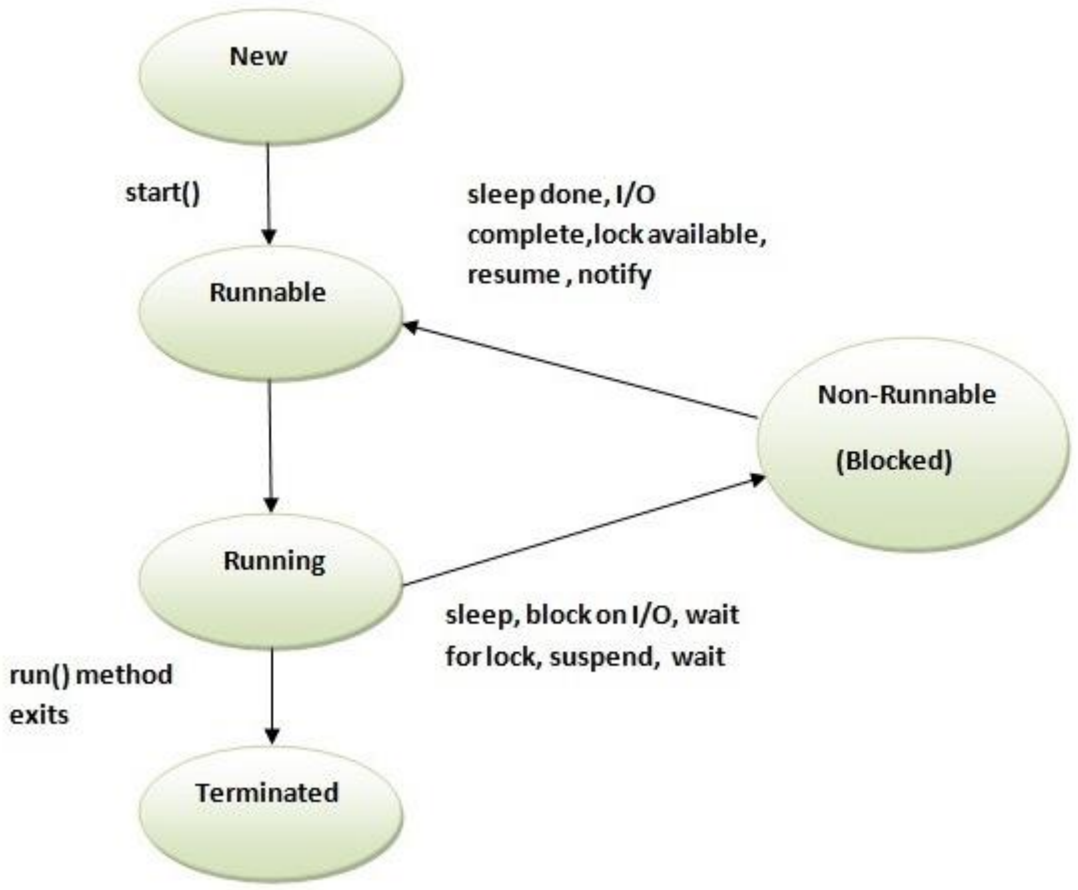
In order to define a thread:

- Extend Thread
- Implement Runnable

Each thread overrides the **run()** method => this will be executed during the thread lifecycle

Threads in Java

Threads Lifecycle



(From http://www.javatpoint.com/life-cycle-of-a-thread)

Threads in Java

Example

```
class MyClass extends Thread{
         public void run() {
              while(condition){
                // code for processing
                sleep(1000);
         }
}
```

Lifecycle:

- Starting a thread
  - Create a thread
  - Call the **start()** method on it => starts executing the **run()** method
- Terminating a Thread
  - When it exists the **run()** method => the thread is dead =>once it is dead it cannot be restarted
- Pausing, suspending and resuming a thread
  - Suspend for a certain amount of time using the **sleep(milis)** method
  - Use The **wait()** method and **notify mechanism**;
- Thread cleanup
  - As long as some other active object holds a reference to the terminated thread object, other threads can execute methods on the terminated thread and retrieve that information.

Threads in Java

1. Create a thread by providing a *Runnable* object

The ***Runnable*** interface defines a single method named *run* – the method will be implemented by the class implementing the interface and it will include the code that will be executed in the thread

```
class MyClass implements Runnable {

  public void run() {
    //Display info about this particular thread
      System.out.println(Thread.currentThread());
  }
}
…
Thread t = new Thread(new MyClass());
t.start();
```

The ***Runnable*** object is passed to the ***Thread*** constructor

2. Create a thread by extending the *Thread* class

The ***Thread*** class itself implements ***Runnable*** - its *run* method does not do anything. A class can extend the Thread class and can provide an implementation of the run method.

```
class MyClass extends Thread{

  public void run() {
    //Display info about this particular thread
      System.out.println(Thread.currentThread());
  }
}
…
MyClass t = new MyClass ();
t.start();
```

Thread vs Runnable

- Inheritance Option
  - Extends Thread => cannot have other inheritance
- Reusability
  - Implements Runnable =>contains only the functionality we want in the run method
  - extends Thread" contains both thread and job specific behavior code
- Object Oriented Design
  - Implements Runnable => Composite Design => A thread has-a Runnable behavior.
  - "extends Thread" is not a good Object Oriented practice.
- Loosely Coupled
  - Implements Runnable => loosely coupled => splits code into 2 parts: behavior and thread
  - Extends thread => tightly coupled
- Functions overhead
  - "extends Thread" means inheriting all the functions of the Thread class which we may do not need

Threads – pausing execution

Thread.sleep()

- **Used to suspend the execution of a running thread for a specified duration**
- **The current thread will be put in the *wait* state until the wait time ends**

```
public class App
{
   public static void printMessages() throws InterruptedException {
      for (int i=0; i< 10; i++){
         System.out.println( "Sending message number " + i);
         Thread.sleep(4000);
      }
   }

   public static void main( String[] args ) throws InterruptedException {
      App.printMessages();
   }
}
```

Exception thrown in case the current thread is interrupted by another thread while *sleep* is active.

The time for suspending the execution of a running thread must be given in milliseconds and must be a positive number. In the example, the thread's execution is suspended for 4 seconds.

Threads – Interrupts

A thread *t* in the waiting or sleeping state can be interrupted by calling the *interrupt* method declared in the *Thread* class => the thread *t* will exit the wait/sleeping state and will throw an *InterruptedException*

The interrupted thread *t* must handle its interruption using one of the methods below [1]

The thread is invoking methods that throw *InterruptedException* => returns from the *run* method after catching the exception

```
public class MyThread implements Runnable{
  public void run() {
      for (int i=0; i< 10; i++){
         System.out.println( "Sending message number " + i);
         try {
            Thread.sleep(4000);
         } catch (InterruptedException e) {
            System.out.println("Someone interrupted me!");
            return;
         }
      }
   }
}
…
 Thread thread = new Thread(new MyThread());
 thread.start();  thread.interrupt();
```

The thread is not invoking methods that throw *InterruptedException* => it will periodically invoke *Thread.interrupted* to check if an interrupt has been received

```
public class MyThread implements Runnable{
  public void run() {
      for (int i=0; i< 10; i++){
         System.out.println( "Sending message number " + i);
         if(Thread.interrupted()){
            System.out.println("Someone interrupted me!");
            return;
         }
      }
  }
}…
 Thread thread = new Thread(new MyThread());
 thread.start();  thread.interrupt();
```

Scheduled Execution [2]

Timers - used to schedule the specified task for repeated fixed-delay execution, beginning after the specified delay.

- **Steps for scheduling a task using *Timer***

Step 1: Create a subclass of the *TimerTask* class and override the *run* method by specifying the instructions to be executed.

***Step 2**:* Create a thread using the *Timer* class.

- Each *Timer* object has a corresponding background thread that will execute the timer's tasks sequentially

***Step 3**:* Create an object of the subclass created at Step 1.

Step 4: Plan the execution of the object created at Step 3 using the schedule methods from the *Timer* class.

```
public class SendingMessageTask extends TimerTask { //Step 1
   private String message;
   public SendingMessageAction(String aMessage){
      this.message = aMessage;
   }
   @Override
   public void run() {
      System.out.println("Sending the message " + this.message);
   }
}
```

```
…
Timer aTimer = new Timer(); //Step 2
SendingMessageTask sendingMessageTask = new SendingMessageTask("Hello"); //Step 3

//Step 4 – schedule the task for repeated fixed-delay executions – e.g. delay = 1000 milliseconds, time between successive task executions = 2000 milliseconds
aTimer.schedule(sendingMessageTask, 1000, 2000);
```

Thread Safety

1) Stateless objects – no state for that class

- I.e. a class that has no instance vars => its state cannot change by running methods in different threads
- Local vars (in method) are independent per thread (each has its own stack)

=> Thread Safety

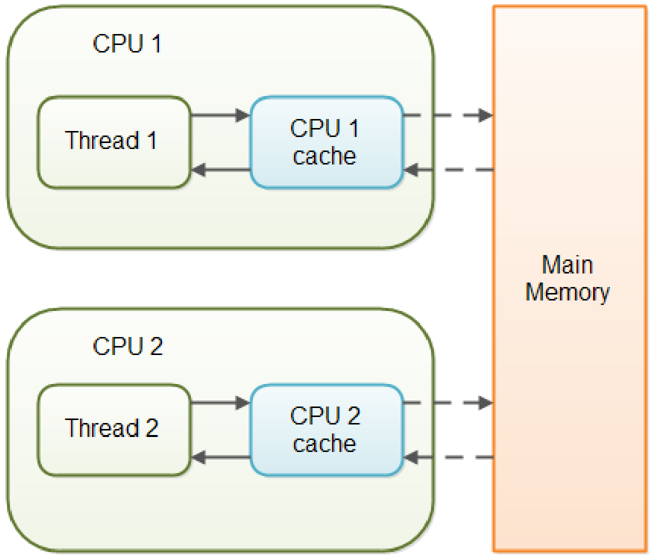
Thread Safety

2) Volatile variables

"The Java volatile keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache."

http://tutorials.jenkov.com/java-concurrency/volatile.html

=> Thread Safety



Thread Safety

3) Locks

- guard a shared resource from accessing or modifying it
    - guard resources in a block using **synchronize**
    - guard resources across blocks using **ReentrantLock**
    - allow multiple accesses to same resource :

CountDownLatch

=> Thread Safety

synchronize

```
synchronized(MyClass.class){
   // some code
 }
Or
synchronized(this){
   // some code
 }
```

ReentrantLock

```
private ReentrantLock lock;
public void foo() { ...
  lock.lock();
  ...}
public void bar() {...
  lock.unlock();
  ...}
```

CountDownLatch

```
CountDownLatch latch= new
CountDownLatch(3);

Public void foo(){...
 latch.countDown();
...}
```

Thread Safety

3) Locks

=> can lead to deadlocks

Solution: sort input accounts and synchronize in-order

Synchronize -deadlock

```
public void transfer(Account a, Account b, double sum)
{

 synchronized(a){   // Th1 locks account a
                    //Th2 locks account b
   synchronized(b){
   //transfer sum
   }
 }

}



Call function :

Th1: transfer(a,b,sum1);
Th2: transfer(b,a,sum1);
```

Thread Safety

4) Atomicity – needed to avoid problems in case of :

- a) ***Problem*: race conditions**

  (getting the right answer depends on lucky timing)

  Solution: synchronization. How?

  1. synchronize entire method

     (inefficient once the instance is created)

  2. synchronize the instantiation piece of code

     (i.e. only if "instance == null")

=> Thread Safety

Race Condition

```
public static Operation getInstance(){
   If(instance ==null){
    instance = new Operation();
 }
return instance;
}
```

- Th2 running
- Th1 running

=> two instances are created

Synchronized

```
public static Operation getInstance(){
   If(instance ==null){
   synchronized(Operation.class){
      If(instance ==null){
         instance = new Operation();
      }
    }
  }
return instance;
}
```

Thread Safety

4) Atomicity – needed to avoid problems in case of :

- **b) *Problem*: compound actions**

  i++;

  Get I value & add one to it => two operations;

  Solution: Atomic data types:

  Ex. AtomicInteger

  => Thread Safety

Compound Operations

```
int i=0;
i++;
/*Accessed simultaneously by both Th1 and Th2
Can lead  to inconsistencies:
- result can be 1(both threads got 0 and
incremented to 1)
- result can be 2(second thread got the value 1 incremented by
the first thread)
```

Atomic Operations

```
AtomicInteger i= new AtomicInteger();
i.getAndIncrement ();
```

Thread Safety

5) Use thread safe collections

A) synchronized collections - synchronizes all methods

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

(!!! For iteration the collection needs to use external sync)

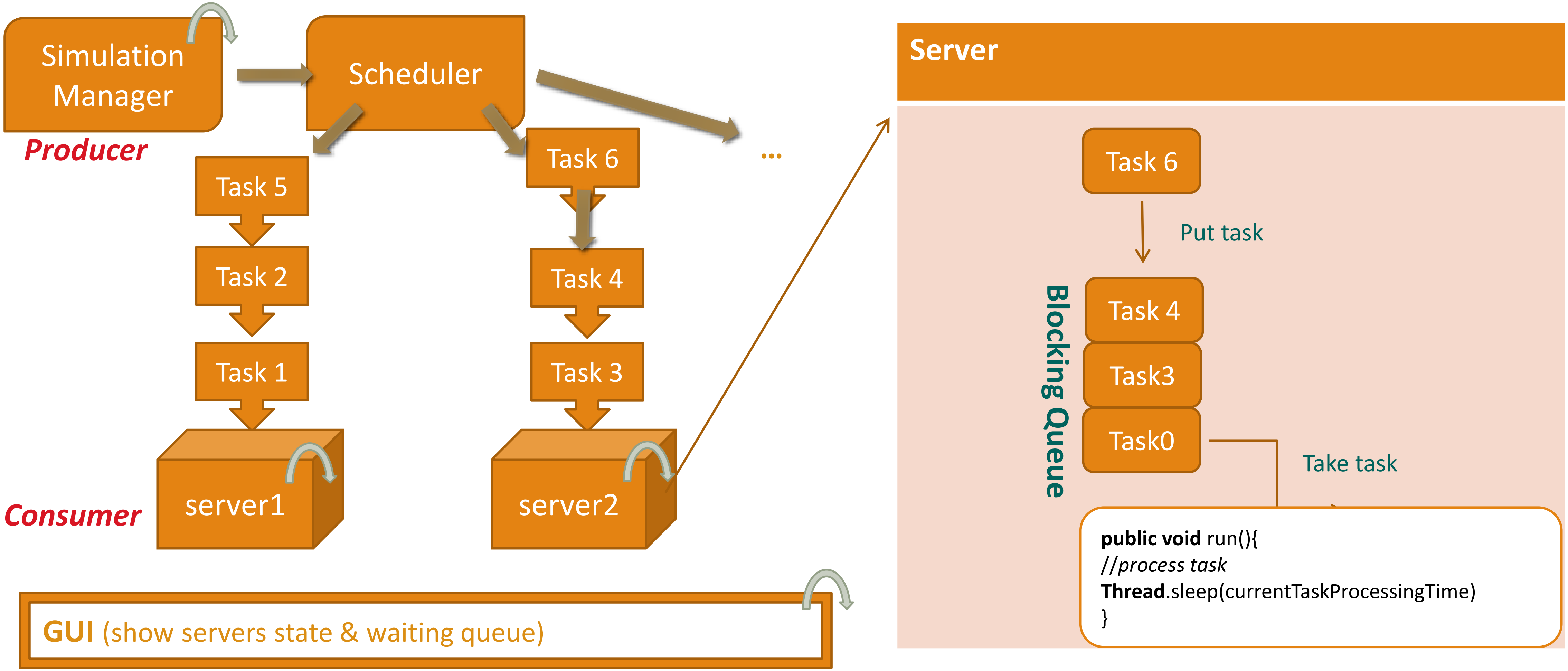
B) use concurrent collections

BlockingQueue

ConcurrentMap - uses a multitude of locks, each lock controls one segment of the hash.

CopyOnWriteArrayList -achieves thread-safety by creating a separate copy of List for each write operation.

Assignment 2 Implementation using Threads



Assignment 2 Implementation using Threads

Task

- Modeled using:
  - arrivalTime
  - finishTime
  - processingPeriod

```
public class Task {

    private int arrivalTime;
    private int processingTime;
    ....
```

- FinishTime = arrivalTime + processingPeriod+ waitingPeriodOnChosenServer

Assignment 2 Implementation using Threads

Server -- Runnable

- Modeled using:
  - Tasks (BlockingQueue<Task>…)
  - WaitingPeriod (AtomicInteger)
    - decremented by current thread once a task is completed
    - incremented by scheduler thread adding new tasks

```java
public class Server implements Runnable {
    private BlockingQueue<Task> tasks;
    private AtomicInteger waitingPeriod;

    public Server() {
        //initialize queue and waitingPeriod
    }

    public void addTask(Task newTask) {
      //add task to queue
      //increment the waitingPeriod
    }

    public void run() {
        while (true) {
            //take next task from queue
            // stop the thread for a time equal with the task's processing time
            // decrement the waitingPeriod
        }
    }

    public Task[] getTasks() {
    ...
    }
}
```

Assignment 2 Implementation using Threads

Scheduler

- Sends tasks to Servers according to the established **strategy**
- Modeled Using :
  - Servers
  - Constraints:

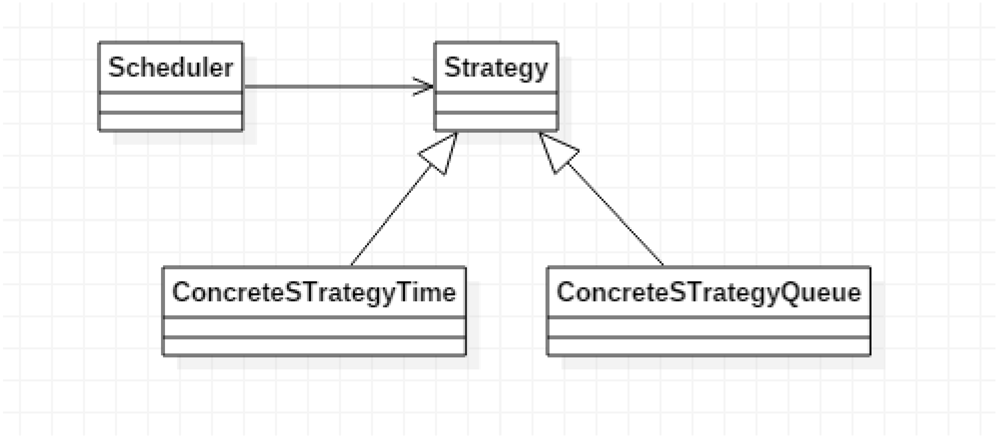
  maxNoServers, maxLoadPerServer

```
public class Scheduler {

    private List<Server> servers;
    private int maxNoServers;
    private int maxTasksPerServer;
    private Strategy strategy;


    public Scheduler(int maxNoServers, int maxTasksPerServer) {
        //for maxNoServers
        // - create server object
        // - create thread with the object

    }


    public void changeStrategy(SelectionPolicy policy){
        //apply strategy patter to instantiate the strategy with the concrete
        //strategy corresponding to policy
        if(policy == SelectionPolicy.SHORTEST_QUEUE){
            strategy = new ConcreteStrategyQueue();
        }
        if(policy == SelectionPolicy.SHORTEST_TIME){
            strategy = new ConcreteStrategyTime();
        }
    }

    public void dispatchTask(Task t) {
        //call the strategy addTask method
    }

    public List<Server> getServers() {
        return servers;
    }
```

Assignment 2 Implementation using Threads

- **Scheduler – Strategy Patter**
- Choose the policy to distribute clients

[(https://en.wikipedia.org/wiki/Strategy_pattern](https://en.wikipedia.org/wiki/Strategy_pattern))




```java
public interface Strategy {

    public void addTask(List<Server> servers, Task t);

}
```

```java
public enum SelectionPolicy {
    SHORTEST_QUEUE, SHORTEST_TIME
}
```

```java
public class ConcreteStrategyTime implements Strategy {

    @Override
    public void addTask(List<Server> servers, Task t) {
        // TODO Auto-generated method stub

    }

}
```

Assignment 2 Implementation using Threads

Simulation Manager- Runnable

- Generates randomly the tasks with:
  - Arrival time
  - processingPeriod
- Contains simulation loop:
  - CurrentTime
  - Call scheduler to dispatch tasks
  - Update UI

```java
public class SimulationManager implements Runnable{
    //data read from UI
    public int timeLimit = 100; //maximum processing time - read from UI
    public int maxProcessingTime = 10;
    public int minProcessingTime = 2;
    public int numberOfServers = 3;
    public int numberOfClients = 100;
    public SelectionPolicy selectionPolicy = SelectionPolicy.SHORTEST_TIME;

    //entity responsible with queue management and client distribution
    private Scheduler scheduler;
    //frame for displaying simulation
    private SimulationFrame frame;
    //pool of tasks (client shopping in the store)
    private List<Task> generatedTasks;

    public SimulationManager(){
        // initialize the scheduler
        //      => create and start numberOfServers threads
        //      => initialize selection strategy => createStrategy
        // initialize frame to display simulation
        // generate numberOfClients clients using generateNRandomTasks()
        //and store them to generatedTasks
    }

    private void generateNRandomTasks(){
        // generate N random tasks:
        // - random processing time
        //minProcessingTime < processingTime < maxProcessingTime
        // - random arrivalTime
        //sort list with respect to arrivalTime
    }
```

Assignment 2 Implementation using Threads

Simulation Manager- Runnable

- Generates randomly the tasks with:
  - Arrival time
  - processingPeriod
- Contains simulation loop:
  - CurrentTime
  - Call scheduler to dispatch tasks
  - Update UI

```java
@Override
public void run() {
    int currentTime = 0;
    while (currentTime < timeLimit){
        // iterate generatedTasks list and pick tasks that have the
        //arrivalTime equal with the currentTime
        //     - send task to queue by calling the dispatchTask method
        //from Scheduler
        //     - delete client from list
        // update UI frame
        currentTime++;
        // wait an interval of 1 second
    }
}

public static void main(String[] args){
    SimulationManager gen = new SimulationManager();
    Thread t = new Thread(gen);
    t.start();
}
```

Java Concurrency

Advanced Concepts

Contents

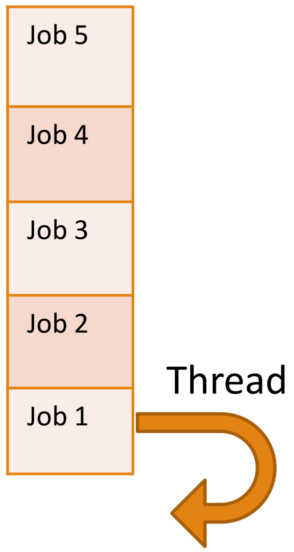
- Single thread design vs Multi-Thread Design
- Java Executor Framework
- Result bearing Jobs: Callable vs Runnable
- Swing Concurrency Support

Job Execution

Web Application executing user jobs

1. Single thread design

- Create one thread per application
  - Poor performance handling only one job at a time
  - The other jobs are waiting for the previous to complete
  - => poor responsiveness

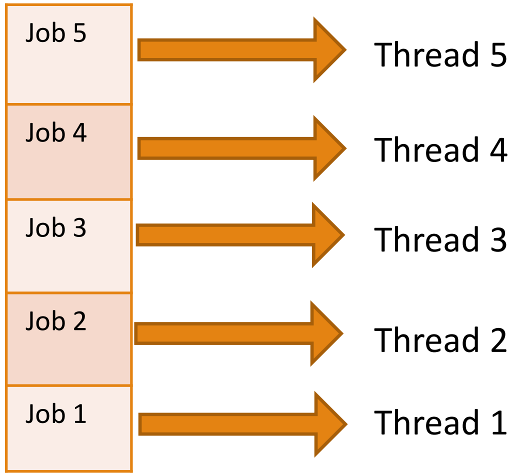


Job Execution

Web Application executing user jobs

2. Multiple threads design

- Create one thread for each job
  - Improved performance – jobs are completed in parallel
  - **!!! Job handling must be thread safe**



Job Execution

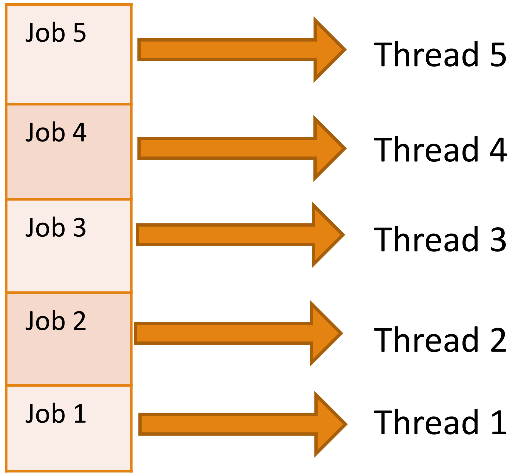
Web Application executing user jobs

2. Multiple threads design

Unbound Threads Creation

1. Thread lifecycle overhead
   - Thread creation and tear down are not free
2. Resource consumption
   - Creating more threads than the available processors does not help

   (it may even hurt : put pressure on garbage collector and compete for CPU)
3. Stability
   - Each system has a limit of threads that can be created influenced by:
     - JVM parameters, stack size, underlying OS, etc.
     - Exceed limit => OutOfMemoryError



Job Execution

Web Application executing user jobs

3. Executor Framework

- **Executor –** simple interface that provides the basis for a flexible and powerful framework
  - Lifecycle support , hooks for adding statistics gathering, application management, and monitoring
  - Select the optimal policy at runtime, depending on the available hardware

Executor

```
Executor exec=Executors.newFixedThreadPool(100);

Runnable task = new Runnable() {
    public void run() {
        //execute job
    }
};
exec.execute(task);
```

Job Execution

Web Application executing user jobs

3. Executor Framework- Thread pools

- *newFixedThreadPool* - fixed size of threads
- *newCachedThreadPool* -
- *newSingleThreadExecutor* –one thread (automatically replaced if it dies)
  - Tasks processed sequentially (FIFO, LIFO, priority, etc.)
- *newScheduledThreadPool* – fixed sized; supports delayed and periodic execution

Job Execution

Web Application executing user jobs

3. Executor Framework -Lifecycle

- Non-daemon threads (failing to shot down an Executor, could prevent JVM from exiting; i.e. thread continues to run even after main tread terminates)
- ExecutorService - interface extending Executor and providing methods for handling lifecycle management operations:
  - shutdown() //graceful shutdown – no new tasks are accepted; the previously submitted tasks are allowed to complete
  - shotdownNow() //abrupt shutdown - cancels all running tasks
  - isShotdown()
  - isTerminated() // after all tasks have terminated , the Executor transitions to **terminated** state;
  - awaitTermination(long timeout, Timeunit unit

Job Execution

Web Application executing user jobs

4. Result bearing Jobs

- Callable vs Runnable
  - Callable = runnable on steroids
- Future – the result of submitting a Callable or a Runnable task to the Executor Service

Callable vs Runnable

```
public interface Callable<V> {
V call() throws Exception;
}

public interface Runnable{
void run();
}

public interface Future<V> {
  boolean cancel(boolean mayInterruptIfRunning);
  boolean isCancelled();
  boolean isDone();
  V get() throws InterruptedException, ExecutionException;
  V get(long timeout, TimeUnit unit)
      throws InterruptedException, ExecutionException, TimeoutException;
}
```

Job Execution

Web Application executing user jobs

4. Result bearing Jobs

- Executor service usage:
  - execute(Runnable) – executes the task – no way of obtaining the result
  - submit(Runnable) - returns a **Future** object – one can check if the Runnable has finished execution

    future.get(); //returns null if the task has finished correctly.
  - submit(Callable) – returns a **Future** object

    future.get(); //returns the value returned by call – it's blocking

Swing Concurrency Support

SwingWorker

- Support for UI (using FutureTask and Executor) :
  - Cancellation
  - Completion notification
  - Progress indication

- *doInBackground* – executes the long job
  - One can publish intermediate results (publish method)
- *done* – called once the *doInBackground* finishes
  - One can access the result by calling get() (see Futures)
- *process* – called asynchronously to process the published information

*** UI components should only be allocated in *done* or *process* which are executed on the Event Dispatch Thread**

```java
public class SwingWorkerExample extends SwingWorker<Integer, Integer> {

  @Override
   protected Integer doInBackground() throws Exception {

           Thread.sleep(1000);
           publish(1);

           Thread.sleep(1000);
           publish(2);

           Thread.sleep(1000);
           publish(3);

           return 13;
    }

   @Override
    protected void done() {
         try {
              JOptionPane.showMessageDialog(null, get());
         } catch (Exception e) {
              e.printStackTrace();
         }
    }

    @Override
    protected void process(List<Integer> v) {
          for (int i=0; i < v.size(); i++) {
             System.out.println("received values: " + v.get(i));
           }
    }
}
```

References

[1] https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

[2] https://docs.oracle.com/javase/8/docs/api/java/util/Timer.html

[3] http://www.tutorialspoint.com/java/util/timer_schedule_period.htm

[4] http://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html

[5] B. Goetz et al., Java Concurrency in Practice, Addison-Wesley Professional; 1 edition (May 19, 2006)