ADVANCED JAVA

Introduction To Concurrency

Vivek Shah bonii@di.ku.dk

August 20, 2018

DIKU, University of Copenhagen

OUTLINE

Concurrency Primer

Thread Creation

Thread Waiting and Interruption

High-Level ThreadPool Abstractions

CONCURRENCY PRIMER

WHAT IS CONCURRENCY?

WHAT IS CONCURRENCY?

- Appear to run several programs or several parts of a program simultaneously.

WHAT IS CONCURRENCY?

- Appear to run several programs or several parts of a program simultaneously.
- Historical progression of concurrency:

WHAT IS CONCURRENCY?

- Appear to run several programs or several parts of a program simultaneously.

- Historical progression of concurrency:
  1. Batch Processing (Hardware). Separate programs.

WHAT IS CONCURRENCY?

- Appear to run several programs or several parts of a program simultaneously.

- Historical progression of concurrency:
  1. Batch Processing (Hardware). Separate programs.
  2. Processes (OS). Separate address spaces.

WHAT IS CONCURRENCY?

- Appear to run several programs or several parts of a program simultaneously.

- Historical progression of concurrency:
  1. Batch Processing (Hardware). Separate programs.
  2. Processes (OS). Separate address spaces.
  3. Threads (Application). Same address space.

WHAT IS CONCURRENCY?

- Java is a multi-threaded language
    - Thread support for sequential programs.
    - Well-defined memory model.

WHAT IS CONCURRENCY?

- Java is a multi-threaded language
  - Thread support for sequential programs.
  - Well-defined memory model.
- Advantages of using Threads:
  1. Light-weight.
  2. Flexibility.
  3. Ease of resource sharing.

WHAT IS CONCURRENCY?

- Java is a multi-threaded language
  - Thread support for sequential programs.
  - Well-defined memory model.
- Advantages of using Threads:
  1. Light-weight.
  2. Flexibility.
  3. Ease of resource sharing.
- Disadvantages of using Threads:
  - With great powers come great responsibilities.

THE NEED FOR CONCURRENCY

THE NEED FOR CONCURRENCY

1. Multi-processor system
    - Faster execution.

THE NEED FOR CONCURRENCY

1. Multi-processor system
    - Faster execution.
2. Single processor system
    - Faster execution/performance.
    - Sounds counter intuitive. What about the overheads of context switches ?

THE NEED FOR CONCURRENCY

1. Multi-processor system
    - Faster execution.
2. Single processor system
    - Faster execution/performance.
    - Sounds counter intuitive. What about the overheads of context switches ?
    - Waiting
        - For Memory, Disk, Network or something else.
        - For responsiveness in user interactive interfaces (Event driven programming).

THE NEED FOR CONCURRENCY

1. Multi-processor system
    - Faster execution.
2. Single processor system
    - Faster execution/performance.
    - Sounds counter intuitive. What about the overheads of context switches ?
    - Waiting
        - For Memory, Disk, Network or something else.
        - For responsiveness in user interactive interfaces (Event driven programming).
    - Improving code design
        - Modeling simulations (games).
        - No need to worry about scheduling.
        - Modeling distributed systems (e.g., actor model)

THREAD CREATION

THREADS IN JAVA

The `java.lang.Runnable` interface:

```java
public interface Runnable {
    void run();
}
```

- A simple interface to a runnable piece of code.
- A thread's execution entry point is specified using `Runnable`.

THREADS IN JAVA - THE IMPORTANT METHODS

```java
class Thread implements Runnable {
  Thread(Runnable target)
  static Thread currentThread()
  void run()
  void start()
  static void sleep(long millis) throws InterruptedException
  static void yield()
  void join() throws InterruptedException
  void interrupt()
  boolean isInterrupted()
  static boolean interrupted()
}
```

CREATING THREADS

Method 1: Implement the `java.lang.Runnable` interface.

```
public class HelloWorldRunnable implements Runnable {
    public void run() {
        System.out.println(”Hello World”);
    }

    public static void main(String[] args) {
        new Thread(new HelloWorldRunnable()).start();
    }
}
```

CREATING THREADS

Method 2: Extend the `java.lang.Thread` class.

```
public class HelloWorldThread extends Thread {
    public void run() {
        System.out.println(”Hello World”);
    }

    public static void main(String[] args) {
        new HelloWorldThread().start();
    }
}
```

CREATING THREADS

- A thread performs the task defined in `run` method.
- `implement java.lang.Runnable interface` or `extend java.lang.Thread class`

CREATING THREADS

- A thread performs the task defined in `run` method.
- `implement java.lang.Runnable interface` or `extend java.lang.Thread class`
    - Implementing `java.lang.Runnable` provides purer composition.
    - Extending java.lang.Thread is useful for simple applications but limited owing to sub-classing.

THREAD WAITING AND INTERRUPTION

THREAD WAITING

- `Thread.sleep()` and `Thread.join()` cause the current thread to wait.
- `Thread.sleep(x)` - Wait for x milliseconds.
    - Why is sleep static?

THREAD WAITING

- `Thread.sleep()` and `Thread.join()` cause the current thread to wait.
- `Thread.sleep(x)` - Wait for x milliseconds.
  - Why is sleep static?
  - Because a thread cannot put another thread to sleep.
- `t.join()` - Wait until thread t dies.

THREAD INTERRUPTION

- Calling `x.interrupt()` signals thread x to stop doing what it is doing and do something else. (e.g., terminate).
- From the JavaDoc:
  "If this thread is blocked in an invocation of ... `join()` *... or* `sleep(long)`*, then its interrupt status will be cleared and it will receive an* `InterruptedException`*."*.
  ...
  "If none of the previous conditions hold then this thread's interrupt status will be set."

THREAD INTERRUPTION

- What happens when a thread is interrupted ?
    - If the thread is waiting, `java.lang.InterruptedException` is thrown.
    - Otherwise, nothing happens.
    - Unless the thread inspects its interrupted status regularly.

THREAD INTERRUPTION

- What happens when a thread is interrupted ?
  - If the thread is waiting, `java.lang.InterruptedException` is thrown.
  - Otherwise, nothing happens.
  - Unless the thread inspects its interrupted status regularly.

- Good practice:
  - Do not use `while(true)` to make a thread run forever.
  - Use `while(!Thread.interrupted())` instead.

THREAD MANIPULATION

- `Thread.yield()` signals the scheduler to schedule another thread for execution. Just a hint.

THREAD MANIPULATION

- `Thread.yield()` signals the scheduler to schedule another thread for execution. Just a hint.
- `Thread.setPriority()` can set the scheduling priority of a thread.
  - `MAX_PRIORITY`, `MIN_PRIORITY`, `NORM_PRIORITY`.

THREAD MANIPULATION

- `Thread.yield()` signals the scheduler to schedule another thread for execution. Just a hint.
- `Thread.setPriority()` can set the scheduling priority of a thread.
  - `MAX_PRIORITY`, `MIN_PRIORITY`, `NORM_PRIORITY`.
- User Threads and Daemon Threads.
  - JVM runs a process until at least one user thread is running.
  - `Thread.setDaemon(boolean)` function can turn on/off the daemon status of a thread.

THREAD MANIPULATION

- Sending and handling Interrupts.

THREAD MANIPULATION

- Sending and handling Interrupts.
- `t.interrupt()` sends an interrupt to thread t for handling. Thread t can detect the interrupt by

THREAD MANIPULATION

- Sending and handling Interrupts.

- `t.interrupt()` sends an interrupt to thread t for handling. Thread t can detect the interrupt by
    - Catching `java.lang.InterruptedException`.
        - If you do nothing, set the status back.
        - `Thread.currentThread.interrupt()`.

THREAD MANIPULATION

- Sending and handling Interrupts.
- `t.interrupt()` sends an interrupt to thread t for handling. Thread t can detect the interrupt by
  - Catching `java.lang.InterruptedException`.
    - If you do nothing, set the status back.
    - `Thread.currentThread.interrupt()`.
  - Checking status using `Thread.interrupted()`. Do not check after catching an `InterruptedException`.

THREAD MANIPULATION

- Sending and handling Interrupts.

- `t.interrupt()` sends an interrupt to thread t for handling. Thread t can detect the interrupt by
  - Catching `java.lang.InterruptedException`.
    - If you do nothing, set the status back.
    - `Thread.currentThread.interrupt()`.
  - Checking status using `Thread.interrupted()`. Do not check after catching an `InterruptedException`.

- Interrupts are used to cancel a thread. Good programs support thread cancellation.

THREAD MANIPULATION

- Sending and handling Interrupts.
- `t.interrupt()` sends an interrupt to thread t for handling. Thread t can detect the interrupt by
  - Catching `java.lang.InterruptedException`.
    - If you do nothing, set the status back.
    - `Thread.currentThread.interrupt()`.
  - Checking status using `Thread.interrupted()`. Do not check after catching an `InterruptedException`.
- Interrupts are used to cancel a thread. Good programs support thread cancellation.
- Let us see thread waiting and manipulation in action.

what can we do (safely)?

- Concurrency with no shared memory.
    - No communication (besides interruptions).
    - Data parallelism.

what can we do (safely)?

- Concurrency with no shared memory.
  - No communication (besides interruptions).
  - Data parallelism.

- Parallel version of quicksort.
- Parallel linear search.
- Parallel binary search.

WHAT CAN WE DO (SAFELY)?

- Concurrency with no shared memory.
    - No communication (besides interruptions).
    - Data parallelism.

- Parallel version of quicksort.
- Parallel linear search.
- Parallel binary search.

- Algorithms where threads can work independently.

HIGH-LEVEL THREADPOOL ABSTRACTIONS

USING EXECUTORS AND THREADPOOLS

- Thread creation and destruction are expensive.
- Large scale applications require separate thread management and creation.
- `java.util.concurrent` package provides thread management via Executors.
- Executors are a layer of indirection between client and execution of a task.
- Multiple types of threadpools like `CachedThreadPool`, `FixedThreadPool`.
- Frees the application from thread-management and focus on resource sharing and co-ordination among threads.

CALLABLE AND FUTURE

- `java.util.concurrent.Callable` is an extension of the `java.lang.Runnable` interface which represents a task that returns a result or throws an exception.
- Must implement `call()` method instead of `run()` which returns a result.

CALLABLE AND FUTURE

- `java.util.concurrent.Callable` is an extension of the `java.lang.Runnable` interface which represents a task that returns a result or throws an exception.
- Must implement `call()` method instead of `run()` which returns a result.
- Is only supported for execution through `java.util.concurrent.Executor` interface.
- Submitting a task obtains a `java.util.concurrent.Future` object.

callable and future (contd.)

- Calling `get()` on the object blocks if the thread has not completed.
  - Timed version of `get()` is available.
  - `isDone()` is a non blocking version which can check to see if the result is available.
- Calling `cancel(bool)` interrupts the thread executing the task.

CALLABLE AND FUTURE (CONTD.)

- Calling `get()` on the object blocks if the thread has not completed.
  - Timed version of `get()` is available.
  - `isDone()` is a non blocking version which can check to see if the result is available.
- Calling `cancel(bool)` interrupts the thread executing the task.
- Let us see `Executors` and `Callable` in action.

THREAD SHARING AND SYNCHRONIZATION - SAFETY

- Threads do not always operate on independent resources or wait for other threads to finish.

THREAD SHARING AND SYNCHRONIZATION - SAFETY

- Threads do not always operate on independent resources or wait for other threads to finish.

- With concurrency resource contention and sharing is a problem that needs to be tackled.
- Proper access to resources must be ensured.

THREAD SHARING AND SYNCHRONIZATION – SAFETY

- Threads do not always operate on independent resources or wait for other threads to finish.

- With concurrency resource contention and sharing is a problem that needs to be tackled.
- Proper access to resources must be ensured.

- Mechanisms that can be used.
    - Monitors.
    - Using explicit Lock objects.
    - Atomic classes and volatility.
    - High-level synchronizers.

WORK SESSIONS

- Assignment 1 will be released.
- Lab sessions (work on assignment) will take place from 13:30 – 17:00.
- Work in groups of 2-3. Inform us of your group by 14:00.
- No need for a report, comment your code :-).
- Deadline 23:59 hrs today.
- **Do not hesitate, ask questions if need be.**
- Any solution that conforms to the handed out interfaces is acceptable.