Concurrency

Fundamentals

Why we need Concurrency?

1. Resposiveness (Concurrency Multitasking)
2. Performance (Parallelism)

Key Points

A process is completely isolated from any other process.

In Multithreaded application each thread comes with its own `stack` and `Instruction Pointer.`

Multiple threads in a single process share:

- The Heap (Data)
- Code
- The process's open files
- The process's metadata

Context Switch

- Stop Thread 1
- Scheduled Thread 1 out
- Scheduled Thread 2 in.
- Start Thread 2

Threads consumes less resources than processes. `Context Switching` between thread from the same process is cheaper.

Too Many Threads - `Thrashing` - spending more time in management than real productive work.

See Image Gallery for more visualization.

Thread Creation (Java)

There are two ways to run code on a new thread:

1. Implements Runnable interface, and pass to a new Thread Object.
2. Extends Thread class, and create and object of that class.

`Thread class` - Encapsulates all thread related functionality.

Code Snippet 1:

```java
        public static void main(String [] args) {
            Thread thread = new TaskThread1();
            thread.start();
        }

        private static class TaskThread1 extends Thread {
            @Override
            public void run(){
                System.out.println("Hello from new thread");
            }
        }
```

Code Snippet 2:

```java
        public static void main(String [] args) {
            Thread thread = new Thread(new Task2());
            thread.start();
        }

        private static class Task2 implements Runnable {
            @Override
            public void run(){
                System.out.println("Hello from new thread");
            }
        }
```

Thread Termination (Interrupt)

Good to know

- Thread Consumes resources
  - Memory and Kernel Resources
  - CPU cyles and cache Memory
- If a thread finished its work, but the application is still running, we want to clean up the thread's resources. (Memory and Kernel Resources).
- The application will not stop as long as at least one thread is still running.

Thread.interrupt()

1. If the thread is executing a method that throws an InterruptedException.
2. If the thread's code is handling the interrupt signal explicitly.

Daemon Thread

Background threads that do not prevent the application from exiting if the main thread terminates.

```
    Thread th = new Thread();
    th.setDaemon(true);
```

Thread Coordination

Why do we need it?

- Different threads run independently.
- Order of execution is out of control.
- Thread Dependency.

Approach 1(Naive Solution): Thread B runs in a loop and keeps checking if Thread A's result is ready.
Cons: Burn CPU cycles unneccessary.

Approach 2: `Thread.join()`

```
     // method signature
        public final void join();
        public final void join(long millis, int nanos);
        public final void join(long millis);
```

Tips:- Threads may take unreasonably long time. Always use the `Therad.join(..)` with a time limit.

Performance In Multithreading

Performance Measurement:

- Latency
- Throughput

`Latency:` The time to completion of a task. Measured in time untis.

`Throughput:` The amount of tasks completed in a given period. Measured in tasks/time unit.

Improving Latency:
Suppose we a divide the task in N different independent task and for each task create a seperate thread.
Then, *Theoreticaly* reduction of latency by N = Performance improvement by a factor of N.
Cons: All task can't divided in to N independent task.

Open Question:

1. N = ?, How many substasks/thread to break the original task?
2. Does breaking original task and aggregation results come for free?

3. There is **inherent cost** of running an algorithm by multiple threads.
4. Can we break any task into subtasks?

Inherent cost

- Breaking task into multiple tasks.
- Thread creation, passing tasks to threads
- Time between thread.start() to thread getting scheduled
- Time until the last thread finishes and signals
- Time until the aggregating thread runs.
- Aggreagation of the subresults into a single arttifact.

Throughput (improvise):
Let's run N task in parallel. So, in this way Theoreticaly, we can do N task in same amount of time.

Pros:

1. No Need to break the task.

Open Questiton:

1. Value of N? , How many task should run in parellely.
2. Can we fixed the cost of creating thread (irrespective of task)?
3. Lets there 100 task and I want to run parallely, how to maintain effectively.

Inherent cost

- ~~Breaking task into multiple tasks.~~
- Thread creation, passing tasks to threads
- Time between thread.start() to thread getting scheduled
- ~~Time until the last thread finishes and signals.~~
- ~~Time until the aggregating thread runs.~~
- ~~Aggreagation of the subresults into a single arttifact.~~

Value of N

`# threads` = `# cores` is optimally only if all threads are runnable and can run without interruption (no IO/blocking calls/ sleep etc).

The assumption is nothing else is running that consumes a lot of CPU.

Data Sharing between Threads

The Memory Region where Data Resides

- Stack Memory Region
- Heap Memory Region

Each Thread has it's own `stack`. Stack is a memory region where:

- Methods are called.

- Arguments are passed.
- Local variables are stored.

Stack + Instruction Pointer = State of each thread's execution.

Stack's Properties:

- All varialbles belong to the thread executing on that stack.
- Statically allocated when the thread is created.
- The stack's size is fixed, and relatively small (platform specific).
- If our calling hierachy is too deep. We may get an StackOverflow Exception.(Risky with recursive calls).

Heap Memory
Heap Memory is a shared memory region belongs to the *process*.

What is allocated on the Heap?

- Objects(anything created with the new operator)
- Members of classes.
- Static Variables.

Heap Memory Management

- Governed and managed by Garbage Collector.
- Objects- stay as long as we have a reference to them.
- Members of classes - exist as long as their parent objects exist (same life cycle as their parents).
- Static variables- stay forever.

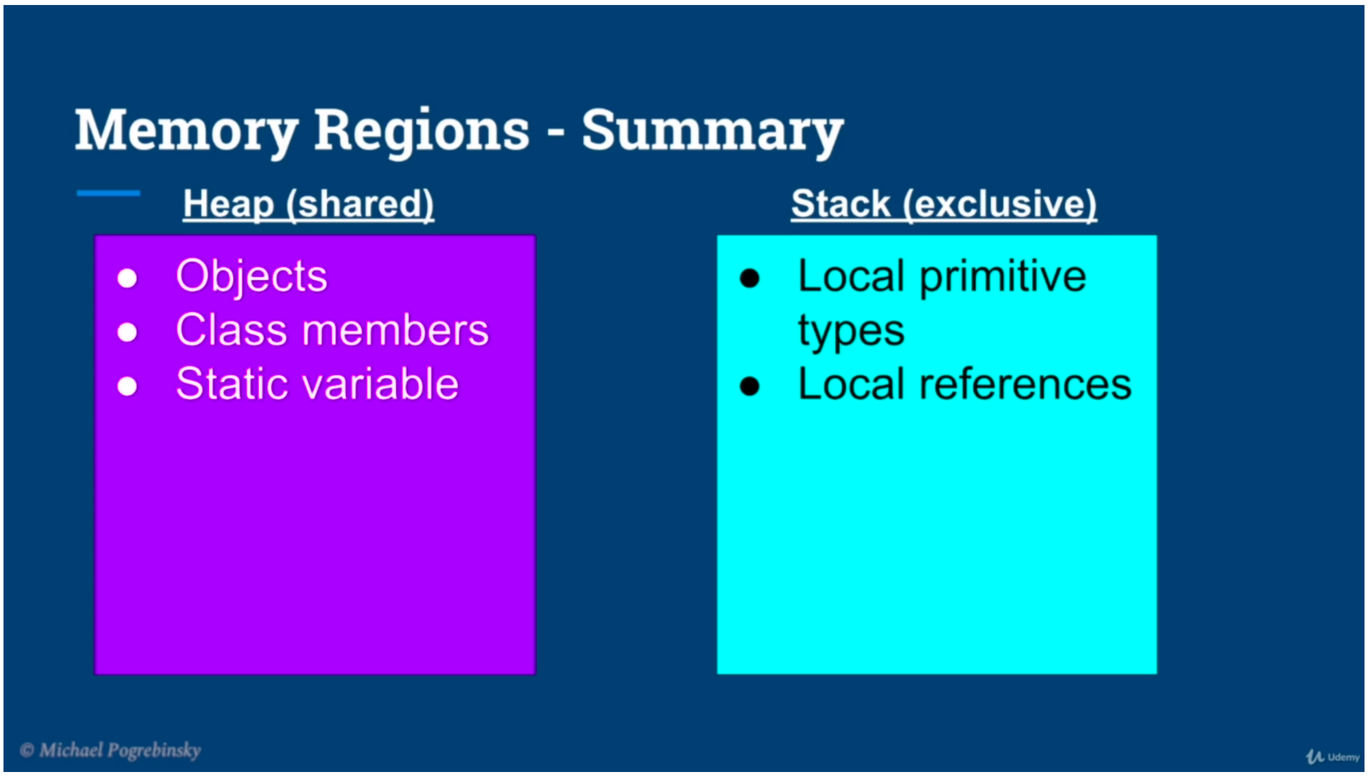
References Allocation:

- Can be allocated on the *stack*.
- Can be allocated on the *heap* if they are members of a class.

Objects Allocation:

- Always allocated on the *heap*.

Summary:



Memory Regions - Summary

Heap (shared)

- Objects
- Class members
- Static variable

Stack (exclusive)

- Local primitive types
- Local references

© Michael Pogrebinsky
Udemy

Concurrency Challenges

What is Resource?

- Variables (Integers, Strings ..)
- Data Structure
- File or connection handles
- Message or work queues
- Any Objects ...

Atomic Operation:

- **Def:** An operation or a set of operation is considered atomic, if it appears to the rest of the system as if it occurred at once.
- Single step - "all or nothing"

Synchronization in Java:

1. First way of using `synchronized` keyword:

```
    public class ClassWithCriticalSection {
        public synchronized void method1() {
            ... Thread A
        }

        public synchronized void method2() {
            ...
        }
    }
```

Only one thread will able to execute either of these two methods.
Let Thread A is using method1 then, Thread B will be deprived of using method1 as well as method2.

2. Second way of using `synchronized` keyword:

```
    public class ClassWithCriticalSection {
        Object lockingObject = new Object();
        public  void method1() {
            synchronized (lockingObject){
                ....
                critical section ....
                ....
            }
        }

        public  void method2() {
            ...
        }
    }
```

Here lockingObject can be any object. Only one thread can access a critical section locked by a single object.

Gives more flexiblity, can have multiple different critical section and multiple thread can access it.

Summary:

- We can identify the code that we need to execute **atomically** , by declaring that code as `critical section`.
- Use of synchronized keyword to protect the critical section in two ways
  - Simple way (in front of a method)
  - On an explicit Object- More flexible and granular, but also more verbose.

Solution to Concurrency

Naive Solution

- Every method which has accessed to share variable make it as synchronous method. (Excessive Synchronization)

- **Cons:** - No use of multithreading, rather it will detroiate the performance.

Atomic Operation

- All reference assignment are atomic.
- We can get and set references to objects atomically.
- All getters and setters.
- All assignments to primitive types are safe except *long and double*.

Use `volatile` keyword for long and double to make it atomic.

Race Condition:

- Condition when multiple threads are accessing a shared resource.
- At least one thread is modifying the resource.
- The timing of threads scheduling may casuse incorrect results.
- The core of the problem is non atomic operations performed.

Race Condition - Solution

- Identification of critical section where the reace condition is happening.
- Protection of the critical section by a *synchronized* block.

Data Race: Problem *May Lead to unexpected, paradoxical and incorrect results!.*

- Compiler and CPU may execute the instruction Out or Order to optimize performance and utilization.
- They will do so while maintaining the logical correctness of the code.
- Out of Order of execution by the compiler and CPU are important features to speed up the code.

Why compiler and CPU re-arranges the code: The compiler re-arranges instructions for better

- Branch predication (Optimized loops, "if" statements etc.)
- Vectorization - parallel instruction execution (SIMD)
- Prefetching instructions- better cache performance.

CPU re-arranges instructions for better hardware units utilization.

Solutions: Two ways to solve the Data Race Problem

- *Synchronization* of methods which modify shared variables.
- Declaration of shared variables with the *volatile* keyword.

Summary:

- Two problems with multithreaded applications
  - Race Conditions
  - Data Races
- Both involve
  - Multiple threads.
  - At least one is modifying a shared variable.
- Both problems may result in unexpected and incorrect results.