Multithreading, Concurrency and Parallel Performance

With Notes from Java Interview Guide : 200 + Questions

Motivation and OS Concepts

- We need thread for Responsive and Performances
  - Responsiveness – Using multiple thread (separate thread for each task) creates the illusion of executing multiple tasks – Concurrency
  - Performance – Finish more work in the same time – Parallelism
- Different from single thread programming
- If 2 threads are waiting for a thread. The scheduler picks the thread with the highest priority
- What thread are
  - The Process Contains
    - PID Mode
    - Priority
    - Files
    - Data (Heap)
    - Code
    - Main Thread (Stack, Instruction Pointer)
  - Each thread comes with its own Stack and Instruction pointer
  - Stack are region in memory where local variables are stored an passed into functions
    - Each thread is executing a different instruction in a different function at any given moment

Operating Systems Fundamentals -- Part 2

- Context Switch
  - Each instance of an application runs independently from other process
  - Each process may have 1 or more threads
  - The act of stopping a thread and scheduling the next process
  - Context is not cheap and is the price of multitasking
    - Each Thread consumes resources in the CPU and memory.
    - The data for thread must be stored and the data for current thread scheduled must be restored.
  - Too many threads causes thrashing
    - Spend more time context switching then productive work
- Threads consume less resources than processes.
  - Context switching between threads from the same process is cheaper
- Thread Scheduling
  - First Come First Serve – If a long thread arrives first then causes starvation for other threads
    - Big problem for UI Threads – less responsive
    - UI Threads are generally shorter

Operating Systems Fundamentals -- Part 2

- Shortest Job First
  - The longer tasks that involve calculations will never be executed
- Time Slices
  - Divides the time in pieces called epochs
  - Not all thread run or complete in the epoch
  - Dynamic Priority = Static Priority + Bonus
    - Bonus can be negative
    - Static priority is set by the developer programmatically
    - Bonus is adjusted by the OS in every epoch for each thread
      - The OS will give preference for interactive thread
      - The OS will give preference to thread that did not complete in the last epochs or did not get enough time to run
        - preventing starvation
- Threads vs Processes
  - Multithread
    - task must share a lot of data
    - much faster to create an destroy
    - Switching between thread of the same process is much faster
  - Processes
    - Security and stability are of higher importance – The process are completed isolated.
      - One thread can bring down a higher application
    - Task are unrelated to each other

Threads Creation – Part 1 Thread Capabilities and Debugging

- Example – Start a thread
  - public static void main(String[] args) {
    - public void run() { System.out.println(”We are in a new thread “ + Thread.getCurrentThread().getName() // code that will run in the new thread });
  - thread.start();
- Thread useful methods
  - currentThread().getName() // Gets name of thread class
  - sleep() ` // instructs the current OS to not schedule this thread until the time has elasped
  - setName(); // Sets the name of a thread
  - setPriority
  - setUncaughtException(Thread, Throwable) → Set an exception handler for an entire thread.
  - Yield() – Changes state from Running to Runnable since the Thread does not need more time
    - If there is no other thread with a higher priority then the thread might get picked again
  - ThreadLocal –
    - When we know that threads will sue the same type of variable to perform a task, but each thread will hold its own copy of the variable then either we can create a new local variable every time we create a thread
      - Each thread that is accessing its own independently initialized copy of the variable.
    - API use get/set
- Thread Debugging
  - When we hit a breakpoint all threads freeze

Threads Creation – Part 2 Thread Capabilities and Debugging

- Example
  - private static class NewThread extends Thread {
    - @override
    - public void run() { System.out.println(“Hello from “ + this.currentThread().getName(); }
    - pu0blic static void main (String args[]) { Thread thread -= new NewThread(); } }
  - This example is decoupling the code running from the thread with the thread class.
- States
  - New
  - Runnable
  - Running
  - Blocked/Waiting/Sleeping
    - Sleep is a blocking operating that keeps a hold on the monitor/lock of the shared object for a certain time
    - Used for polling or to check results at a regular interval
    - Wait pauses the thread until it is notified or the time has elapsed
    - Used with notify and notifyAll to achieve synchronization and avoid wait conditions
  - Terminated/Dead

Thread Executor Service

- ExecutorService
  - A new way of executing tasks asynchronously in the background
  - Similar to a thread pool
  - interface
  - Example – Create a single thread pool (can also have a submit function too.)
    - ExecutorService executorService = Executors.newSingleThreadExecutor();
    - ExecutorService.execute(new Runnable() { public void run () { System.out.println(“test”); } });
    - ExecuteService.shutdown().
- Different ways of creating Executor Services
  - Create one Thread
    - ExecutorService executorService = Exectuors.newSingleThreadExecutor();
    - Create a Thread pool that reuses a fixed number of threads operating off a shared unbounded queue.
      - Parameter → specifics the most thread that will be active processing tasks
      - ExecutorService executorService2 = Executors.newFixedThreadPool(10);
      - Create a thread pool that can schedule commands to run after a given delay or execute periodically
        - ExecutorService executorService3 = Executors.newShceduledTheadPool(10);

Thread Executor Service

- Executor custom implementation rather than a fixed or cached thread pool executor from a built in service
  - Fixed Thread Pool
    - Always keep a fixed number of thread evenn if they are unused.
    - At lowest traffic still have the same amount of threads
  - Cached thread pool
    - Starts with zero threads and can to up to Integer.MAX_VALUE
    - If the app demands low latency and is highly loaded then this may cause us to run out of memory increasing latency as the OS Starts Paging
  - Custom Thread Pool
    - Can define the number of initial threads and the number of maximum threads
    - Even the time to for a thread being reused can be controlled
    - Better for the owner since they know the ins and outs of their architecture

Thread Termination & Daemon Thread

- Thread Termination
  - Thread consume resources (memory and kernel resource even when the thread is not running
  - If the thread is misbehaving we want to stop it.
  - Want to stop or close the entire application
    - The application will not stop as long as one thread is running
- Thread.interrupted
  - Each Thread Object has a member function called interrupt
  - Example
    - Currently in Thread A
    - In threadB threadA.interrupt() → Sends a signal from Thread A to interrupt Thread B
  - Use Cases
    - If thread is executing a method that throws an InterruptedException
    - If the Thread’s code is handling the interrupt signal explicitly

Thread Termination & Daemon Thread

- Example
    - public class Main {
        - public static void main(String[] args) {
            - Thread thread = new Thread(new BlockingTask());
            - thread.start();
        - }
        - private static class BlockingTask implements {
            - public static void run() {
                - try {
                    - Thread.sleep(100000);   // Throws an InterruptedException when thread is interrupted externally
                - } catch(InterruptedException e) {
                    - System.out.println(“Exiting block thread”);
                - }
            - }
        - }
    - }
    - Result : The Main Thread is long gone, but we are waiting on the BlockingTask to finish
-

Thread Termination & Daemon Thread

- Example
- public class Main {
  - public static void main(String[] args) {
    - Thread thread = new Thread(new BlockingTask());
    - thread.start();
    - thread.interrupt();   // An interrupted exception is thrown and the applications ends before the BlockingTask is fully executed
  - }
  - private static class BlockingTask implements Runnable {
    - public static void run() {
      - try {
        - Thread.sleep(100000);   // Throws an InterruptedException when thread is interrupted externally
      - } catch(InterruptedException e) {
        - System.out.println(“Exiting block thread);
      - }
    - }
  - }
- }
- Result

Thread Termination & Daemon Thread

- Example
  - public class Main {
    - public static void main(String[] args) {
      - Thread thread - new Thread(new LongComputationTask(new BigInteger(“20000”), new BigInteger(“1000000000”);
      - thread.interrupt();   // Just calling interrupt is not enough when we don’t have a method to handle it by checking for isInterrupted
    - }
    - private static class LongComputationTask implements Runnable {
      - private BigInteger   base;   private BigInteger   power;
      - public LongCompuationTask(BigInteger base, BigInteger power) { this.base = base; this.power = power; }
      - public static void run() { System.out.println(“base + “^” + power + “ = “ + pow(base,power);
    - }
    - private BigInteger pow(BigInteger base, BigInteger power) {
      - BigInteger result = BigInteger.ONE;
      - for (BigInteger I = BigInteger.ZERO; I.compareTo(power) != 0 ; I = I.add(BigInteger.ONE)) {
        - if (Thread.currentThread().isInteruppted()) {   // Checks if there is an interrupt
          - System.out.println(“Prematurely Interrupted”);
          - return BigInteger.ZERO;
          - result = result.multiply(base);
      - }
      - return result;
    - } } }

Thread Termination & Daemon Thread

- Daemon Threads
  - Background thread that do not prevent the application from exiting if the main thread terminates
  - Use Cases
    - Background Tasks that should not block our application from terminating
      - File save thread in a Text Editor
    - Code in a worker thread is not under our control and we do not want it to block our application from terminating
      - Worker thread that uses an external library
  - thread.setDaemon(true) – The thread will not stop the application from terminating
- From https://www.baeldung.com/java-daemon-thread
  - Two Type of threads
    - User threads – High Priority and the JVM will wait for any user to complete its task before terminating it
    - daemon threads are low priority threads whose only role is to provide services to user
      - won’t prevent the app from exiting .once the user threads have finished
      - good place to have infinite loops
      - thread.join() could block the shutdown.
      - Use in JVM – releasing memory of unused object / removing unwanted entries from the cache

Joining Threads

- Thread Coordination
  - Different threads run independently
    - Do not rely on the order of execution
  - Order of execution is out of our control
  - What if one thread depends on another thread
- Busy Wait → When a thread checks (burn cpu cycles) another if another thread completed its work.
  - Inefficient and wasteful → The calling thread is wasting CPU cycles checking the thread
- Thread B goes to sleep and Thread A finishes it works and then Thread B will wake up and take the fully completed result
- join() – calling thread goes into a waiting state until all referenced threads terminates
  -

Joining Threads

- public static void main(String[] args) throws InterruptedException {
- List<Long> inputNumbers = Arrays.asList(100000000L, 3435L, 35435L, 2324L, 4656L, 23L, 5556L);
- List<FactorialThread> threads = new ArrayList<>();
- for (long inputNumber : inputNumbers) { threads.add(new FactorialThread(inputNumber)); }
- for (Thread thread : threads) { thread.setDaemon(true); thread.start(); }
- for (Thread thread : threads) { thread.join(2000); }
  - **// Race Condition – Without the join the Main Thread and the factorial thread the code below could not be created before the next section of code**
- for (int i = 0; i < inputNumbers.size(); i++) {
- FactorialThread factorialThread = threads.get(i);
- if (factorialThread.isFinished()) { System.out.println("Factorial of " + inputNumbers.get(i) + " is " + factorialThread.getResult()); } else {
- System.out.println("The calculation for " + inputNumbers.get(i) + " is still in progress"); } }

Joining Threads

- What if one number is very large
  - Decision – How long are we willing to wait.
  - Execution : thread.join(2000);   // if the thread has terminated in two seconds the join will return
    - The created thread is still running
    - can put a Thread.setDaemon(true) before the join so the application can terminate

Introduction to Performance & Optimizing for Latency

- How can we reduce latency with multithread programming
  - Break a task down into subtasks that can executed independently on different threads
  - Want to achieve Latency = T/N where T is the number of Tasks
    - Theoretical reduction of latency by N = Performance by a factor of N
  - What is N
    - How many subtasks/threads to break the original task
    - On a general computer N should be as close as possible to the number of core
      - reduction in latency in running these subtasks as parallel
      - OS will do its best to put each task on a separate core
        - More thread than core will increase the latency since it will keep increase context switching
      - # threads = #cores is optimal only if ll threads are runnable and can run without interruption (no IO/ blocking calls / sleep)
        - very rarely the case, but can get close
        - The assumption is nothing else is running that consume a lot of CPU
          - Can run it on its own server
      - Most computer have hyperthreading (some hardware units duplicated and some hardware units are shared)
  -

Introduction to Performance & Optimizing for Latency

- Inherent cost of parallelization and Aggregation
  - cost : Breaking task into multiple task
  - cost : Thread creation and passing tasks to threads
  - cost : Time between thread.start() to thread getting scheduled
  - cost : time to last thread finishes and signals
  - cost : time until the aggregating thread runs
  - cost : Aggregation of the subresults into a single artifact
  - Take away : Small and trivial task are not worth making concurrent and making parallel

Introduction to Performance & Optimizing for Latency

- Performance
  - Ex High Speed Trading Systems
    - Measure using latency
    - The faster the transaction the more performance the application is considered to be
    - measured in units of time
  - Ex. Video Play
    - Want to show the user the video at the correct frame rate with very little or no jitter
    - precision and accuracy of the frame rate
  - Ex Machine Learning
    - Provides machine learning on a large volume of data and provides a prediction
    - The metric is the throughput.
- Performance in Multithreading – In general we have two measures
  - Latency – The time to complete a task measured in time units
  - Throughput – The amount of task completed in a given period measured in tasks/time unit
  - Improving latency or throughput could have positive/negative/neutral effect on the other

Optimizing for Latency

- Some image processing notes
  - Create RGB
    - rgb |= blue;
    - rgb |= green << 8
  - Determine grey (red - green), (red - blue), (green - blue) must be less then 30
- Mutlithreaded Solution
  - Break the image into an equivalent amount of pieces according to the number of core in the cpu
    - If have a quadcore (hyperthread) then break it into 8 threads and each thread will process it portion
- Performance
  - With Single Threads : When ran on the original data the single thread solution was 1.2 seconds and the mutlithreaded solution was 1.3
  - With Two Threads : When ran on the original data the single thread solution was 1.2 seconds and the multithred solution was 731 milliseconds
  - With four threads: When ran on the original data the single thread solution was 1.2 seconds and the multithread solution was 473 milliseconds
- As we increased the number of core (Example Quadcore, but not virtual thread (so 4 and not 8)) there were noticeable gains for each thread added
  - More threads than cores is counterproductive

Optimizing for Latency

- As we increased from the core count to the count of hyperthreads the effect was less noticeable and started to become counter productive then single threaded
  - pair of virtual cores share resources among themselves
  - The computer was not fully dedicated to the application
  - Number of threads beyond virtual cores → There was none
- Resize the photo
  - As it got smaller threads became less of an issue and they decreased performance
- By serving each task on a different thread, in parallel, we can improve throughput by N

Optimizing for throughput

- Approach 1
  - If all Tasks takes time t to computer then one task takes 1/t
  - If the system can be broken down the Latency become T / N (where t is number of tasks and n is number of threads)
  - Run the subtask in parallel
    - Throughput is N / T
      - where T is time to execute original task
      - N - # subtasks / # threads
    - The through put will be less then the theoretical because o the same issues mentioned in optimizing for latency
  - Inherent cost of parallelization and Aggregation
    - Break task into multiple tasks
    - Thread creation, passing tasks into threads
    - Time between thread.start()to thread getting scheduled
    - Time until last thread finishes and signals
    - Time until aggregating thread runs
    - Aggregation of the sub-results into a single artifact
    - For throughput combing them into a single result is not needed

Optimizing for throughput

- Approach 2 – Schedule running tasks in parallel
  - Schedule each task in a separate thread
  - In practice more likely to achieve theoretical throughput then reduce the latency
  - The task are inherently unrelated and independent from each other
    - From the Inherent cost of parallelization and Aggregation
    - Don’t need to break task into multiple task
    - Don’t need the aggregating thread
    - Don’t need to combine the subresults into a result
      - Have only one result for each task
  - We don’t need to wait for one task to complete the other
    - Can even minimize contributors to the cost to achieve the optimal throughput.
- Threadpooling
  - Use a Queue to store the treads and each thread is taking tasks whenever that thread is available
  - If we keep the threads and utilized we can get the maximum throughput and utilization
  - An example would be the Executor from Java

Optimizing for throughput

- com.sun.net.httpserver.HttpServer
  - create → creates the server with an ipAddress
  - createContext → Add the path that that will allow use to get to the application
    - handles each request and sends a response
  - Can add a set of worker threads (Executor thread pool) to manipulate the data
- Apache Jmeter – Java performance automation tool that does not require writing any code
  - Test Plan – Load a file which contain a list of words. For each word send an HTTP Request and wait for an response
- For throughput – improve all the way to the number of virtual cores since we do not have to break the data into smaller chunks which takes time.

Optimizing for throughput Example

```
/**
 * Optimizing for Throughput Part 2 - HTTP server + Jmeter
 * https://www.udemy.com/java-multithreading-concurrency-performance-optimization
 */
public class ThroughputHttpServer {
    private static final String INPUT_FILE = "./resources/war_and_peace.txt";
    private static final int NUMBER_OF_THREADS = 8;

    public static void main(String[] args) throws IOException {
        String text = new String(Files.readAllBytes(Paths.get(INPUT_FILE)));
        startServer(text);
    }

    public static void startServer(String text) throws IOException {
        HttpServer server = HttpServer.create(new InetSocketAddress(8000), 0);
        server.createContext("/search", new WordCountHandler(text));
        Executor executor = Executors.newFixedThreadPool(NUMBER_OF_THREADS);
        server.setExecutor(executor);
        server.start();
    }

    private static class WordCountHandler implements HttpHandler {
        private String text;

        public WordCountHandler(String text) {
            this.text = text;
        }

```

Optimizing for throughput Example

```
•    @Override
•     public void handle(HttpExchange httpExchange) throws IOException {
•         String query = httpExchange.getRequestURI().getQuery();  String[] keyValue = query.split("=");
•         String action = keyValue[0]; String word = keyValue[1];
•         if (!action.equals("word")) { httpExchange.sendResponseHeaders(400, 0); return’ }
•
•         long count = countWord(word);
•
•         byte[] response = Long.toString(count).getBytes();
•         httpExchange.sendResponseHeaders(200, response.length); OutputStream outputStream = httpExchange.getResponseBody();
•         outputStream.write(response); outputStream.close();
•     }
•
•     private long countWord(String word) {
•         long count = 0; int index = 0;
•         while (index >= 0) {
•             index = text.indexOf(word, index);
•
•             if (index >= 0) { count++; index++;  }
•         }
•         return count;
•     } } }
```

Stack and Heap Memory Regions

- Heap
  - A shared memory region that belongs to the process
  - All processes can access/allocated object
    - Object (anything with the new operator)
    - Members of classes)
    - static variables (member of class object assicated with that object)
- Heap Memory Management
  - Governed and managed by Garbage Collector
  - Objects – stay as long as we have a reference to them.
  - Member of classes - exist as long as their parent object exist (same lifecycle as their parents)
  - static variables stay for the lifetime of the class
- Reference declared as local reference are on the stack
- Reference declared in a class are on the heap.

Resource Sharing Between Threads

- What is a resource
  - Variables, Data Structure File or Connection handles, Message or Work Queues, Any object that we create
- What resource can threads share inside a process
  - Heap (shared)
    - Objects
    - Class members
    - static variables
- Why would we want to share resources ?
  - Example Text Editor (an UI Thread, Document Server)
  - Work Queue
    - A dispatcher threads put the threads on a shared queue and uses a small pool of worker threads
    - Leads to low latency since we don not create a new thread
  - A shared connection to a microservice
- What is the problem with sharing resources
  - Variables being changed are shared objects and operations are happening at the same time to them.
  - Those operations are not atomic.

Resource Sharing Between Threads

- Atomic Operations
  - An operation that appears to the rest of the system as it occurred at once.
  - No Intermediate States
- Cannot be interrupted
  - Java Provides AtomicOperations
    - AtomicInteger
    - Increment and get

Critical Section and Synchronization

- Concurrency Problem
  - Perform a set of operation in a such a way that they will appear as an atomic operation and no two threads can be performing those operations simultaneously
  - Known as the critical section
    - if no threads are in the critical section the thread can enter
    - If a thread is in the critical section then another thread tries to enter the critical section in will be suspended until the original thread exits the critical section
- Synchronized Keyword
  - Locking mechanism
  - Used to restrict access to a critical section or entire method to a single thread at a time
  - Can be used on a method or as block of code
  - Monitor
    - Example
      - public class ClassWithCriticalSections() { // When one thread is executing a method another thread cannot execute either methods
        - public synchronized method1() {}
        - public synchronized mehtod2() {} }
    - Synchronized is applied per object largely equivalent to using synchronized(this)
      - Another term is monitor
  - Can static method be synchronized – yes – I believe that the class cannot be used as a lock.

Stack and Heap Memory Regions

- Stack
  - A memory region where methods are executed
  - When arguments are passed they are put on the stack
  - When local variables are stored they are put on the stack.
- Remember : Each thread is executing a different line of code.   The stack and the instruction pointer is the overall state of each thread’s execution
- Stack Frame – Space allocated when a new method is entered
  - A new stack frame is created when a new method is entered
  - Each method has access to variables in its own stream
- Properties
  - All variables belong to the thread executing on that stack and no other threads have access to them.
  - Statically allocated when thread is created and the size is fixed an relatively small
  - If our calling hierarchy is to deep. A stack overflow exception may get thrown

Critical Section and Synchronization

- Example
  - public class ClassWithCriticalSections {
    - Object lockingObject = new Object(); // Serves as lock
    - Oblect lockingObject2 = new Object();
    - public void mehtod1() {
      - synchronized(lockingObject) {
        - …
        - critical section
        - … }}
      - synchronized(lockingObject2) {
        - …
        - critical section
        - … }}
  - More code can be executed without synchronization by using blocks instead of classes or methods
- Synchronized block is Reentrant
  - If thread A is currently in a synchronized method then it cannot
  - A thread cannot prevent itself from entering a critical section

Critical Section and Synchronization

- Example
  - public static void main(String[] args) {
    - SharedClass sharedClass1 = new SharedClass(); SharedClass sharedClass2 = new SharedClass();
    - Thread thread1 = new Thread(() → { while (true) { shraedObject1.increment() } }); Thread thread2 = new Thread(() → { while (true) { shraedObject1.increment() } });
    - Thread1.start(); Thread2.start();
    - public static class SharedClass {
      - private int counter = 0;
      - public synchronized void increment() { this.counter++; }
  - }
  - Outcome: When thread1 is executing sharedObject1.increment(); thread2 can execute execute sharedObjet2.increment() since synchronization happens on the object level thread1 and thread2 are operating on two different, independent objects.
  - Extra information : Synchronization is happening on an object level and not Class level. Since thread1 and thread2 are executing methods from different instances they do not block each other
- Synchronization and Visibility
  - Synchronized key enforces a mutex that prevents more than one thread at a time from entering a synchronized block
  - Enforces certain memory and visibility rules
    - Caches are flushed when exiting a synchronized block and invalidate when entering one
    - Compiler does not move instructions outside the synchronized block
- Volatile goes directly to memory for read and writes

Critical Section and Synchronization

- Notes on volatile
  - You can use a volatile variable if you want to read and write long and double variable automatically.
  - It can be used as an alternative way of achieving synchronization in Java.
  - All reader threads will see the updated value of the volatile variable after completing the write operation. If you are not using the volatile keyword, different reader thread may see different values.
  - It is used to inform the compiler that multiple threads will access a particular statement. It prevents the compiler from doing any reordering or any optimization.
  - If you do not use volatile variable compiler can reorder the code, free to write in cache value of volatile variable instead of reading from the main memory.
- Interesting Points
  - You can use the volatile keyword with variables. Using volatile keyword with classes and methods is illegal.
  - It guarantees that value of the volatile variable will always be read from the main memory, not from the local thread cache.
  - If you declared variable as volatile, Read and Writes are atomic
  - It reduces the risk of memory consistency error.
  - Any write to volatile variable in Java establishes a happen before the relationship with successive reads of that same variable.
  - The volatile variables are always visible to other threads.
  - The volatile variable that is an object reference may be null.
  - When a variable is not shared between multiple threads, you do not need to use the volatile keyword with that variab

Critical Section and Synchronization

- If you start two threads and they each execute without synchronizing on any common monitors or volatile you can predict nothing about the relative order in which actions in one thread will execute with respect to actions in the other thread
- Ordering guarantees
  - Each action in a thread happens before every action in that thread that comes later in the program order.
  - An unlock on a monitor happens-before every subsequent lock of that same volatile
  - A write to a volatile field happens before every subsequent read of that same volatile
  - A call to Thread.start on a thread happens before any actions are started
  - All actions in a thread happen before any other thread successfully returns from a Thread.join() on that thread.

Atomic Operations, Volatile and Metrics practical example

- Making all methods synchronized reduces the speed of the application.
  - No parallel execution without multiple threads
- Atomic Operations
  - All reference assignment are atomic
    - We can get and set references atomically
    - Example; Object a = new Object(); Object b = new Object(); a = b; // atomic
  - All assignments to primitive types are safe except long and double
    - Solution : Volatile read and write to long and double are threadsafe (guaranteed to be atomic)
    - Volatile → A declaration that a variable can be accessed by multiple threads and should not be cached
- Data being used by thread, but not changed does not need to be synchronized

Race Conditions and Data Races

- Race Condition
  - Condition when multiple threads are accessing a shared resource
  - At least one thread is modifying a resource and the time of threads scheduling may cause incorrect results
  - The core of the problem is non atomic operations performed on a resource
  - Solution
    - Identification of the critical section where the race condition is happening
    - Protection of the critical section by a synchronized block
- Data Race
  - Compiler or CPU may execute the instructions out of order to optimize performance and utilization
    - It is an important feature to speed the code up
    - The compiler re-arranges instructions for better
      - Optimized loops, “If” Statement
      - Vectorization – parallel instruction execution (SIMD)
      - Prefetching instructions – better cache performance
    - CPU re-arranges for better hardware units utilization
    - May lead to a paradoxical/incorrect result

Race Conditions and Data Races

- Solution
  - Establish a Happens Before semantics – When it come to operations executed by different threads concurrently excepted a few cases
    - Synchronization of methods which modify share variables
    - Declaration of shared variables with the volatile keyword
      - reduce overhead of locking and guarantee order
      - All instructions before the volatile will be executed before
      - All instructions after the volatile will be executed after
      - Equivalent to a memory fence or barrier
- As a rule of thumb every shared variable (modified by at least one thread should be either)
  - Guarded by a synchronized block (or any type of lock)
  - Declared Volatile
    - When one thread can write to a variable and all others read from it then volatile guarantees that all threads read the value of the variable will the latest value
    - Every thread accessing a volatile field will read the variable’s current value instead of using a cached value
    - Example multiple Change Listeners for a variable and only one thread performing the write action

Locking Strategies and Deadlocks

- Fine Grained Locking vs Coarse-Grained Locking
  - Another way of stating the above : Should we have one single lock (Coarse Grained on all the shared resource or a separate lock for each resource (Fine Grained Resources)
  - Coarse Grained
    - Advantage : Only a single lock to worry about
    - Implement by using a synchronized keyword for all the methods containing the shared resources
    - Simple to maintain, but overkill
    - Worse Case Scenario → Only one thread at a time could access those resources (rare threads are doing other things)
  - Fine Grained
    - Lock on every resource individually
    - Allows for more parallelism and less contention
    - Problem: Deadlock
      - A circular dependency
      - Where Thread A is using resource B and needs resource C and Thread B is using resource C and needs resource B
      - A circular dependency
  -

Locks Overview

- Lock
  - Used to divide methods into different blocks
  - 10 methods can be divided into different blocks which can be synchronized on different values
  - Allows for more parallel processing
  - For the 10 methods they may not need to be synchronized at the same time
- Example
  - Final transient ReentrantLock lock = new ReentrantLock();
-
  - // In different code
  - Public E set(int index, E Element) {
  - Final ReentrantLock lock = this.lock;
  - Lock.lock();
  - Try { Object[] elements = getArray }
  - Finally { Lock.unlock(); }

Locking Strategies and Deadlocks

- Deadlock Conditions
  - Mutual Exclusion – Only one thread can have exclusive access to a resource
  - Hold and Wait – At least one thread is holding a resource and is waiting for another resource
  - Non Preemptive allocation – A resource is released only after the thread is done using it.
  - Circular Wait – A chain of at least two thread each one is holding one resource and waiting for another resource
- Solutions to the deadlock problem
  - Make sure one the Deadlock conditions is not met.
  - Easier way to avoid a deadlock is to avoid “Circular Wait”
    - Enforce a strict order in lock acquisition
    - Acquire the locks on the resource on the same order and stick to that order for each critical section
      - Well never have a deadlock
    - Seem to be the best solution
  -

Locking Strategies and Deadlocks

- The “Circular Wait” solution
  - Hard to accomplish if there are many locks in different places
  - Other Solutions
    - Deadlock detection – Watchdog
      - microcontroller → A routine periodically checks the status of a register which is update by the thread every few instructions.
        - If the register has not been set in a time interval then threads will be restarted
      - Thread interruption and tryLock operations not possible with synchronized
        - tryLock → Checks if a lock is acquired by another thread

ReentrantLock

- Reentrant Lock
  - Works just like the synchronized keyword applied on an object
  - Requires explicit locking and unlocking
  - Example
    - Lock lockObject = new ReentrantLock();
    - Resource resource = new Resource();
    - public void method() {
      - lockObject.lock();   // protect a resource from concurrent access
      - use(resources);
      - lockObject.unlock();
  - Disadvantage : Leave the lock locked forever (bugs and deadlocks)
    - Problem – Exceptions can get thrown when an object is locked or a return statement can be called.
      - Make sure the critical section is in a try block and unlock is put into a finally
    - Advantageous
      - rewarded with more control over the lock
      - More Lock operations

ReentrantLock

- Operations good for testing
  - getQueuedThreads() – Returns a list of threads waiting to acquire a lock
  - getOwner – Returns the thread that currently owns the lock
  - isHeldByCurrentThread() – Quereis if the lock is held by the current Thread
  - isLocked() – Queries if the lock is held by an thread
- Operations for fairness
  - Another area where the ReentrantLock shines is the control over lock's fairness
  - By default, the ReentrantLock as well as synchronized key do not guarantee any fairness
    - A thread may be starved for time
  - ReentrantLock(true) – Guarantees fairness, may reduce throughput of the application since acquisition of the lock may take longer
- RenentrantLock.lockInterruptibly
  - Useful
    - very used if implement Watchdog for deadlock detecting and recovery
    - Walking up threads to do clean and close the application
  - Example
    - Another thread throws someThread.interrupt();
    - @Override
    - public void run() {
      - ReentrantLock lockObject = new ReentrantLock();
      - while (true) {
        - try {
          - lockObject.lockInterruptibly(); // When an interrupt is called then it wakes up and jumps to the catch block
        - } catch(InterruptedException exception) {
          - cleanUpAndExit();
        - } finally {
          - lockObject.unlock();
        - } } }

ReentrantLock

- ReentrantLock.tryLock()
  - boolean tryLock()
  - boolean tryLock(long timeout, TimeUnit unit)
    - Returns true and acquires the lock if available
    - Returns false and does not get suspended, if the lock is unavailable.
      - Instead of blocking
  - Example
    - if (lockObject.tryLock()) { // Acquires the lock
      - try {
        - useResource();
      - finally {
        - lockObject.unlock();
      - } }
  - Difference between lock and tryLock for the lock the thread is block, but with tryLock the thread does not block and returns immediately
  - Real time applications where suspending a thread on a lock() method is unacceptable.
    - Such as video/Image Processing or High Speed/Low Latency Trading Systems or User Interface Applications
    - Need to share a resource with other threads

Reentrant Lock Example

- The application will have two thread
  - A GUI Thread – animation in back, mouse envent and display prices
    - Running on a single thread.
  - A background thread – Connect to exchange and get the data
  - A shared resource that contains the investment (stock, cryptocurrency) and the price
    - Two threads are going to access this structure we need a lock
    - Want to see all price updated at once, not individually
    - Went with trylock() to make the UI snappy
      - Avoid blocking the real time thread
      - Kept application responsive
      - Performed operations atomically and avoid the race condition

Reentrant Read Write Lock and Database Implementation

- Race Conditions
  - Happens when multiple threads are sharing a resource and at least one is modifying a resource
  - Solution: Regardless of Operation(read/write/both) locks and allow only one thread into a critical section
- Workload involving mostly reading and very little writing
  - Multiple thread can read from a resource so long as they are not modifying the state in any way.
  - Synchronized and ReentrantLock do not allow multiple reads to access a shared resource concurrently
  - In general not a big problem in the general case
    - Keep critical section short the chances of a contention over a lock are minimal
  - Where read operations are predominant or when read operations are not as fast read from many variables, read from complex data structure
    - Mutual exclusion of reading threads negatively impacts the performance.
  - This is where reentrant Read Write Locks come in handy
  - When there are more write the Reentrant ReadWrite Lock can actually perform worse.

Reentrant Read Write Lock and Database Implementation

- Example
  - ReentrantReadWriteLock rwLock = new ReentrantReadWriteLock(); // only provides query methods and 2 internal locks
  - Lock readLock = rwLock.readLock();
  - Lock writeLock = rwLock.writeLock();
  - writeLock.lock();
  - try {
    - modifySharedResources();
  - } finally {
    - writeLock.unlock();
  - }

```
readLock.lock();
try {
        readFromSharedResources();
} finally {
        readLock.unlock();
}
```

- Multiple Threads can acquire the readLock simultaneously, but a single can acquire the writeBlock and all other threads accessing the block will be blocked
- Mutual exclusion between reads and writers
  - If a write block is acquired then no thread can acquire a read lock
  - If at one thread holds a read lock then no thread can acquire a write lock
- In the demo the ReentrantLock took 3000 ms and the Reentrant Read Write Lock took 1400 milliseconds

Semaphore – Scalable producer/consumer implementation

- Can be used to restrict the any number of users to a particular resources or a group of resources.
  - Example Parking Lot
- Example
  - Semaphore semaphore = new Semaphore(NUMBER_OF_PERMITS);
  - semaphore.acquire();   // NUMBER_OF_PERMITS -1 now available   // Can acquire more permits (ex. semaphore.acquire(5)) – 5 permits retrieved
  - useResource();
  - semaphore.release();   // NUMBER_OF_PERMITS +1 now available   // Can release more permits (ex. semaphore.release(5)) – 5 permits released
- If the thread request more permits than the semaphore has access to then the thread is block until permits are retrieved
- Semaphore vs Locks
  - A lock is a semaphore with one permit to give
  -

Semaphore – Scalable producer/consumer implementation

- Not a great choice for a lock
  - Semaphore does not have the notion of an owner thread
    - Semaphore can be release by any thread even can be release by a thread that has not actually acquired it.
    - Could create a situation where a thread in the critical zone accidentally release the semaphore and thread acquires it and enter the critical zone.
  - Many threads can acquire a permit
  - The same thread can acquire the semaphore multiple times
  - The binary semaphore (initialized with 1) is not reentrant
    - Example – if the thread get the semaphore and acquires it again and the semaphore has no more permits then the thread is blocked
- Producer Consumer Relationships
  - Semaphore full = new Semaphore(0);
  - Semaphore empty = new Sempahore(1);
  - item = null;

```
Producer                        Consumer
while(true) {                   while(true) {
    empty.acquire();                full.acquire();
    item = produceNewItem();        consume(item);
    full.release();                 empty.release();
}                               }
```

  - If the producer is slower than than the consumer then it would spend most of the time in suspended mode and will not consume any CPU
  - If the consumer is slower than the producer the producer is guaranteed that the producer will not produce more items until the consumer is done

Semaphore – Scalable producer/consumer implmentation

- Producer Consumer Relationships with many producers and consumers

• CAPACITY = 5	// Number of elements to produce at conce
• Semaphore full = new Semaphore(0);	
• Semaphore empty = new Semaphore(CAPACITY);	// The capacity allows us to
• Lock lock = new ReentrantLock();	// Protect the queue from access of multiple producers or consumers
• Queue queue = new ArrayDequeue();	
• Producer	Consumer
• while(true) {	while(true) {
• empty.acquire();	full.acquire();
• lock.lock();	lock.lock();
• queue.offer(item)	Item item = queue.poll();
• lock.unlock();	lock.unlock();
•	consume(item):
• full.release();	empty.release();
• }	}

Condition Variables – All purpose inter-thread Communication

- Example of Thread Communication
  - Thread A interupts Thread B
  - join() – One thread wakes up another
  - semaphore release
- Interthread Sempaphore as Condition var
  - Calling the acquire() on a Semaphore is equivalent to checking the condition “Is Number of permits > 0 “
  - If the condition is not met – Thread A goes to sleep until another thread changes the semaphore’s starte
  - When thread B calls the release() method, Thread A wake up
  - Thread A checks the condition “Is Number of Permits > 0”
    - If it is, Thread A continues to the next instruction
- Inter-thread Condition Variable
  - Check condition
  - Condition is not met then Thread A is suspended
  - Thread B changes state and Signals Thread A to wake up
  - Thread A checks condition and is Satisfied

Condition Variables – All purpose inter-thread Communication

- Condition variable is always associated with a lock
  - lock ensures atomic check and modification of the shared variables involved in the condition
- Example
  - Lock lock = new ReentrantLock();
  - Condition condition = lock.newCondition();
- Example – Producer, Consumer
  - Example – A GUI Thread to get the username/password and a Thread to lookup the username and password in the db to see if they match
  - Code
    - Lock lock = new ReentrantLock();
    - Condition condition = lock.newCondition()
    - String username = null, password = null;
    - Thread
      - lock.lock()   // Acquire the lock on the two shared variables
      - try (while(username == null) {
        - if (password == null) {
          - condition.await()   // condition is not met then unlocks lock and puts the thread asleep
          - }
        - {
        - finally { lock.unlock();
        - }
      - doStuff();

Condition Variables – All purpose inter-thread Communication

- Gui Thread
  - lock.lock();
  - try {
    - username = userTextbox.getText();
    - pasword = passwordText.getText();
    - condition.signal()
  - } finally {
    - lock.unlock();
  - }
- void Condition.await() – unlocks lock, wait until signaled
- void Condition.signal – wakes up a single thread, waiting on the condition variable
  - A thread that wakes up has to a reacquire the lock associated with the condition variable.
  - if no thread is waiting on the condition variable the signal method does not do anything
    - The signal is not saved anywhere meaning a future variable will not know if it is unlock

- Other forms of the await method
  - void await() – unlock lock, wait until signalled
  - long awaitNanos(long nanosTimeout) – wait no long than nanosTimeout
  - boolean await(long time, TimeUNit unit) – wait no long than time in the given time units
  - boolean awaitUnti(Date deadline) – Wakeup before the deadline data
-

Condition Variables – All purpose inter-thread Communication

- Condition.signleAll() – Broadcast a signal to all the threads currently waiting on the condition variable
  - Does not need to know home many (if at all) threads are waiting on the condition variables

Objects as Condition Variables wait(), notify() or notifyAll()

- Object Class contains the following methods
  - wait() throw Interrupted Exception
  - notify
  - notifyAll
- We can use any object as a condition variable smf lock (using the synchronized keyword)
  - We can use any object for interthread communication
- wait() – Cause the current thread to wait until another thread wakes it up
  - In the wait state the thread is not consuming any CP
  - 2 Way to wake up a thread
    - notify-- Wakes up- a single thread waiting on that object
    - notifyAll – Wakes up all the threads waiting on the object
- To call wait(), notify() or notifyAll() we need to acquire the monitor of that object (use synchronized on that object)
  -

Objects as Condition Variables wait(), notify() or notifyAll()

- Example
  - public class mySharedClass { Multiple tasks are sharing the object
    - private boolean isComplete = false;
    - public void waitUnitComplete() {
      - synchronized(this)
        - while (isComplete == false) {
          - this.wait();
        - }
      - }
    - }
  - }
  - public void complete () { Synchronized(this); isComplete = true; this.notify(); } }
- Similar to conditions variables
- When the object is the current object the synchronized keyword can be moved to the declaration
- Backpressure to guard against OutOfMemory Exception
  - Use a queue to decouple multithreaded components and apply backpressure to limit the size of the queue
  - Has a wait synchronized block that check to see if the queue reach capacity. If it did wait until the consumer wake the Queue up.by using notify

Introduction to Lock-free Algorithms Data Structures and Techniques

- What wrong with locks
  - Majority of multi-threaded programming is still done with locks
  - Most of the concurrency problems are easier and safer to solve with locks.
  - Locks have been amount for a long time. They have great software and hardware support
  - Using locks we can solve all concurrency issues.
  - Issues
    - Deadlocks
      - Generally unrecoverable unless you took the time and effort to implement detect and resolution logic
      - Can bring your application to a complete halt
      - The more locks in the application,. the higher the changes for a deadlock
    - Multiple threads using the same lock
      - One thread can hold the lock for a very long time
      - Slow down all the other threads
      - All the threads become slower than the slowest thread
    -

Introduction to Lock-free Algorithms Data Structures and Techniques

- Priority Inversion
  - Two thread sharing the same lock (resource).
  - Low priority thread (document save) and the User Interface will have higher priority thread
  - Low Priority Thread acquires the lock and is preempted (scheduled out)
  - High Priority Thread cannot progress because of the low priority thread is not scheduled to release the lock
- Thread not releasing a lock (Kill Tolerance)
  - Thread dies, gets interrupted or forgets to release a lock
  - Leaves all threads hanging forever
  - Unrecoverable, just like a deadlock
  - To avoid, developers, developers need to write more complex code (Need to wrap every critical section with a try catch block and a finally)
- Performance
  - Thread A acquired a lock and Thread B tries to acquire a lock and gets blocked and Thread B is scheduled out (context switch)
    - Problem is an extra context switch and bring the thread back when the lock is released.
    - Additional overhead may not be noticeable for most applications
    - For Latency sensitive applications, this overhead can be significant
-

Introduction to Lock-free Algorithms Data Structures and Techniques

- Why did we need locks
  - Multiple thread accessing sheared resources
  - At least on thread is modifying the shared resources
  - Non atomic operations
  - How did we end up with non atomic operations
    - A single Java Operation turns into one or more hardware operations
    - Example counter++ turns into 3 hardware instructions (read, calculate, write)
      - Another Thread can modify count between execution of the instructions
- Lock Free Solution
  - Utilize operations which are guaranteed to be one hardware operation
  - A singe hardware instruction is atomic by definition and is thread safe
- Atomic Instructions we learned
  - Read/Assignment on all primitive Types, references, volatile long/double
    - Avoid data races make all shared variables without using a lock volatile
    - java.util.concurrent.atomic → Atomic Type such as Long, Adders, Accumulators etc..

Atomic Integers & Lock Free E-Commerce

- Example
  - int initialValue = 7;
  - AtomicInteger atomicInteger = new AtomicInteger(initialValue);
  - atomic.incrementAndGet();   // return the new value
  - atomic.getAndIncrement();   // return the previous value
  - atomic.decrementAndGet();   // return the new value
  - atomicInteger.getAndDecrement ()   // return the previous value
  - atomicInteger.addAndGet(initialValue)’   // returns the new value which is old value + 7
  - atomicInteger.addAndGet(initialValue)   // returns the new value which is old value
- Pros – No need for locks or synchronization and no race conditions or data races
- cons : Only the operation itself is atomic
  - Still a race condition between two separate operations
- Performance
  - On par and sometimes more performant than a regular integer with a lock as protection
  - If used only by a single thread, a regular integer is preferred since certain hardware/software optimizations become imposbile because of a data race.

Atomic References, Compare and Set, Lock Free High Performance Data Structure

- AtomicReference(V inititalValue)
  - wraps the reference of a class
  - Has the ability to perform atomic operations on that class
  - V get()   // Returns the current value
  - void set(V newValue)   // Sets the value to newValue
-

Atomic References, Compare and Set, Lock Free High Performance Data Structure

- boolean compareAndSet(V expetedValue, V newVlaue)
  - Assigns new value if currentValue == expectedValue
  - Ignores the new value if the current value != currentValue
  - Example
    - String oldName = “old name”;
    - String newName = “new name”;
    - AtomicReference<String> atomicReference = new AtomicReference(oldName)
    - if (compareAndSet(oldName, newName) {
      - System.out.println(“New Value is “ + atomicRerference.get()) }
    - else
      - System.out.println(“Nothing Happen”);
  - The Compare and Set compares the oldName passed into the function with current value of the oldName.   If they are the same then prints out the second parameter of the compare and set.
  - Change the value in the atomic reference when creating the instance and then run it will print “Nothing has happen”
  - Protects from another thread changing the value

Atomic References, Compare and Set, Lock Free High Performance Data Structure

- CompareAndSet
  - Available in all atomic classes
  - Compile into an atomic hardware operation
  - Many other atomic methods are internally implemented using CompareAndSet
- Implementing a Stack
  - The standard Stack in the Java API using the Vector and is based on Locks
  - Our implementation will use the LinkList where the first element is the first element of the list
  - 1st Implementation crate a class where the push and pop methods are synchronized
  - Example
    - public static class {
      - private AtomicReference<StackNode<T> head = new StackNode<>(value);
      - pubic void push(T value) {
        - StackNode<T> newHeadNode = new StackNode<>(value);
        - while (true) {
          - StackNode<T> currentHeadNode = head.get();
          - newHeadNode.next = currentHeadNode;
          - if (head.compareAndSet(currentHeadNode, newHeadNode) break else lockSupport.parkNanos(1);
          - // If the currentHead and newHead are not the same it is true

Atomic References, Compare and Set, Lock Free High Performance Data Structure

  - public T pop() {
    - StackNode<T> currentHeadNode = head.get()
    - StackNode<T> newHeadNode
    - while (currentHeadNode != null) {
      - newHeadNode = currentHeadNode.next;
      - if (head.compareAndSet(currentHeadNode, newHeadNode)) {
        - break; // Successfully pop the stack
      - } else { // Failed head changed by another thread
        - LocksSupport.parkNanos(1);
      - }
    - }
    -
    - return currentHeadNode != null ? currentHeadNode.value : null;
  - }
  -

- Both the push and pop method change the head.
  - The head is a sharedResource that could cause thread issues if not synchronized properly
  - Remember CompareAndSet is one instruction

Atomic References, Compare and Set, Lock Free High Performance Data Structure

- Performance
  - Blocking with 158,000,000 million in 10 seconds
  - Lock Free with 509,000,000 million in 10 seconds

Futures and Callable &Iterators

- How do you check whether an ExecutionService task executed successfully
  - Future checks a return value
  - Example
    - Future future = executorService1.submit(new Runnable() { public void run() { System.out.println(“test”); } });
      - future.get() //   returns null if has finished correctly
  - Callable
    - Return a result from the thread
  - Example
    - Future furtureFromCallable = executorService1.submit(new Callable() { public void String call() throws Exception { return “result” } });
    - System.out.println(futureFromCallable.get());
- Fail-Safe vs Fail Fast Iterators
  - Fail Fast iterators throw a ConcurrentModificationException if there is a modification to the underlying collection is modified
  - Fail Safe Iterators do not throw iterators do not throw exception, but Takes a copy of the data structure and iterates over that