**Multithreading**

**Why we need multithreading?**

Responsiveness – concurrency

Performance – parallelism

**Basics of Operating System**

1. When we turn on our computer a special program called OS is loaded from the disk to memory.

2. OS takes over, provides abstraction for us, and helps us interaction with hardware and the CPU.

3. All application resides on a disk in form of a file.

4. When user runs an application, the OS takes the program from the disk and creates the instance of the application in the memory. That instance is called a process or context of application.

5. Each process is completely isolated from any other process that runs on the system.

6. Few things that the process contains are metadata like

· PID (Process ID)

· Files that the application opens for reading & writing.

· The code, which is going to be executed.

· The heap, which contains all the data our application needs and finally at least one thread called Main Thread.

7. Thread contains two main things, the stack and instruction pointer,

8. All rest of the components in the process are shared by all the threads.

Stack – region in memory where the local variables are stored, and passed into systems.

Instruction Pointer- Address of the next instruction to execute.

**Context Switch**

Each instance of the application runs independently from other processes.

Each processes has one or more threads. All these threads are competing with each other to be executed on the CPU.

Even if we have multiple cores, we have way more threads than cores. So, the OS will have to run one thread then stop it, run another thread, stop it and so on.

The act of stopping thread 1.

Schedule thread 1 out.

Schedule thread 2 in.

Start thread 2.

is called a context switch.

**Context switch is not cheap**, and is the price of multitasking (concurrency).

Each thread consumes resources in the CPU and memory.

When we switch to different thread, we need to store the data for one thread and restore data for another thread.

Too many threads can cause thrashing.

**Thrashing** – spending more time in managing than in real productive work.

Thread consume less resource than processes.

Context switching b/w two threads from the same process is cheaper than context switch b/w different processes.

**Thread scheduling**

FCFS

Problem – Long thread can cause starvation.

May cause UI threads being unresponsive – Bad user experience.

Shortest Job First

There are user related events coming all the time and UI threads are usually smaller.

Therefore, we keep scheduling the shortest job first, all the time. The longer task than involve computation is never be executed.

In most OS, divides the time into moderately sized pieces called epochs.

In each epochs, OS allocates a different time slice to each thread. Not all the thread is co

**Dynamic priority = static priority + Bonus**

Developer sets static priority programmatically.

Bonus is adjusted by the OS in each epoch, for each thread.

OS give preference to UI threads.

OS give preference to threads that did not complete in last epoch, or did not get enough time to run preventing starvation.

**When to prefer Multithreaded Architecture**

Prefer if the tasks share a lot of data.

Threads are much faster to create and destroy.

Switching b/w threads of the same process is faster (Shorter context switches).

Security and stability are of higher importance.

Tasks are unrelated to each other.

public class ThreadBasicExample {

public static void main(String[] args) throws InterruptedException {

Thread thread = new Thread(new Runnable() {

@Override

public void run() {

// code will run in a new thread

System.*out*.println("We are in Thread: " + Thread.*currentThread*().getName());

System.*out*.println("Current Thread Priority: " + Thread.*currentThread*().getPriority());

}

});

thread.setName("New Worker thread");

thread.setPriority(Thread.*MAX\_PRIORITY*);

System.*out*.println("We are in Thread: " + Thread.*currentThread*().getName() + " Before starting a new thread.");

// this will instruct the jvm to create a new thread and pass it to the OS

thread.start();

System.*out*.println("We are in Thread: " + Thread.*currentThread*().getName() + " After starting a new thread.");

// It will not spin a loop, it instruct the OS to not schedule the current thread until the time passes.

// During that time this thread does not consume any CPU.

Thread.*sleep*(10000);

}

}

**Thread Termination & Daemon Thread**

Stopping one thread from another thread.

**Thread Termination**

* Thread consume resources
  + Memory & kernel resources.
  + CPU cycles and cache memory.
* If a thread finishes its work, but the application is still running, we want to clean up the thread resources.
* If a thread is misbehaving, we want to stop it.
* By default, the application will not stop as long as at least one thread is still running.

thread.interrupt()

ThreadA ThreadB

| |

| |

| |

threadB.interrupt()----->InterruptSignal

| |

| |

V V

**When can we interrupt a thread?**

1. If a thread is executing a method that throws an interrupted exception.

2. If the thread's code is handling the interrupt signal explicitly.

private static class BlockingTask implements Runnable {

@Override

public void run() {

try{

Thread.sleep(5000000);

} catch(InterruptedException e) {

}

}

}

psvm(-) {

thread.start();

thread.interrupt();

}

OR

for(int i = 0; i < MAX; ++ i) {

if(thread.currentThread().isInterrupted()) {

sout("Interrupted");

}

}

**Daemon Thread**

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

**Scenario 1:**

Background tasks, that should not block our application from terminating.

Example: File saving thread in a text editor.

**Scenario 2:**

Code in the worker thread is not under our control, and do not want it to block our application from terminating.

Example: Worker thread that uses an external library.

thread.setDaemon(true);

**Joining threads**

Threads coordination

- Different threads run independently.

- Order of execution is out of our control.

**What if one thread depends on another thread?**

Thread A

| output

|

|

V input

Thread B

**Naive Solution**

Thread B runs in a loop and keeps checking if Thread A's result is ready.

void waitForThreadA() {

while(!threadA.isFinished()) {

// burn CPU cycles

}

}

waste waste

| ThreadBcheck | ThreadAdoingwork | ThreadBcheck |

**Desired Solution**

Thread B check and goes to sleep and let thread A do its work.

Only when Thread A is done, thread B wake up and takes result.

join(), join(long millis, int nanos), join(long millis)

**Performance & Optimizing for latency**

**Latency -** time to completion of a task. measured in time units.

**Throughput -** amt. of tasks completed in a given period. Measured in task/time unit.

Task 1

...

...

Task N

|--------| Latency = T / N

**N = ?**

N = no. of cores

Core 1 ----> task 1

Core 2 ----> task 2

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

Assumption is nothing else is running that consumes a lot in CPU.

**Hyperthreading -** Hyper-threading is a process by which a CPU divides up its physical cores into virtual cores that are treated as if they are actually physical cores by the operating system. These virtual cores are also called threads.

**Inherent cost of parallelisation and aggregation**

1. Breaking task into multiple tasks.

2. Thread creation, passing tasks to threads.

3. Time b/w Thread.sleep() to thread getting started.

4. Time until the last thread finishes and signals.

5. Time until the aggregating thread runs.

6. Aggregation of subresults into a single artifact.

**Optimizing throughput**

**Thread pooling**

Creating the threads once and reusing them for future tasks.

T7--->T6---->Task5 | T1 T2 |

| T3 T4 |

-----------

4 tasks/second

* Implementing a thread pool is not trivial.
* JDK comes with a few implementation of thread pools.
* Fixed thread pool executor.

Executor executor = Executors.newFixedThreadPool(4);

Runnable task = ...;

executor.execute(task);

**STACK & HEAP MEMORY REGIONS**

**Stack Memory Region**

* Memory region where
  + methods are called
  + arguments are passed
  + local variable are stored.
* Stack + Instruction pointer = stack of each thread's execution

void main(string[] args) {

int x = 1;

int y = 2;

int result = sum(x, y);

}

**Stack's properties**

* All variables belong to 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 hierarchy is too deep. we may get an stackOverflow exception. (Risky with recursive calls).

**Heap Memory Region**

* Objects (anything created with the new operator).
  + String
  + Object
  + Collections
* Members of classes.
* Static variables.
* Governed and managed by garbage collector.
* Objects stay as long as we have a reference to them.
* Members of classes - exists as long as their parent object exist. (same life cycle as their parent)
* static variables - stay forever.
  + References != Objectd

Object ref1 = new Object();

Object ref2 = ref1;

ref1 --------------

|---> Object

ref2 --------------

**References**

Can be allocated on the stack.

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

**Objects**

Always allocated on a heap.

Heap (Shared) - Objects, class members, static variables.

Stack (Exclusive) - Local primitive types, local references.

**Resource Sharing b/w threads**

**Resource**

Something that represent data or a state.

* Variables (Integers, String, ...)
* Data Structures
* File or connection handles.
* Message or work queues.
* Any objects ...

**Race Condition**

one thread - incrementing thread.

second thread - decrementing thread.

Unexpected result

- Inventory counter is a shared object.

- the items member is shared b/w the two threads.

- items ++ and item -- are happening at the same time and not an atomic operations.

**Atomic operation**

* An operation or a set of operations is considered atomic, if it appears to the rest of the system as if it occurred at once.
* Single step - "all or nothing".
* No intermediate state.

**Items ++**

1. Get current value of item.

2. increment current value by 1.

3. store the result into items.

**Critical Sections**

void aggregateFunction() {

**enter critical section**

operation1();

operation2();

**exit critical section**

}

Only one thread can enter the critical section at once.

**Synchronized - Monitor/Lock**

**synchronized**

* Locking mechanism
* Used to restrict access to a critical section or entire method to a single thread at a time.

**1. Synchronized - Monitor**

public class ClassWithCriticalSections {

public synchronized void method1() {

...

}

public synchronized void method2() {

...

}

}

When multiple threads are going to call the method on the same object of this class, only one thread will be able to execute either of those methods.

Note: If Thread A is executing method1, Thread B can't execute any method i.e. method1 & method2 because synchronized is applied per object.

Every synchronized method is a different door to a room, where if you lock one door, all the other doors get locked immediately & automatically.

private static class InventoryCounter {

public int items = 0;

public synchronized void increment() {

items ++;

}

public synchronized void decrement() {

items --;

}

public synchronized int getItems() {

return items;

}

}

**2. Synchronized - Lock**

public class ClassWithCriticalSections {

Object lock = new Object();

public void method1() {

synchronized(lock) {

critical section

}   
}

}

* synchronized block is reentrant. (A thread can access another synchronized method/block).
* A Thread cannot prevent itself from entering a critical section.

public synchronized method1() {

method2();

}

public synchronized method2() {

...

}

**Atomic Operations**

* All reference assignments are atomic.
* We can get and set references to objects automatically.

Object a = new Object();

Object b = new Object();

a = b;

getters & setters

* All assignments to primitive types are safe except long and double.

long x = 5;

long y = 10;

x = y;

**Upper 32 bit | lower 32 bit**

because there may be two write operations by the CPU.

Solution: volatile double x = 1.0;

volatile double y = 9.0;

x = y; // atomic

Now, they are guaranteed to be performed in a single hardware operation.

* Classes in java.util.concurrent.atomic;
* Those are more advanced operations.

**Race Conditions**

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

Incrementing thread Decrementing thread

1. currentVal <- items = 0

2. newVal <- currentVal + 1 = 1

3. currentVal <- items = 0

4. newVal <- currentVal - 1 = -1

5. items <- newVal = -1

6. items <- newVal = 1

**Solution**

* Identification of CS where the race condition is happening.
* Protection of CS by a synchronizing block.

**Data Race**

public class SharedClass {

int x = 0;

int y = 0;

public void increment() {

x ++;

y ++;

}

public void checkForDataRace() {

if(y > x) {

throw new DataRaceException("...");

}

}

}

Logically x >= y should always hold.

**Problem:**

* Compiler and CPU may execute the instructions out of order to optimize performance and utilization.
* They will do so while maintaining the logical correctness of the code.
* Out of order execution by the compiler and CPU are important to speed up the code.
* The compiler re-arranges instructions for better
  + Branch prediction (optimized loops, "if" statements, etc.)
  + Vectorization - parallel instruction execution (SIMD).
  + Prefetching instructions - better code performance.
* CPU re-arranges instructions for better hardware units utilization.

May lead to unexpected, paradoxical and incorrect results in case of multithreading.

**Data Race - Solution**

Establish a happens - Before semantics by one of these methods:

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

Rule of Thumb

* Every shared variable (modified by at least one thread) should be either.
  + Guarded by a synch block or Declared Volatile.

**Locking Strategies & Deadlocks**

Fine-Grained Locking vs Coarse Grained Locking

Separate lock for all resources. Single lock for all the resources.

**Coarse grained locking**

Single lock to worry about.

public class SharedClass {

private DatabaseConnection dbConnection;

private List<Task> taskQueue;

public synchronized Item getItemFromDB() {

...

}

public synchronized void addTaskQueue() {

...

}

}

Drawback: One thread at a time can make progress.

**Fine Grained locking**

Separate lock for every resource.

public Item getDbConnection() {

synchronized(dbConnection) {...}

}

public void addTaskToQueue() {

synchronized(taskQueue) {...}

}

Allow more parallelism and less contention.

Problem we may run into when we have multiple lock is Deadlock.

**Deadlock**

Situation where everyone is trying to make progress but cannot because they are waiting for another party to make a move.

Eg. I will move if you move first.

1. lock(A)

2. lock(B)

3. lock(A)

4. lock(B) delete(B, item)

delete(A, item) add(A, item)

add(B, item) unlock(A)

unlock(B) unlock(B)

unlock(A)

**Deadlock Conditions**

* Mutual Exclusion - Only one thread at a time 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 threads each one is holding one resource and waiting for another resource.

**Solution to deadlock**

At least one of the conditions does not met and deadlock cannot happen.

- Avoid Circular Wait- Enforce a strict order in lock acquisition.

Thread 1 Thread 2

lock(A) lock(A)

lock(B) lock(B)

... ...

unlock(B) unlock(A)

unlock(A) unlock(B)

* Easy to do with small no. of tasks.
* Maybe hard to accomplish if there are many lock in different places.

**Other Techniques:**

* Deadlock detection- watchdog
* Thread Interruption(not possible with synchronized).
* tryLock operations(not possible with synchronized).

**Advanced locking**

java.util.concurrent.locks.ReentrantLock

* works just like the synchronized keyword applied on an object.
* Require explicit locking and working.

Lock lock = new ReentrantLock();

Resource res = new Resource();

public void method1() {

lock.lock();

...

use(resource);

lock.unlock();

}

**Disadvantage-**

public void use() throws Exception{

lock.lock();

throw Exception();

lock.unlock(); <-- never Reached

}

Resources will remain locked forever.

**Solution-**

public int use() throw Exception{

lockObj.lock();

try {

operation();

return val;

} finally {

lockObj.unlock();

}

}

More control and more operations over the lock.

* getQueuedThread() - Returns a list of threads waiting to acquire a lock.
* getOwner() - current thread owns the lock.
* isHeldByCurrentThread() - Queries if the lock is held by the current thread.
* isLocked() - Queries if the lock is held by any thread.

**Fairness -**

* ReentrantLock(true)
* May reduce the throughput of the application.

**LockInterruptibly -**

* Watchdog for deadlock detection and recovery.
* Waking up threads to do clean and close the application.

**tryLock() -**

boolean tryLock()

* Return true and acquire a lock if available.
* Returns false and does not get suspended, if lock is unavailable.

if(lockObj.tryLock()) {

try {

useResource();

} finally {

lockObj.unlock();

}

} else { ... }

Note: under no circumstances does the tryLock() method block!

**TryLock() use case-**

Real time applications where suspending a thread on a lock() method is unacceptable.

Ex- Video/image processing

High speed/ low latency trading systems.

Use interface applications.

**Reentrant Read Write Lock**

Read lock & Write Lock

Why?

* Race condition require
  + Multiple threads sharing a resource.
  + At least one thread modifying the resource.
* Solution - Complete mutual exclusion.
  + Regardless of operation(read/write/both).
  + Lock and allow only one thread to CS.

Multiple Threads can safely read from a shared resource.

Use case -

* Sync and Reentrant lock do not allow multiple readers to access a shared resource concurrently.
* Not a big problem in general case.
* If we kept the CS short, the chances of contention over a lock are minimal.

When to use

* When read ops are predominant.
* Or when the read ops are not as fast.
  + read from many variables.
  + read from a complex data structure.
* Mutual Exclusion of reading threads negatively impacts the performance.

ReentrantReadWriteLock rwlock = new ReentrantReadWriteLock();

Lock readLock = rwlock.readLock();

Lock writeLock = rwlock.writeLock();

writeLock.lock(); readLock.lock();

try { try {

modifySharedResource(); modifySharedResource();

} finally { } finally {

writeLock.unlock(); writeLock.unlock();

} }

Note:

* Multiple threads can acquire the read lock simultaneously.
* only a single thread is allowed to lock a write lock.
* Mutual Exclusion between readers and writers.
  + If a writ e lock is acquired, no thread can acquire a read lock.
  + If at least one thread holds a read lock, no thread can acquire a write lock.

**Inter Thread Communication**

**Semaphores**

* Can be used to restrict the no. of "users" to a particular resource or a group of resources.
* Unlike the locks that allows only one "user" per resource.
* The semaphore can restrict any given no. of users to a resource.

Ex - Parking lot.

Semaphore semaphore = new Semaphore(NO\_OF\_PERMITS);

semaphore.acquire(5);

useResource();

semaphore.release(5);

**Semaphore == lock ?**

No. of permits = 1

NOTE: Semaphore is not a great choice for a lock because:

* semaphore doesn't have a notion of owner thread.
* Many threads can acquire a permit.
* The same thread can acquire the semaphore multiple times.
* The binary semaphore(init with 1) is not reentrant.
* semaphore can be released by any thread.

**Semaphore - Producer Consumer**

Semaphore full = new Semaphore(0);

Semaphore empty = new Semaphore(1);

Item item = null;

while(true) { while(true) {

empty.acquire(); full.acquire();

item = produceNewItem(); consume(item);

full.release(); empty.release();

} }

This will work for one producer & one consumer.

Semaphore full = new Semaphore(0);

Semaphore empty = new Semaphore(CAPACITY);

Queue queue = new ArrayDeque();

Lock lock = new ReentrantLock();

Producer Consumer

while(true) { while(true) {

Item item = produce(); full.acquire();

empty.acquire(); lock.lock();

lock.lock(); Item item = queue.poll();

queue.offer(item); lock.unlock();

lock.unlock(); consume(item);

full.release(); empty.release();

} }

Interrupt(), join(), acquire(), release()

Condition condition = lock.newCondition();

condition.await();

condition.signal();

**What's wrong with locks?**

* Deadlock
  + generally unrecoverable
  + can bring the app to a complete halt.
  + slow
* priority inversion
  + two threads sharing a lock.
    - low priority thread (document saver).
    - high priority thread (UI)
  + 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)
  + threads dies or forgot to release a lock
    - unrecoverable
* Performance
  + performance overhead in having contention over a lock.
    - thread A acquires a lock.
    - thread B tries to acquire a lock and gets blocked.
    - thread B is scheduled out (context switch).
  + For latency sensitive applications, this overhead is very significant.

**Lock free techniques**

utilize operations which are guaranteed to be one hardware operation.

**AtomicInteger**

Pros:

* Simplicity
* No need for lock & synchronization.
* No data race & race condition.

Cons:

* Only the operation itself is atomic.
* there's still race condition b/w 2 seperate atomic operations