JVM exits after all non-daemon threads have completed

setDaemon(true) to set a thread as daemon, by default all threads are non-daemon threads

Daemon threads are killed whenever the main thread completes.

Read and write operations are atomic for volatile variable including long and double

Java64 treats long and double write/read operation as 2- 32-bit operations

Liveness: state in which thread is unable to make progress

Race condition - correctness of the code depends on timing of multiple threads

Java Threads are OS threads (Native/Kernel)

**Parallelism**

When multiple threads can run at the same

Doing parts of the same tasks in different CPU's

ThreadPools, Executors, ForkJoinPool

**Concurrency**

When threads need to coordinate with each other or shared resource is being accessed by the threads

Doing loosely related tasks in same CPU

Locks, Atomic classes, Concurrent data structures, CompletableFuture, CountDownLatch/Phaser/Semaphore

**Contention**

When threads are trying to access the same resource (may or may not be a lock)

**Context switching**

Storing the state of 1 task to pause and loading the saved state of another task to resume the task

**Volatile** is used to address visibility problem

When a variable is shared between 2 threads, then updates made to variable by thread 1 is not visible to thread 2 since the update is visible in local cache of thred-1, Using volatile forces the thread-1 to flush the value from local cache to shared cache and the thread-2 updates its value in the local cache from the shared cache

-Used in flags/or when

**Atomic variables are used for compound operations**

**CAS (Compare and Swap)**

1. **Get value from memory as current value**
2. **Increment the local value**
3. **If the current value is same as value in memory update memory with new value, else do step 1-3**

**AtomicInteger //**uses CAS

**AtomicLong //** slower because of too many retries to increment or decrement in case of large no of threads (CPU cycle wasted)

**AtomicBoolean**

**AtomicReference**

**AtomicIntegerArray**

**AtomicLongArray**

**AtomicReferenceArray**

**LongAdder //** each threads maintains its own variables which are incremented internally

The increment or decrement operations internally hold a Cell array which maintains the value calculated by each thread, the array can grow as no of threads increase. (Higher memory usage)

On calling sum(), the elements in the Cell array are added up

**LongAccumulator** - generic version of LongAdder can be used to perform any operations based on lambda expression

Suitable when there is high contention

**ThreadLocal**

Used to create object per thread, instead of creating object per task

All updates done to the object are visible in that particular thread, so the variable should be removed(remove()) after completing the task because the thread can reuse this value when it runs again

**Happens-before relationship**

If there are instructions before volatile/synchronized/lock context in one thread and these are instructions are present after the volatile/synchronized/lock/join/start context on another thread then they will also have the updated values

Writes before the volatile write is visible to statements after volatile variable has been read

JVM does not do any reordering of instructions in this case (reordering of the no-volatile variables can be done)

int a, b;

volatile int x;

Write thread

a=1;b=1;x=1;

Read thread

temp=x;

c=a + b;

**Executors**

Decouple task submission from task execution

**shutdown**() - initiates the shutdown, task which are queued and task currently running in threads are allowed to complete

**shutdownNow**() - tasks which are queued are not run and are returned, only tasks running in threads are allowed to complete

**isTerminated() -** returns true if all tasks are completed

**awaitTermination() -** blocks until all tasks are completed or till timeout

**Types of Executors**

CachedThreadPool

Creates new thread as and when required

Makes use of synchronous queue which contains only 1 task, if a thread is busy it will create a new thread and add it to pool

It kills the thread if it is idle for 60s

FixedThreadPool

Reuses the fixed threads created

Makes use of blocking queue for tasks

SingleExecutableThread

Uses blocking queue

Creates thread even if the thread is killed

Used when sequencing of tasks is required

ScheduledExecutorService

**schedule()** - run the task after delay time

**scheduleAtFixedRate()** - run the next task after every delay period

**scheduleWithFixedDelay()** - run the next task after the previous task is completed + delay time

Uses a delay queue in which task are ordered according to delay

Creates new thread as and when required

It kills the thread if it is idle for 60s

Both scheduleAtFixedRate and FixedDelay accept runnable whereas schedule accepts both callable or runnable

execute accepts Runnable, submit accepts both runnable and callable, invoke accepts only callable

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Executor type** | **Core pool size** | **Max pool size** | **Queue** | **Timeout** |
| FixedThreadPool | Parameter | Same as core pool | LinkedBlockingQueue | N.A |
| CachedThreadPool | 0 | Integer max value | SynchronousQueue | 60 |
| ScheduledThreadPool | Parameter | Integer max value | DelayedWorkQueue | 60 |
| SingleThreaded | 1 | 1 | LinkedBlockingQueue | NA |

LinkedBlockingQueue is unbounded but ArrayBlockingQueue has fixed size

**Rejection policies**

AbortPolicy: New task submitted will throw RejectedException

DiscardPolicy: New tasks will be discarded

DiscardPolicy: New tasks will replace the oldest task

CallerRunsPolicy: New task will be run by the caller itself

ThreadFactory - extend this class to customize thread creation

ForkJoinPool is suitable when a large task needs to divided

ThreadPool is for independent tasks

**isDone()** always returns true when cancel() is called

**cancel(true)** will throw InterruptedException when task is still running

**invokeAll / invokeAny** with timeout

Will wait for the specified timeout for the get() methods to complete, if any task takes longer time then all subsequent running/queued tasks are cancelled (Cancellation Exception is thrown)

Returns a list of completed futures(gets blocked till all runnable are executed) in the order of how the runnable were provided whereas submit returns a pending future(gets blocked till that runnable is executed)

**CompletableFuture** extends Future **(Asynchronous API)**

provides completeExceptionally() which can be used to handle exceptions inside the task, so that get() will throw and Exception

supplyAsync() takes a Supplier<T> as argument and returns CompleteableFuture<T>

By default it uses the forkjoinpool but other executors can be provided

join() method is similar to get() but does not catch chekced Exceptions

CompletableFuture and parallel streams provide same responses since they use the forkjoinpool, but CompleteableFuture can provide better performance with an executor

thenApply() is used to apply function on the CompletableFuture it does not block for the CompletableFuture to complete

thenCompose() is used to pipleline async operations, providing the output of one async operation as input to other one

thenCombine() is used to combine result of 2 async operations where ther is no waiting involved

thenAccept() consumes the CompletableFuture

All the methods have async form in which the particular operation will be exeucted in different thread

**Synchronizers**

**CountDownLatch**

Count down latch is synchronizer which causes the thread calling the await() to be blocked until the count reaches to 0

Thread 1 needs to get some task done by Thread 2

CountDownLatch created with count 1

Thread 1 will call the await on CountDownLatch

Thread 2 will execute and after execution calls countDown() method on the latch

It provides method for getting the current count

In countDownLatch await() does not throw a TimeOutException if time elapsed before count down instead it returns false and execution is resumed

**Join vs CountDownLatch**

Join allows the thread to start only after the other thread completes, while CountDownLatch is more based on condition rather than completion

Join needs to be called on the thread, when using executor service this may not be possible

**CyclicBarrier**

When multiple threads need to wait for each other so that once all the threads reach a particular point(barrier) some works needs to be done.

It allows a runnable to be executed when the barrier is reached, the last thread recaching the barrier executes the runnable

The count can be reset and used again

It provides mechanism to determine the no of threads waiting at the barrier and count required to trop the barrier.

Broken Exception is thrown when

Awaiting thread times out (await(timeout)) - It throws a timeoutException and barrier is broken other threads waiting get BrokenException

Barrier is reset (reset())

Barrier action failed due to exception - All awaiting threads throw BrokenException

Barrier is broken

await() - return arrival index of that thread (noOfParties-1), last thread to arrive has value 0

**Phaser**

Works as both CountDownLatch and CyclicBarrier

latch.await() - awaitAdvance(1)

latch.countDown() - arrive()

barrier.await() - arriveAndAwaitAdvance(1)

It allows dynamic registration using register(), it can also increase the parties using the bulkRegister(), it can arrived and deregister using arriveAndDeregister()

onAdvance() is similar to barrier action(Runnable)

**Exchanger**

Synchronous queue if the consumer thread is busy then producer thread is blocked when it calls put(), on when the take() is called producer thread unblocks

In blocking queue the producer does not wait while calling put if the queue is empty

Size is 0

Similar to synchronous queue but the exchange takes place between both threads in both direction

Buffers (Producer exchanges a full buffer to consumer and consumer exchanges empty buffer to producer)

**Semaphore** (Manage access to limited resources)

Mechanism to restrict number of concurrent accesses by use of permits.

Each thread will acquire a permit and release a permit after doing the work

If no permits are available the thread will be blocked, tryAcquire() is non-blocking which return false if no permit is available.

Semaphore can ensure fairness so the maximum waiting thread can be provided with permit by using fairness overloaded parameter while creating a semaphore

Also provides the no of available permits

A thread can acquire more than 1 permit using aquire() method but it has to release the same number of permits

A thread can release more no of permits than it acquired but other threads cannot acquire this increased permits, it can only access permits that are available

**Locking framework**

|  |  |
| --- | --- |
| Lock (Interface) | ReentrantLock |
| ReadWriteLock (Interface) | ReentrantReadWriteLock |

Before entering a critical section the thread copies the shared resource value from the memory to local thread memory and while exiting the critical section

it writes the value into main memory

Synchronized uses intrinsic locks which are reentrant

Synchronized can be used

1. Static methods - Class level lock
2. Instance methods - Instance level lock
3. Blocks - Instance level lock

Lock are equivalent to synchronized methods or blocks

Condition is equivalent to wait() and notify()

synchronized(){ lock.lock()

object.wait(); condition.await()

} lock.unlock()

synchronized(){ lock.lock()

object.notify(); condition.signal()

} lock.unlock()

condition.signal() will resume the longest waiting thread

Reentrant lock allows calling the lock() method any number of times(Increases getHeldCount)

ReentrantLock(true) //ensures fairness all waiting threads are kept in queue and lock is given to the most waiting thread

ReentrantLock(false) //unfair but fast , thread starvation, default

tryLock() - non blocking, can cause unfairness in accessing lock even if Reentrant lock ensures fairness, so it is better to use tryLock with timeout

**ReadWriteLock**

Readlock and Writelock are created from the same readwrite lock

Either n threads can read or 1 thread can write at any time not both simultaneously

In ReentrantReadWrite lock if thread-1 is reading and thread-2 is waiting for write, thread 3 comes for reading then after thread 1,thread 2 will be allowed to write

Condition present for write lock only

**StampedLock**

Lock returns a stamp(long value) which is used to unlock or check if the lock is still valid

It provides optimistic read locking(non-blocking) using tryOptimisiticRead() in which the stamp is returned which should be validated using validate()

Stamp returned is 0 if it fails to acquire lock

if after acquiring read lock thread sleeps then write lock can be still acquired, read lock is no longer valid

Read lock can be converted to write lock using tryConvertToWriteLock(stamp) and vice versa using tryConvertToReadLock(stamp), it returns 0 if it fails to do so.(Non-blocking)

They are not reentrant so may lead to deadlock

**Advantages of Locks over synchronized keyword**

1. Allows non-blocking version tryLock()
2. Ensures fairness by providing lock to longest waiting thread and provides list of waiting threads
3. Allows calling lock() and unlock() in 2 different methods
4. Required to call unlock() manually, in case of synchronized this is not required
5. Threads waiting for lock cannot be interrupted in case of synchronized, but locks provides lockInterruptibly() to interrrupt waiting thread

**SpinLocks**

Keep trying to acquire the lock without going to wait state (CPU cycles are wasted, applicable only when locks are used for short period, in case of longer duration causes thread starvation)

JVM performs adaptive locking, by profiling the code at runtime so it will try to acquire the lock for some time before going to wait state(wait time depends on the code)

Pre Java8 options to enable spin locking and no of tries before blocking can be provided

**onSpinWait() ??**

**Fork Join Framework**

Implementation of ExecutorService which uses the thread pool called forkjoin pool

RecursiveTask if the task returns anything and RecursiveAction if it does not return anything

Task should extend RecursiveTask

compute() - methods responsible for creating the subtasks (Recursive call)

Use of recursion here, check if task is small enough to be done(exit condition) or else divide the task

Delegate one task by calling fork()

Call compute on the other task

fork() - delegate subtask to another thread (Task is submitted to the forkjoin pool)

join() - wait for the thread to compute result and join the result

Always call join after calling computation on both tasks

calling compute for one task and fork() for another task allows the compute method to be called in the same thread causing reuse of that thread

Work stealing

Threads maintain double linked queue from which a thread takes the task from HEAD,

When a thread has finished tasks allocated to it the thread steals task from another thread by reading task from TAIL

ForkJoinPool invoke() waits for thread to complete

execute() does not wait for result (For getting result use join() method on task)

If any RunTimeException is thrown by the tasks in forkjoinpool other tasks are still done

If checked exception is to be thrown then completeExceptionally() to wrap the exception (The exception will be thrown during join())

You can also cancel a task from another task but it throws CancellationException

**Interrupting threads**

Interrupt() - indicate to stop, may or may not be stopped

isInterrupted() - used to check if interrupt flag is set

Interrupted() - reset the interrupt flag

**Thread safe data structures**

**Concurrent collections**

They do not support null values, and use CAS for operations which are non-blocking

**ConcurrentHashMap**

Reads are non-blocking

Hashmap is divided into segments (default 16, it maintains 16 locks one for each segments so threads can work on each segment)

During put and remove operation based on the hash value of the key, the segment is identified and synchronized block is used for addition or removal

Reads are always non-blocking even if a thread is writing in the same segment, it will read the updated values(volatile)

In case of putAll or clear only some of the values will be reflected

Writes on the same segment are blocking

Hashtable locks the entire collection ,also the get and put operations are synchronized

**ConcurentLinkedQueue** Uses CAS for adding and removing

**ConcurentLinkedDeqeue**

**ConcurrentSkipListMap - sorted based on keys**

**ConcurrentSkipListSet - sorted**

**BlockingQueue**

Not possible to insert null values

Uses internally Locks and condition, so are blocking calls

Used for consumer producer problems

**ArrayBlockingQueue //** bounded as it internally uses array, longest waiting task are placed at head

**DelayedQueue //** tasks must implements Delayed interface(extends comparable) which specify the delay time after which the element will be consumed, so the tasks are stored according the delay times

**LinkedBlockingQueue //** unbounded as internally uses linked list (Max is integer max value)

**PriorityBlockingQueue //** unbounded tasks must implement Comparable

**SynchronousQueue //** contains single task

**add(), remove(), element()** throws Exception

**offer(), poll(), peek()** return true or false with or without timeout

**put() and take()** blocks

**Copy on collections**

Uses internally ReentrantLocks

Write operations are blocking, and creates a new copy of the collection

Read operation are non-blocking

Used when no of reads outnumber write operations

**CopyOnWriteArrayList**

**CopyOnWriteArraySet**

list.iterator() is failsafe and does not throw ConcurrentModificationException, since it is iterating over the copy of the collection

If the collection is modified during iteration it will get the updated collection only during next iteration

**Stopping a thread**

Thread.interrupt()

Future.cancel() and shutdownNow() internally uses interrupts to attempt to stop the thread

You can use volatile or AtomicBoolean to check for condition and exit the thread

**ThreadGroups**

Used to group thread objects, every thread belongs to a thread group, if not explicitly specified thread belongs to **main** thread group

JVM creates 2 thread groups

1. **system** - contains threads used for JCM specific tasks
   1. **main** - contains the main thread
      1. Application created thread groups

You can set priority on the threads within the group as well as interrupt all threads once

If no parent thread group is specified, then **main** becomes the parent thread group