package java.lang;

public class Thread implements Runnable {

public void start( );

public void run( );

public void stop( ); // Deprecated, do not use

public void resume( ); // Deprecated, do not use

public void suspend( ); // Deprecated, do not use

public static void sleep(long millis);

public static void sleep(long millis, int nanos);

public boolean isAlive( );

public void interrupt( );

public boolean isInterrupted( );

public static boolean interrupted( );

public void join( ) throws InterruptedException;

}

1. **Defination of thread**

A thread is a flow of execution in a process. When we run a

program, there is at least one thread of execution for its process.

1. Different thread state ?

Ans : According to javadocs

**public** **enum** State is a inner enum of Thread class, it has few states

***NEW , RUNNABLE , BLOCKED , WAITING , TIMED\_WAITING***

***TERMINATED***

1. Difference between WAITING and BLOCKED ?

Ans :

A thread goes to wait state once it calls wait() on an Object. This is called Waiting State. Once a thread reaches waiting state, it will need to wait till some other thread notify() or notifyAll() on the object.

Once this thread is notified, it will not be runnable. It might be that other threads are also notified(using notifyAll()) or the first thread has not finished its work, so it is still blocked till it gets its chance. This is called Blocked State.

Once other threads have left and its this thread chance, it moves to Runnable state after that it is eligible pick up work based on JVM threading mechanism and moves to run state.

1. Difference between WAITING and TIMED\_WAITING?

Ans : A thread is in the waiting state due to calling one of the

following methods

Object.wait();

t.join();

and thread is in time waiting state when called any of these method with specific time

t.join(long)

Object.wait(long)

Thread.sleep(long)

1. What is interrupt() method?

This is a member method not a static method, so we can interrupt the specific thread

by calling t.interrupt().

The interrupt() method has two effects. First, it causes any blocked thread to throw an InterruptedException. As example the sleep() method is a blocking method. If any  thread interrupts the thread t , while that thread t is executing the sleep() method, the sleep method immediately wakes up and throws an InterruptedException. Other methods that behave this way include the wait() method, the join() method.

The interrupt mechanism is implemented using an internal flag known as the interrupt status. Invoking Thread.interrupt sets this flag. When a thread checks for an interrupt by invoking the static method Thread.interrupted, interrupt status is cleared. The non-static Thread.isInterrupted, which is used by one thread to query the interrupt status of another, does not change the interrupt status flag

1. **Different methods for interruption ?**

Ans :

void interrupt() :   
sends an interrupt request to a thread. The "interrupted" status of the thread is set to true. If the thread is currently blocked by a call to sleep or wait, an InterruptedException is thrown.

static boolean interrupted() :   
tests whether or not the *current* thread (that is, the thread that is executing this instruction) has been interrupted. Note that this is a static method. The call has a side effect—it resets the "interrupted" status of the current thread to false.

boolean isInterrupted() :   
tests whether or not a thread has been interrupted. Unlike the static interrupted method, this call does not change the "interrupted" status of the thread. It is a non static method

1. **Why wait() method should be called inside a while() loop ?**

Ans : The primary reason why while loops are so important is race conditions between threads. Certainly spurious wakeups are real and for certain architectures they are common, but race conditions are a much more likely reason for the while loop.

For example:

synchronized (queue) {

// this needs to be while

while (queue.isEmpty()) {

queue.wait();

}

queue.remove();

}

With the above code, there may be 2 consumer threads. When the producer locks the queue to add to it, consumer #1 may be blocked at the synchronized lock while consumer #2 is waiting on the queue. When the item is added to the queue and notify called by the producer, #2 is moved from the wait queue to be blocked on the queue lock, but it will be *behind* the #1 consumer which was already blocked on the lock. This means that the #1 consumer gets to go forward first to call remove() from the queue. If the while loop is just an if, then when consumer #2 gets the lock after #1 and calls remove(), an exception would occur because the queue is now empty -- the other consumer thread already removed the item. Just because it was notified, it needs to be make sure the queue is still empty because of this race condition.

1. **What is threadgroup ?**

When similar kind of threads are grouped to perform some similar operation.

Assign a thread group like

ThreadGroup tg = new ThreadGroup(“Tg1”)

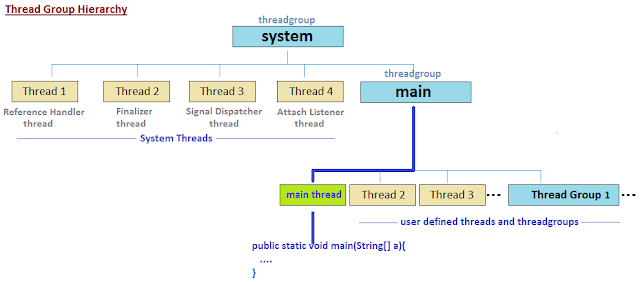
Threadt1 = new Thread (tg, new Runnable(), threadname)

1. **Advantages of thread group ?**

interrupt() , setDaemon(), isInterruptred(), isDaemon()

activeCount(), setPriority(), getMaxPriority(), exception handling by overriding unCaughtException()

1. **Thread group hierarchy structure**



1. **Structure of Threadpool**

Ans :

ExecutorService ex = Executors.*newFixedThreadPool*(5);

5 = corepoolsize

5 = maxpoolsize

In this case corepool size and maxpoolsize is same, that’s why it is fixed pool

This is the actual return statement

**return** **new** ThreadPoolExecutor(nThreads, nThreads,

0L, TimeUnit.***MILLISECONDS***,

**new** LinkedBlockingQueue<Runnable>());

1st and the 2nd arguments are corepoolsize and maxpoolsize i.e. 5 in this example

3rd is keepalive i.e. by default set as 0 for fixed pool

Last argument is the workingqueue it is the blocking queue where threads are being queued

1. **What is Keep-alive times ?**

Ans : If the pool currently has more than corePoolSize threads,

excess threads will be terminated if they have been idle for more

than the keepAliveTime. This provides a means of reducing resource consumption when the

pool is not being actively used. If the pool becomes more active

later, new threads will be constructed. By default, the keep-alive policy

applies only when there are more than corePoolSize threads.

1. **What is core threads?**

Ans : By default, even core threads are initially created and started only when new tasks arrived. core threads is the minimum which is always running just in case you want to pass it a task.

The cached pool by default has a core of 0 as you might expect.

For the fixed thread pool, the core and the maximum are the same i.e. whatever you set the fixed size to.

1. **How the queuing happens in thread pool ?**

Ans :

* If fewer than corePoolSize threads are running, the Executor always prefers adding a new thread rather than queuing
* If corePoolSize or more threads are running, the Executor always prefers queuing a request rather than adding a new thread
* If a request cannot be queued, a new thread is created unless this would exceed maximumPoolSize, in which case, the task will be rejected.

1. **Which tasks are rejected ?**

Ans : New tasks submitted in method #execute(Runnable) of ThreadPoolExecutor will be

Rejected when the Executor has been shut down, and also when the Executor uses finite bounds for both maximum threads and work queue capacity, and is saturated. In either case, the execute method invokes the RejectedExecutionHandler#rejectedExecution(Runnable, ThreadPoolExecutor)

By default ThreadPoolExecutor.AbortPolicy is used and the handler throws a runtime RejectedExecutionException upon rejection.

1. **What are the different running states in threadpoolexecutor ?**

Ans :

RUNNING: Accept new tasks and process queued tasks

\* SHUTDOWN: Don't accept new tasks, but process queued tasks

\* STOP: Don't accept new tasks, don't process queued tasks, and interrupt in-progress tasks

\* TIDYING: All tasks have terminated, workerCount is zero, the thread transitioning to state TIDYING will run the terminated() hook method

\* TERMINATED: terminated() has completed

1. **What are the different process of executing a Callable or a Runnable task?**

Ans :

* ExecutorService ex = Executors.*newFixedThreadPool*(5);

ex.execute(runnable object) , it is a void method

newFixedThreadPool actually returns the object of ThreadPoolExecutor class

* ex.invokeAll(collection of callable object)

Internally it warps up each callable object with FutureTask object which implements RunnableFuture interface , and RunnableFuture interface extends Runnable and Future class. Here Callable, Future, RunnableFuture all are generic class and interfaces. After wrapping up call the same execute method on

Each Futuretask obejcts. This method returns List<Future<T>>

There is one more method ex.invokeAny(collection of Callable objects) which will return the T object only.

* ex.submit(Callable object) or ex.submit(Runnable object)

This will again wrap up both callable and runnable object to Futuretask object and call execute method. This method returns Future<T>

* One more process is to wrap up the callable objects or runnable objects from outside and pass it to execute method

1. What does the future.get() method returns ?

Ans : for callable it will return the type what a call method of callable returns , but for runnable object it will return null as run() method is a void method.

1. How **public** **void** execute(Runnable command) runs?

Ans :

The executor keep creating thread of corepoolsize , when first task came it will pull one of the already created task and execute, otherwise if no of threads submitted is greater than the corepoolsize then excess threads will be

put into the blockingqueue till the blockingqueue is full. When one of the slot is empty then the runnable task is allocated and executed, if the waiting time of the runnable task is more than keppalive time, then the task will be rejected with rejection handler.

1. **What shutdown do?**

Ans : We need the shutdown method just to indicate the client to stop submitting or executing more tasks, so if we don’t call the shutdown method it continuously create the threads in its pool.

It first lock the mechanism

Check whether any tasks are running or not if running then complete them

Interrupt or threads which are submitted after shutdown() method call.

Change the runstate to SHUTDOWN as mentioned before

Release the lock

1. **What is forkjoin pool ?**

Ans : forkjoin pool is the entry point for Forkjointask classes like RecursiveAction<V> and RecursiveTask<Void>. It is like executorservice , we need to submit, invoke or execute the Forkjointask as tasks.

The worker threads which are created to process the tasks are all set as daemon thread. It depends on work stealing algorithm

1. **What is work stealing algorithm ?**

Ans : Each worker thread maintains runnable tasks in its own scheduling queue. Queues are maintained as double−ended queues (i.e., deques, usually pronounced "decks"), supporting both LIFO push and pop operations, as well as a FIFO take operation. Worker threads process their own deques in LIFO (youngest−first) order, by popping tasks. When a worker thread has no local tasks to run, it attempts to take ("steal") a task from another randomly chosen worker, using a FIFO (oldest first) rule. When a worker thread encounters a join operation, it processes other tasks, if available, until the target task is noticed to have completed (via isDone). All tasks otherwise run to completion without blocking.

Result solve(Problem problem) {

if (problem is small)

directly solve problem

else {

split problem into independent parts

fork new subtasks to solve each part

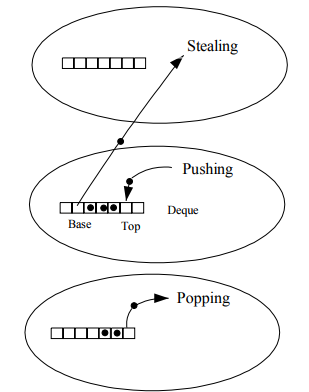
join all subtasks

compose result from subresults

}

}

Those forked subtasks (line 2 in the else block) can recursively create more subtasks themself and thus fill up the working queues of the parallely working threads. If one thread finished and has nothing more to do, he can "steal" the work from the queue of another thread. Follow the below figure



1. **Mention few parameters which are used in forkjoinpool**

Ans :

Parallelism : We have number of cores in cpu like cpu 0, cpu1 etc.

But all of them are not used , in forkjoin pool we can use all of them at same time.

We can define the parallelism count in the constructor.

Suppose if we have 4 cpu , then default parallelism is 4, if we set this to 2 then the forkjoin pool will use only 2 cpus , but if its more than the 4 say 6, then one cpu can have more than one thread.

Steal count : the total number of tasks stolen from one thread's work queue by another.

1. **Difference between ForkJoinPool and Executorservice ?**

Ans : The main difference between the Fork/Join and the Executor frameworks is the work-stealing algorithm. Unlike the Executor framework, when a task is waiting for the finalization of the sub-tasks it has created using the join operation, the thread that is executing that task (called worker thread ) looks for other tasks that have not been executed yet and begins its execution. By this way, the threads take full advantage of their running time, thereby improving the performance of the application.

1. Reference for programming

<https://howtodoinjava.com/java-7/forkjoin-framework-tutorial-forkjoinpool-example/>