TITLE

Laboratory 9. Thread Pools

PREREQUISITES

- OOP basic knowledge
- Java programming language basic knowledge
- Algorithms

RESOURCES

- Course slides
- Java Tutorial
- Java Concurrency

LABORATORY INSTRUCTIONS

A **BlockingQueue** is a queue that blocks when the dequeuing operation is applied when the queue is empty or the enqueue operation is applied but the queue is already full.
A thread trying to dequeue is blocked until some other thread inserts an item into the queue. A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely.

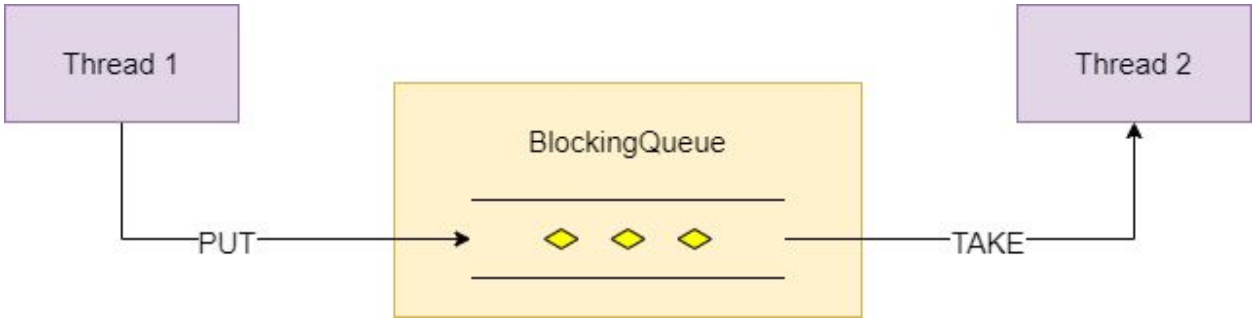

Fig. 1 BlockingQueue Diagram

```
class LockBasedQueue<T> {
   volatile int head, tail;
   T[] items;
   Lock lock;

   public LockBasedQueue(int capacity) {
```

```
        head = 0;
        tail = 0;
        lock = new ReentrantLock();
        items = (T[]) new Object[capacity];
    }

    public T deq() throws EmptyException {
        lock.lock();
        try {
            if (tail == head)
                throw new EmptyException();
            T x = items[head % items.length];
            head++;
            return x;
        } finally {
            lock.unlock();
        }
    }

    public void enq(T x) throws FullException {
        lock.lock();
        try {
            if (tail - head == items.length)
                throw new FullException();
            items[tail % items.length] = x;
            tail++;
        } finally {
            lock.unlock();
        }
    }
}
```

Code 1 - Concurrent queue example from the course slides

```
class WaitFreeQueue<T> {
    volatile int head = 0, tail = 0;
    T[] items;
    public WaitFreeQueue(int capacity) {
        items = (T[]) new Object[capacity];
    }
    public void enq(T x) throws FullException {
        if (tail - head == items.length)
            throw new FullException();
        items[tail % items.length] = x;
        tail++;
    }
    public T deq() throws EmptyException {
        if (tail - head == 0)
            throw new EmptyException();
```

```
        T x = items[head % items.length];
        head++;
        return x;
    }
}
```

Code 2 - Queue implementation without locks

A **thread pool** is a pool of threads that can be reused to execute more than one task. This is an alternative to creating a new thread for each task that needs to be executed.
Creating a new thread is way more resources consuming than reusing a thread that is already created. Also, by using thread pools we have more control on how many threads are active at a time. It is well known that each thread is consuming a certain amount of computer resources, for example RAM.

The thread pool principle works like this:
Instead of creating a new thread for every task to execute concurrently, the task is passed to a thread pool. The task is enqueued in a BlockingQueue from where the threads in the pool are dequeuing from and assigned to an idle thread. The rest of the idle threads will wait for other tasks to be enqueued in the BlockingQueue.

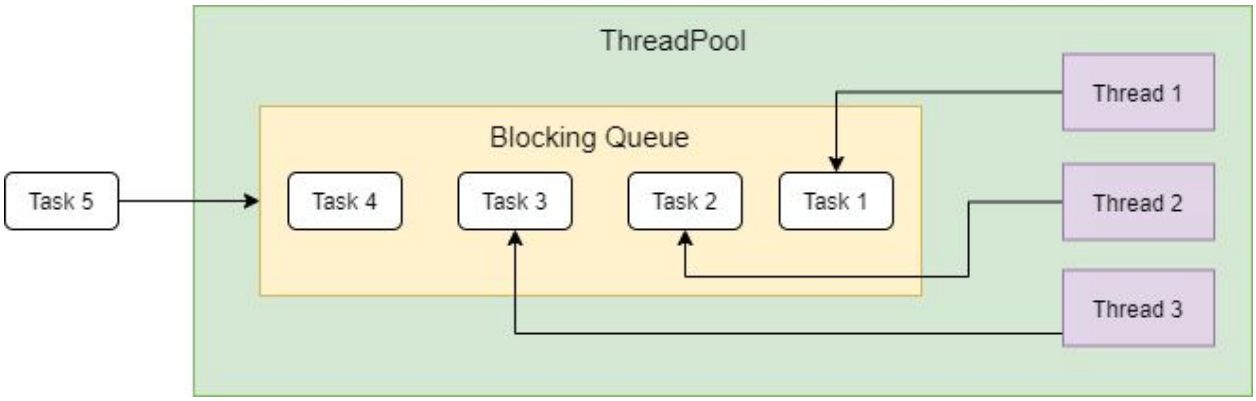

Fig. 2 ThreadPool Diagram

A thread pool use case can be found in multithreaded servers where each new connection is wrapped as a task and passed on a thread pool. The available threads in the thread pool will process the requests on the connection concurrently.

In Java there is an already built in thread pool implementation in ***java.util.concurrent*** package, but for the understanding of the concept, here is a simple thread pool implementation:

```
public class ThreadPool {

    private BlockingQueue taskQueue = null;
```

```
    private List<PoolThreadRunnable> runnables = new ArrayList<>();
    private boolean isStopped = false;

    public ThreadPool(int noOfThreads, int maxNoOfTasks){
        taskQueue = new ArrayBlockingQueue(maxNoOfTasks);

        for(int i=0; i<noOfThreads; i++){
            PoolThreadRunnable poolThreadRunnable =
                    new PoolThreadRunnable(taskQueue);

            runnables.add(new PoolThreadRunnable(taskQueue));
        }
        for(PoolThreadRunnable runnable : runnables){
            new Thread(runnable).start();
        }
    }

    public synchronized void  execute(Runnable task) throws Exception{
        if(this.isStopped) throw
                new IllegalStateException("ThreadPool is stopped");

        this.taskQueue.offer(task);
    }

    public synchronized void stop(){
        this.isStopped = true;
        for(PoolThreadRunnable runnable : runnables){
            runnable.doStop();
        }
    }

    public synchronized void waitUntilAllTasksFinished() {
        while(this.taskQueue.size() > 0) {
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

}

public class PoolThreadRunnable implements Runnable {

    private Thread        thread    = null;
    private BlockingQueue taskQueue = null;
    private boolean       isStopped = false;
```

```
    public PoolThreadRunnable(BlockingQueue queue){
        taskQueue = queue;
    }

    public void run(){
        this.thread = Thread.currentThread();
        while(!isStopped()){
            try{
                Runnable runnable = (Runnable) taskQueue.take();
                runnable.run();
            } catch(Exception e){
                //log or otherwise report exception,
                //but keep pool thread alive.
            }
        }
    }

    public synchronized void doStop(){
        isStopped = true;
        //break pool thread out of dequeue() call.
        this.thread.interrupt();
    }

    public synchronized boolean isStopped(){
        return isStopped;
    }
}

public class ThreadPoolMain {

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

        ThreadPool threadPool = new ThreadPool(3, 10);

        for(int i=0; i<10; i++) {

            int taskNo = i;
            threadPool.execute( () -> {
                String message =
                        Thread.currentThread().getName()
                                + ": Task " + taskNo ;
                System.out.println(message);
            });
        }

        threadPool.waitUntilAllTasksFinished();
        threadPool.stop();

    }
```

```
}
```

Code 3 - Simple ThreadPool Implementation using Java BlockingQueue

This thread pool implementation has two components:

- **ThreadPool** class is the representation of the thread pool
- **PoolThread** class which implement the threads that execute the tasks

A task is executed by calling the method `ThreadPool.execute(Runnable r)` with a Runnable implementation as parameter. The task is then enqued in the blocking queue internally waiting to be dequeued.

An idle PoolThread will dequeue the Runnable implementation and execute it. This can be viewed in the PoolThread's `run()` method. After the execution the pool thread loops and tries to process another task.

LABORATORY TASKS

1. Create a new Java project and implement the simple Thread Pool example above.