Thread Pools
(Worker Queues)

What is a Thread Pool?
• A collection of threads that are created once (e.g. when a server starts)
• That is, no need to create a new thread for every client request
• Instead, the server uses an already prepared thread if there is a free one, or waits until there is a free thread

Why Do We Need a Thread Pool?

Why Using Thread Pools?
• Thread pools improve resource utilization
  • The overhead of creating a new thread is significant
  • Thread pools enable applications to control and bound their thread usage
  • Creating too many threads in one JVM can cause the system to run out of memory and even crash
  • There is a need to limit the usage of system resources such as connections to a database

Thread Pools in Servers
• Thread pools are especially important in client-server applications
  • The processing of each individual task is short-lived and the number of requests is large
  • Servers should not spend more time and consume more system resources creating and destroying threads than processing actual user requests
  • When too many requests arrive, thread pools enable the server to force clients to wait until threads are available

The “Obvious” Implementation
• There is a pool of threads
• Each task asks for a thread when starting and returns the thread to the pool after finishing
• When there are no available threads in the pool the thread that initiates the task waits till the pool is not empty
• What is the problem here?

"Synchronized" model - the client waits until the server takes care of its request.
The “Obvious” Implementation is Problematic

- When the pool is empty, the submitting thread has to wait for a thread to be available
- We usually want to avoid blocking that thread
- A server may want to perform some actions when too many requests arrive
- Technically, Java threads that finished running cannot run again

A Possible Solution

Every thread looks for tasks in the queue

wait()

If Q is Empty

All the worker threads wait for tasks

A Possible Solution

The number of worker threads is fixed. When a task is inserted to the queue, notify is called

notify()

A Possible Solution

The task is executed by the thread

Task Queue

Worker Threads

The remaining tasks are executed by the other threads
When a task ends, the thread is released

While the Q is not empty, take the task from the Q and run it (if the Q was empty, wait() would have been called)

When a task ends, the thread is released

A new task is executed by the released thread

A Possible Solution

Thread Pool Implementation

```java
public class TaskManager {
    private LinkedList taskQueue = new LinkedList();
    private List threads = new LinkedList();
    public TaskManager(int numThreads) {
        for (int i = 0; i < numThreads; ++i) {
            Thread worker = new Worker(taskQueue);
            threads.add(worker);
            worker.start();
        }
    }
    public void execute(Runnable task) {
        synchronized (taskQueue) {
            if (taskQueue.isEmpty()) {
                try {
                    taskQueue.wait();
                } catch (InterruptedException ignored) {
                }
            }
            taskQueue.addLast(task);
            taskQueue.notify();
        }
    }
}
```

Thread Pool Implementation

```java
public class Worker extends Thread {
    private LinkedList taskQueue;
    public Worker(LinkedList queue) {
        taskQueue = queue;
    }
    public void run() {
        Runnable task = null;
        while (true) {
            synchronized (taskQueue) {
                while (taskQueue.isEmpty()) {
                    try {
                        taskQueue.wait();
                    } catch (InterruptedException ignored) {
                    }
                }
                task = (Runnable) taskQueue.removeFirst();
                task.run();
            }
        }
    }
}
```

Risks in Using Thread Pools

- Threads can leak
  - A thread can endlessly wait for an I/O operation to complete
    For example, the client may stop the interaction with the socket without closing it properly
  - What if `task.run()` throws a runtime exception (as opposed to other exceptions that a programmer of a client application has to catch in order to succeed compiling)?
- Solutions:
  - Bound I/O operations by timeouts using `wait(time)`
  - Catch possible runtime exceptions

Pool Size

- What is better: to have a large pool or a small pool?
- Each thread consumes resources
  - memory, management overhead, etc.
  - A large pool can cause starvation
- Incoming tasks wait for a free thread
  - A small pool can cause starvation
- Therefore, you have to tune the thread pool size according to the number and characterizations of expected tasks
- There should also be a limit on the size of the task queue (why?)
Handling too Many Requests

- What is the problem with the server being overwhelmed with requests?
- What can a server do to avoid a request overload?
  - Do not add to the queue all the requests: ignore or send an error response
  - Use several pool sizes alternately according to stress characteristics (but do not change the size too often...)

Tuning the Pool Size

- The main goal: Processing should continue while waiting for slow operations such as I/O
- $WT = \text{estimated average waiting time}$
- $ST = \text{estimated average processing time for a request (without the waiting time)}$
- About $WT/ST + 1$ threads will keep the processor fully utilized
- For example, if $WT$ is 20 ms and $ST$ is 5 ms, we will need 5 threads to keep the processor busy

The java.util.concurrent Package

- Provides a more advanced mechanisms for handling concurrency (Since Java 5.0)
- Includes an implementation of thread pools

Lock

- Synchronized sections are like a trap – once entered, the thread is blocked till ...
- Lock objects provide the ability to check the availability of the lock and back out, if desired
- When using Lock objects, lock() and unlock() are called explicitly

Executor

- The class Executors has 2 static methods to create thread pools
  - ExecutorService newFixedThreadPool(int nThreads)
    - Pool of a fixed size
  - ExecutorService newCachedThreadPool()
    - Creates new threads as needed
    - New threads are added to the pool, and recycled
- ExecutorService has an execute method
  - void execute(Runnable command)

```java
class NetworkService {
    private final ServerSocket serverSocket;
    private final ExecutorService pool;

    public NetworkService(int port, int poolSize) throws IOException {
        serverSocket = new ServerSocket(port);
        pool = Executors.newFixedThreadPool(poolSize);
    }

    public void serve() {
        try {
            for (;;) {
                pool.execute(new Handler(serverSocket.accept()));
            }
        } catch (IOException ex) {
            pool.shutdown();
        }
    }
}
```
class Handler implements Runnable {
    private final Socket socket;

    Handler(Socket socket) {
        this.socket = socket;
    }

    public void run() {
        // read and service request
    }
}