ENSF 607

Introduction to Thread Pools (Worker Queues)

Thread Pool

- Collection of threads that are **created when the server starts** (**i.e. created only once**)

- No need to create a new thread for every client request

- Instead, the server uses an **already existing thread** if there is a **free one**, or **waits** until there is a free thread

Why 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 possibly crash
  - There is a need to limit the usage of system resources such as connections to a database

Using Thread Pools in Servers

- Thread pools are particularly important in client-server applications
  - Processing of each individual task is short-lived and **there is a large number of requests**
  - Servers should **NOT** consume spend more time/system resource in creating and destroying threads, than processing actual user requests
- If **too many requests arrive**, thread pools allow the server to force clients to **wait** until threads are available

Implementation

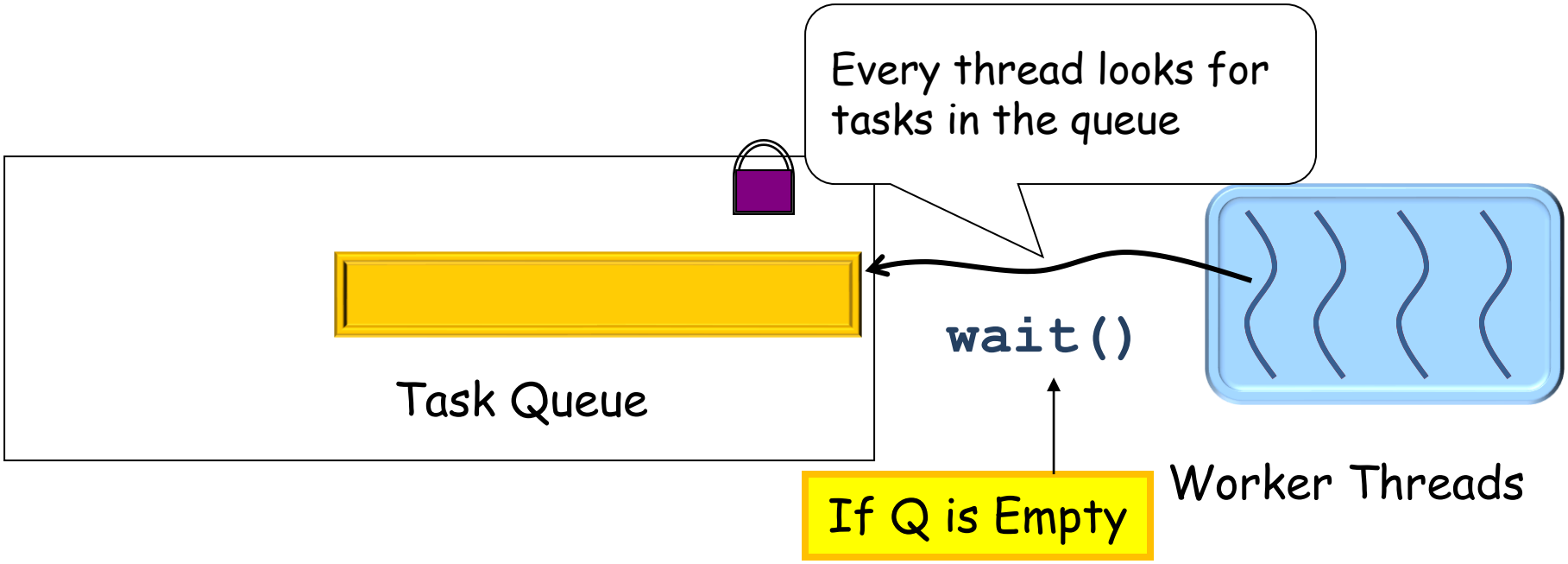
- There is a pool of threads
- Each task asks for a thread when starting, and returns the thread to the pool after finishing
- When there are no available threads in the pool, the thread that initiates the task waits till the pool is not empty
- What is the problem here?

"Synchronized" model - the client waits until the server takes care of its request…

The “Obvious” Implementation is Problematic

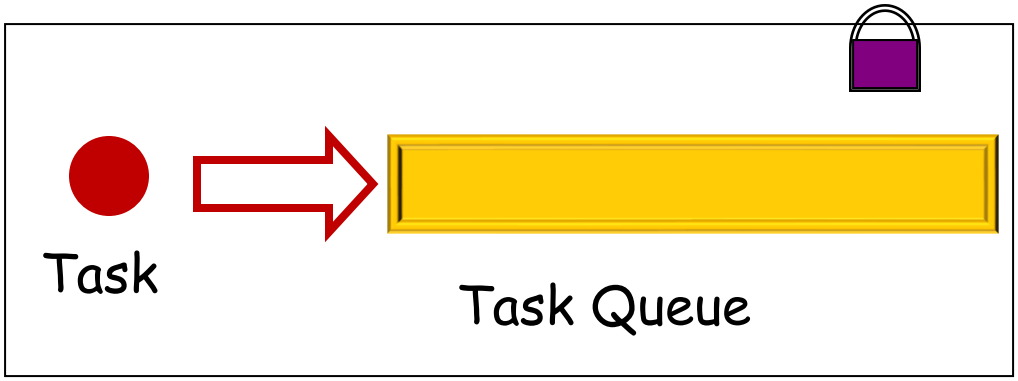
- When the pool is empty, the submitting thread has to wait for a thread to be available
  - We usually want to avoid blocking that thread
  - A server may want to perform some actions when too many requests arrive
- Technically, Java threads that finished running cannot run again

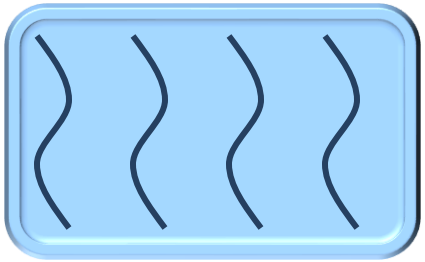
A Possible Solution



All the worker threads wait for tasks

A Possible Solution



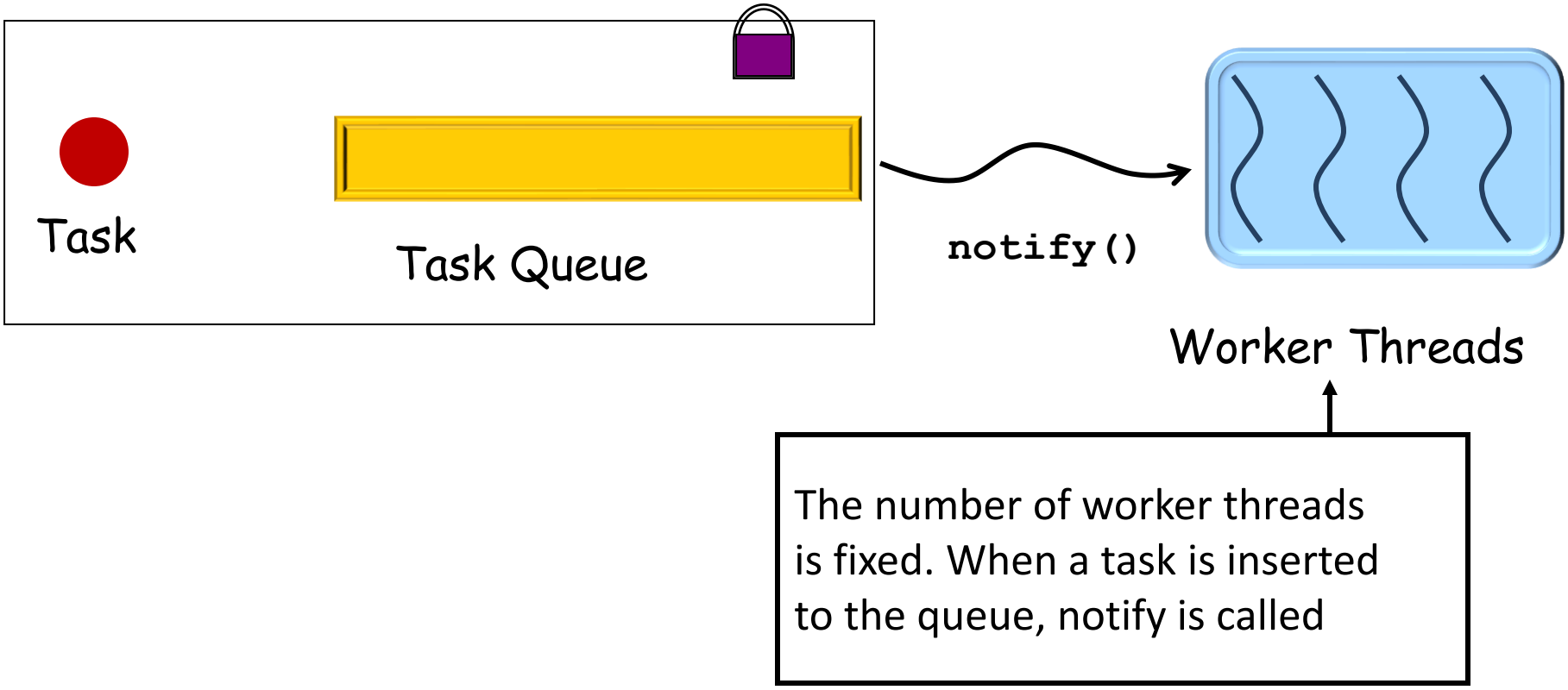


Worker Threads

"A-synchronized" model:
"Launch and forget"

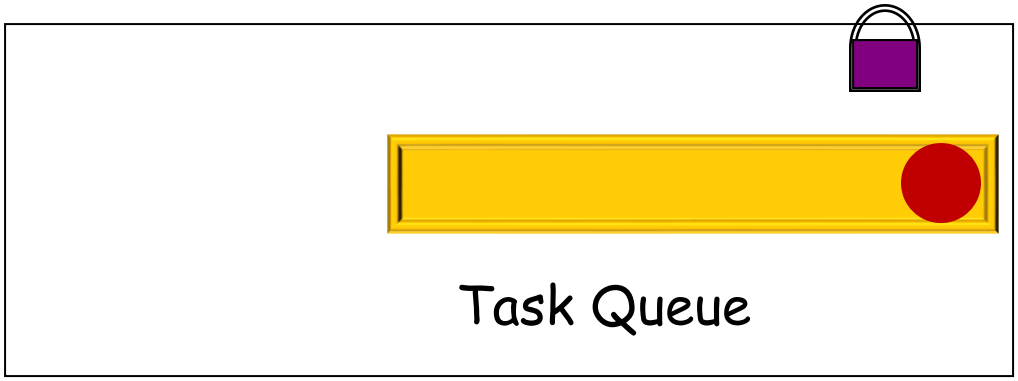
The number of worker threads is fixed. When a task is inserted to the queue, notify is called

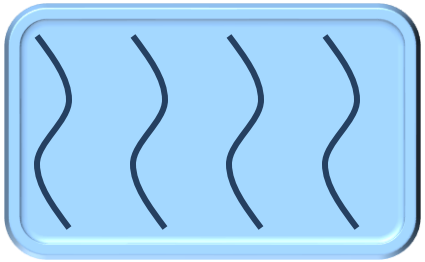
A Possible Solution



A Possible Solution

The task is executed by the thread

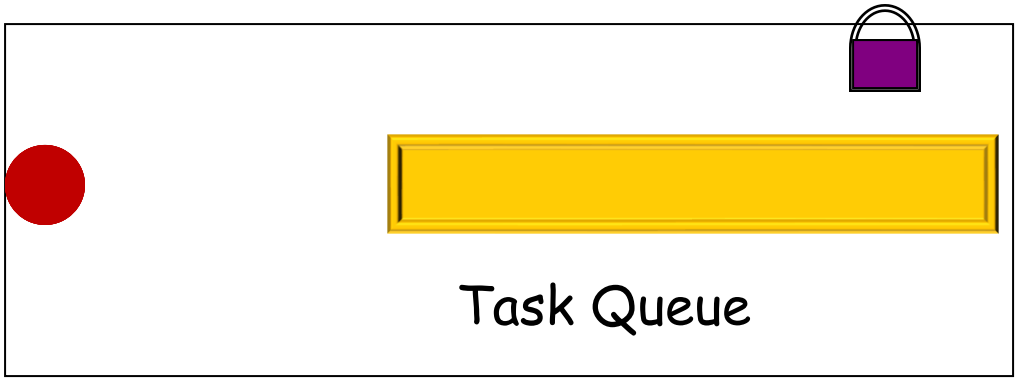


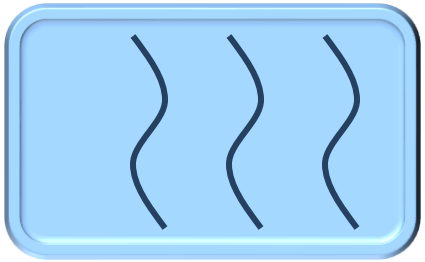


Worker Threads

A Possible Solution

The task is executed by the thread



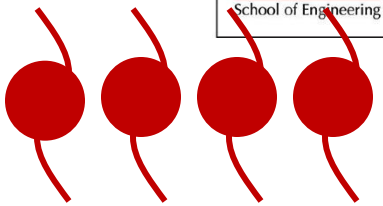


Worker Threads

The remaining tasks are executed by the other threads

A Possible Solution

When a task ends, the thread is released



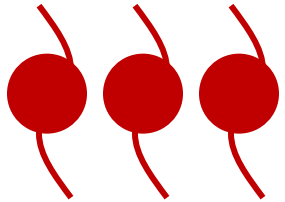
Task Queue

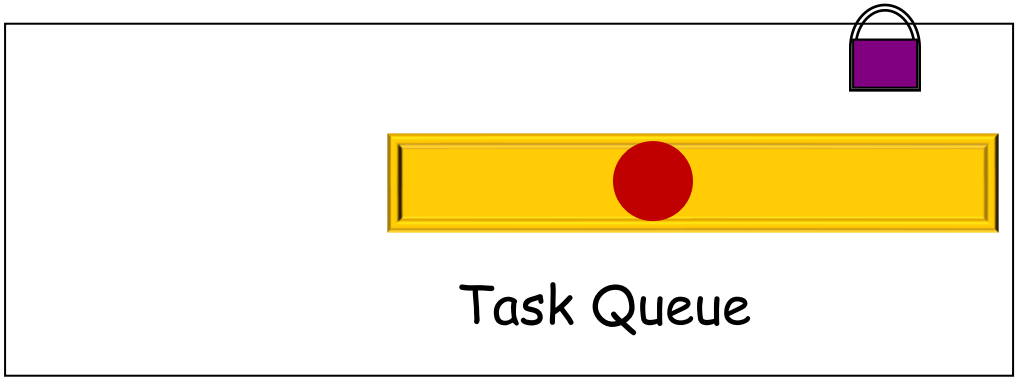
Worker Threads

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)

A Possible Solution

A new task is executed
by the released thread





Worker Threads

Thread Pool Implementation

```
public class TaskManager {
   LinkedList taskQueue = new LinkedList();
   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) {
         taskQueue.addLast(task);
         taskQueue.notify();
} } }
```

Thread Pool Implementation

```
public class Worker extends Thread {
    LinkedList taskQueue = null;
    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 = estimated average waiting time
- ST = 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

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)

```
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 }
}
```