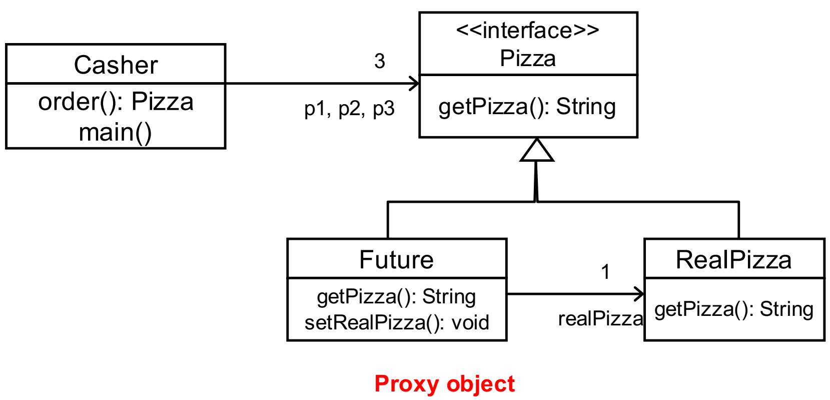
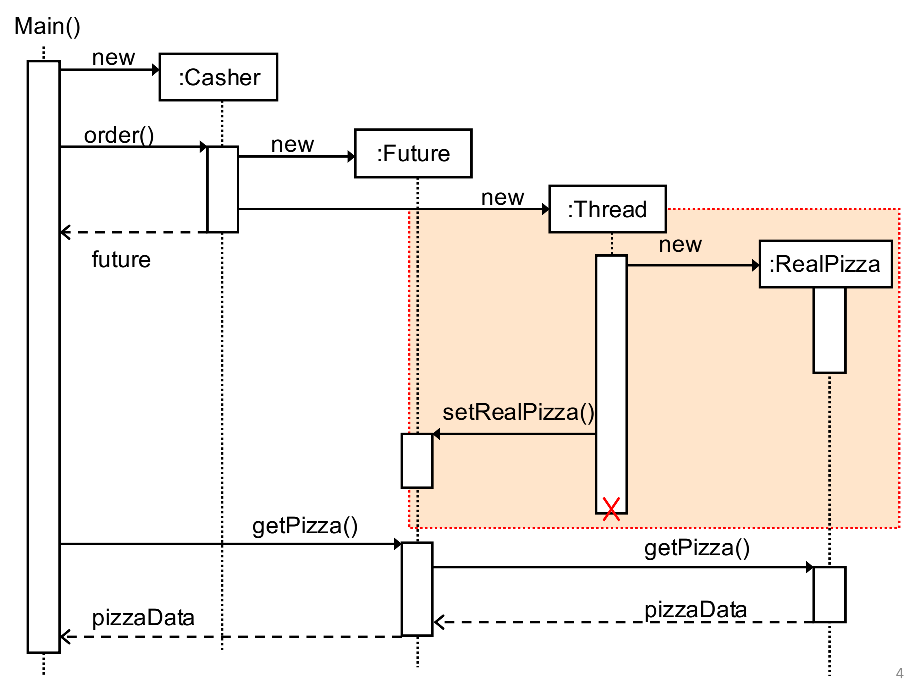
Future Design Pattern

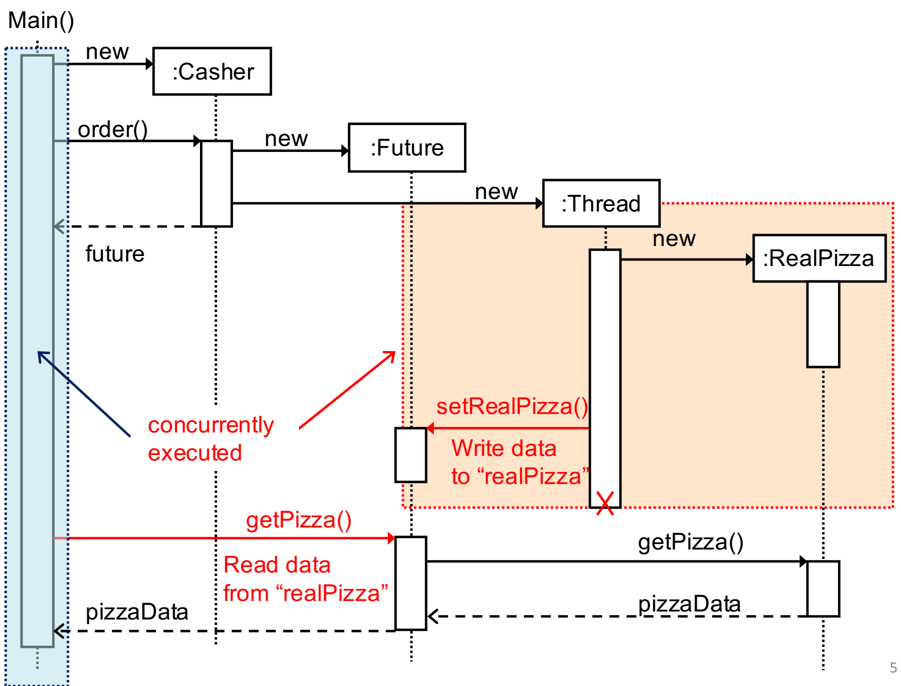
Futures

- Futures contract
  - An agreement traded on an organized exchange to buy or sell assets (commodities or shares) at a fixed price but ***to be delivered and paid for later***.
- At a fast food (e.g. pizza) store…

Sample Code using *Future*







Future

```
public class Future implements Pizza{
    private RealPizza realPizza = null;
    private ReentrantLock lock;
    private Condition ready;

    public Future(){
        lock = new ReentrantLock();
        ready = lock.newCondition(); }

    public void setRealPizza( RealPizza real ){
        lock.lock();
        if( realPizza != null ){ return; }
        realPizza = real;
        ready.signalAll();
        lock.unlock(); }

    public String getPizza(){
        String pizzaData = null;
        lock.lock();
        if( realPizza == null ){
            ready.await();
        }
        pizzaData = realPizza.getPizza();
        lock.unlock(); }}
```

Why using a Lock in Future?

Main thread
getPizza()

If(realPizza == null) -->
true

Reaches the end of
its time slice

Gain a time slice

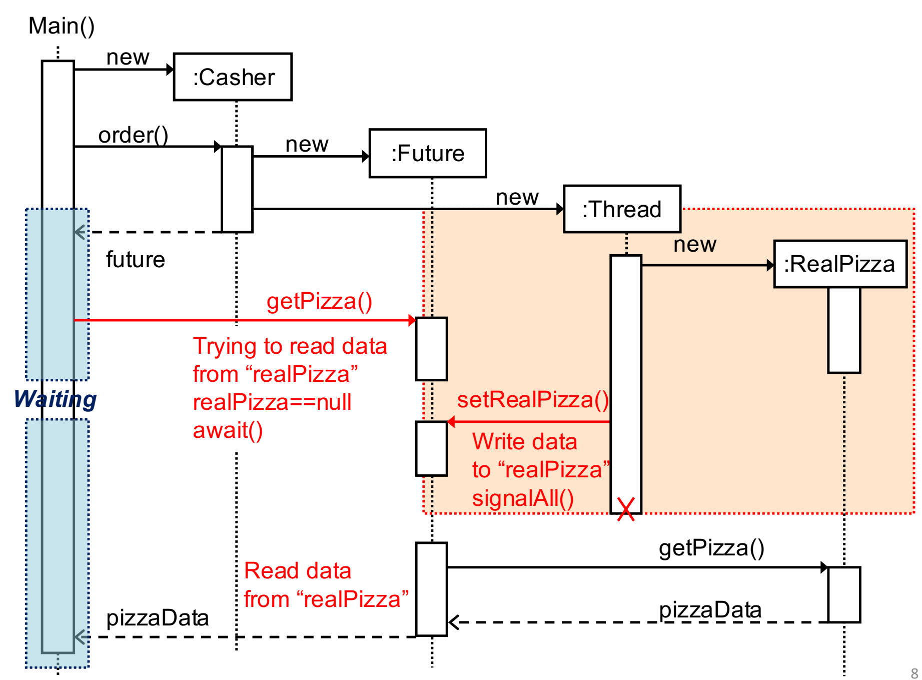
Believes realPizza == null. ***Race condition!***
Calls await(). No threads will call signalAll(). ***Deadlock!***

Another thread
setRealPizza()

Gain a time slice

realPizza = …;

run() returns. Thread termination
realPizza != null



Future

```
public class Future implements Pizza{
   private RealPizza realPizza = null;
   private ReentrantLock lock;
   private Condition ready;

   public Future(){
      lock = new ReentrantLock();
      ready = lock.newCondition();  }

   public void setRealPizza( RealPizza real ){
      lock.lock();
      if( realPizza != null ){ return; }
      realPizza = real;
      ready.signalAll();
      lock.unlock(); }

   public String getPizza(){
      String pizzaData = null;
      lock.lock();
      if( realPizza == null ){
         ready.await();
      }
      pizzaData = realPizza.getPizza();
      lock.unlock(); }}
```

(2) Kitchen thread: signalAll().

(1) Customer (main): thread: Goes to "waiting" and temporarily releases the lock if the pizza is not ready.

(3) Customer thread: Goes to "runnable" Acquire the lock again if it is available. If it is not available, goes to "blocked."

An Example Output

- Output
  – Ordering pizzas at a casher counter.
  – An order is made.
  – An order is made.
  – An order is made.
  – Doing something, reading newspapers, magazines, etc., until pizzas are ready to pick up...
  – A real pizza is made!
  – A real pizza is made!
  – A real pizza is made!
  – Let's see if pizzas are ready to pick up...
  – REAL PIZZA!
  – REAL PIZZA!
  – REAL PIZZA!

"If" v.s. "while"

```
public class Future implements Pizza{
   private RealPizza realPizza = null;
   private ReentrantLock lock;
   private Condition ready;

   public Future(){
      lock = new ReentrantLock();
      ready = lock.newCondition();  }

   public void setRealPizza( RealPizza real ){
      lock.lock();
      if( realPizza != null ){ return; }
      realPizza = real;
      ready.signalAll();
      lock.unlock(); }

   public String getPizza(){
      String pizzaData = null;
      lock.lock();
      if( realPizza == null ){ // IF v.s. WHILE
         ready.await();
      }
      pizzaData = realPizza.getPizza();
      lock.unlock(); }}
```

(2) Kitchen thread:

(1) Customer (main): thread:

- A while loop should be used, not a if statement, if...
  – there exist more than one "customer" threads.
    - C.f. lecture note #8

HW 18

- Casher::order() uses an anonymous thread class

```
– public Pizza order(){
    new Thread(){
        public void run(){
            RealPizza realPizza = new RealPizza();
            future.setRealPizza( realPizza );
        }
    }.start();
```

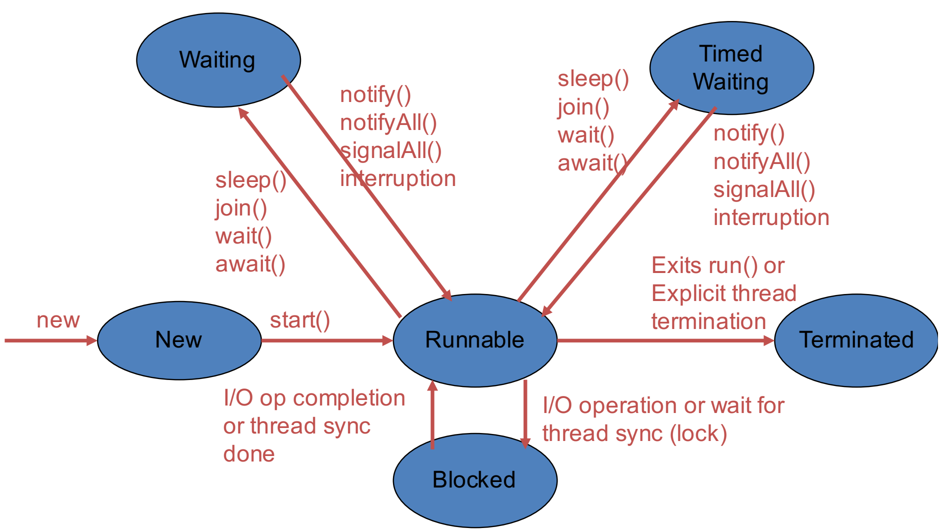
- Modify the red code to use a lambda expression.
  - Runnable is a functional interface that has only one method, run().

```
– new Thread( ()->{…} ).start();
```

- Implement isReady() in Future
  - `public boolean isReady()`
    - Returns true if the real pizza is ready to pick up; otherwise, returns false.
    - Modify main() to use isReady()

```
– while(true){
    lock.lock();
    if( future.isReady() ){
      future.getPizza(); break;   }
    lock.unlock();
    System.out.println("Doing something”);   }
```

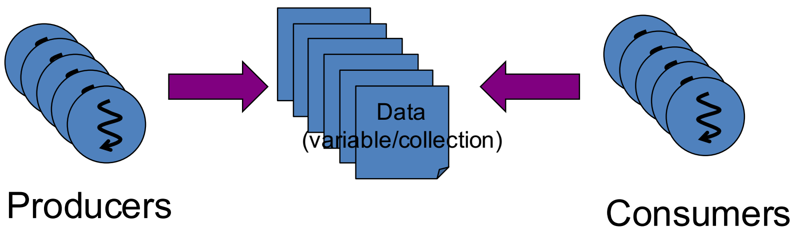
- Implement getPizza(long timeout) in Future
  - `String getPizza(timeout: long) throws CasherTimeoutException`
    - Wait (with await()) for up to a timeout period (in milliseconds) if the real pizza is not ready
      - Timeout is the maximum time to wait.
    - Throws an TimeoutException when a timeout occurs.
      - Currently, Future::getPizza() blocks until a real pizza becomes ready.



Producer-Consumer Design Pattern

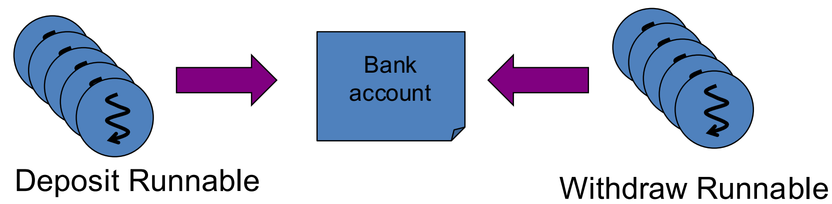
Producer-Consumer Design Pattern

- Producer
  - One or more threads generate data to be processed.
- Consumer
  - One or more threads take and process those data.
  - If no data is available, a consumer(s) wait until a producer generates required data.



Bank Account Example Code

- DepositRunnable (Producer)
  - A thread (or a group of threads) that deposits money to a bank account.
  - If the current balance is over the upper limit, the thread(s) wait(s) until the balance goes below the upper limit.
- WithdrawRunnable (consumer)
  - A thread (or a group of threads) that withdraws money from the account.
  - If the current balance is below a lower limit, the thread(s) wait(s) until the balance exceeds the lower bound.



Future as Producer-Consumer Design Pattern

- Kitchen (producer)
  - Generates the real pizza and sets it to a Future
- Customer (consumer)
  - Takes the real pizza if it is available to pick up.
  - Waits for it until it becomes available.

Exercise: Desktop Search

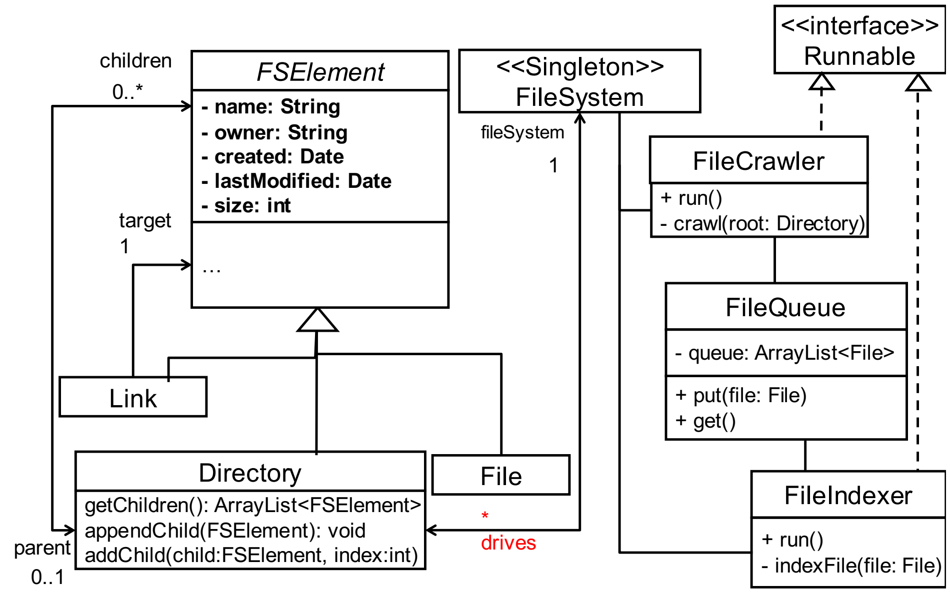
- Imagine an indexing service for a file system
  - e.g., Windows indexing service and Mac/iOS's Spotlight
- Key functions
  - Scan/crawl files in the local file system
  - Index those files for later searching.
    - Keep each file's metadata
      - Metadata: file's attributes (e.g., location, name, file type, author) and file's content

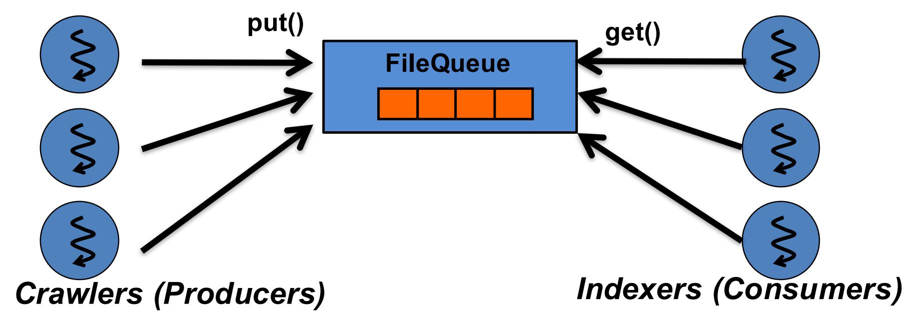
Threads in Desktop Search

- Single threaded
  - Use a single thread for both crawling and indexing
- Multi-threaded
  - Use different threads for crawling and indexing
    - One crawling thread and one indexing thread
    - Multiple crawling threads and multiple indexing threads
  - More efficient than the single-threaded version in multi-core environments
  - Crawlers and indexers interact with each other based on the Producer-Consumer design pattern.

```
• class FileCrawler implements Runnable{
    private Directory dir; //root dir of a given drive (tree structure)
    private FileQueue queue;
    ...
    public void run(){
      crawl(root);
      ...
    }
    private void crawl(Directory root){
      // crawl a given tree structure
      // put files to a queue
    }
  }
• class FileIndexer implements Runnable{
    private FileQueue queue;
    ...
    public void run(){
      while(true){
        indexFile( queue.get() );
      }
    }
    public indexFile(File file){
      // index a given file.
    }
  }
```

HW 19: Implement this.





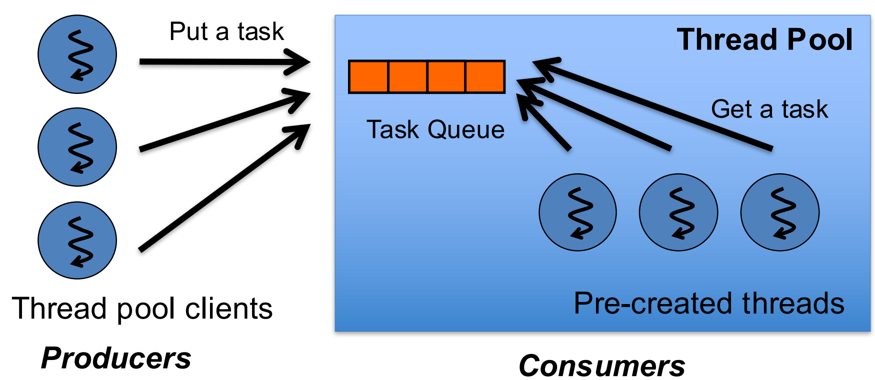
- Assume multiple crawler threads and multiple indexer threads.
  - One thread per a drive.
- A crawler thread
  - traverses a tree structure in a given drive and puts files to the queue.
  - waits, if the queue's size reaches a certain number, until the size becomes below that number.
  - dies when it finishes up traversing a given tree structure or when the main thread tells it to die.
- An indexer thread
  - keeps gets a file from the queue and indexes it.
    - Waits, if no files are available in the queue, until a crawler puts a new file.
  - repeats this forever until the main thread tells it to die.

- No need to include a GUI/CUI
- No need to filter files in crawl()
  - crawl() can queue all files to the file queue
- No need to implement actual indexing logic.
  - indexFile() can just print out each file's metadata on the shell.
- Make multiple drives and assign a crawler thread to each drive.
- Run multiple crawler threads and multiple indexing threads from main().
- Have the main thread stop crawler and indexing threads.
  - Crawler and indexer threads repeatedly checks a flag to see if they should stop and die.
  - The main thread flips the flag (flag-based thread termination)
  - Use a lock to guard the flag.
  - The main thread calls interrupt() on all crawler and indexing threads.
    - in case they are in the waiting state due to await().

Thread Pooling

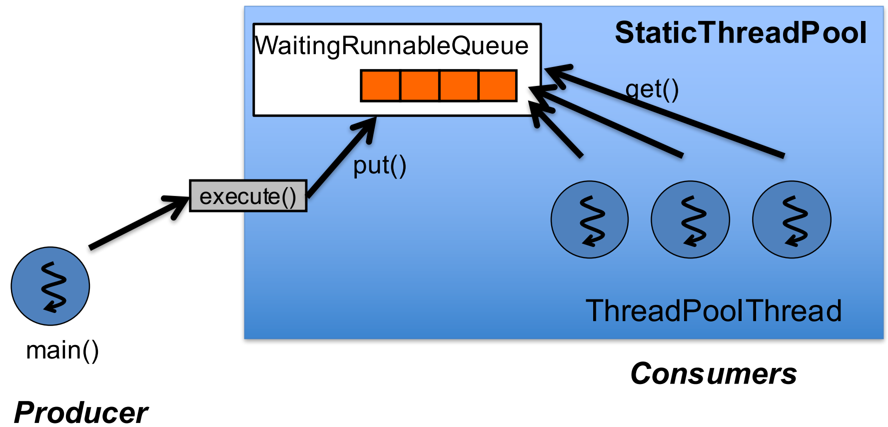
Thread Pool

- A set of pre-created threads that will be used for future data processing
- Benefits
  - Eliminate runtime overhead to create threads
  - Bound the maximum number of threads (i.e., the max amount of resources)
- Examples
  - Web browsers and servers



- When a thread finishes up a given task in a pool, it will get another queued task.
- If no task is available, it goes to the Waiting state until a thread pool client queues a task.

StaticThreadPool.java



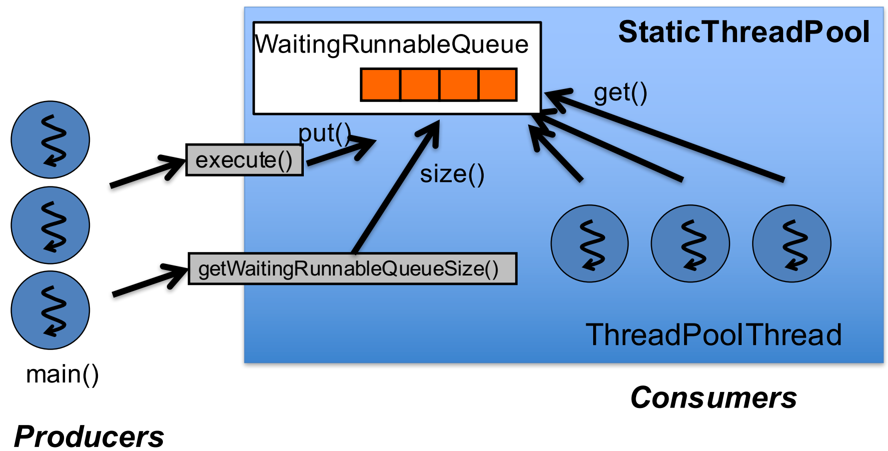
"static" = the number of threads is fixed.

HW 20

- WaitingRunnableQueue uses a Vector to implement a task queue.
  - ```
    private class WaitingRunnableQueue{
      private Vector<Runnable> runnables = new Vector<Runnable>();
      ...
    ```
  - Vector is a synchronized collection.
    - Its public methods (e.g., add() and remove()) are synchronized.
- However, WaitingRunnableQueue's put() and get() use a ReentrantLock to access the task queue.
  - ```
    public void put(Runnable obj){
       queueLock.lock();
       runnables.add(obj);
       runnablesAvailable.signalAll();
       queueLock.unlock();  }
    ```
- This is not a bug; it actually needs to use a lock due to potential race conditions.

- Two locks are used.
  - One in Vector, and the other in WaitingRunnableQueue
  - Unnecessary performance loss
    - Higher # of thread sync = higher overhead
- Modify WaitingRunnableQueue to replace Vector with ArrayList.
  - Directly modify WaitingRunnableQueue, or
  - Define a queue interface, and have WaitingRunnableQueue and your queue class implement the interface (optional; extra points to be considered).
    - Strategy

- Revise StaticThreadPool to be a singleton.
  - Make sure that it is thread safe.
- Revise StaticThreadPool.getWaitingRunnableSize() to be perfectly thread safe.
  - It is "OK" so far because there is only one producer thread now. However, it won't be thread safe when multiple producers run.
  - Acquire a ReentrantLock (queueLock of WaitingRunnableQueue)
    - in getWaitingRunnableSize() and WaitingRunnableQueue.size().



- Implement a “shutdown” feature, which terminate all threads in a thread pool.
  - Define a boolean variable (i.e., flag) “stopped” in ThreadPoolThread
    - Initialized as false
  - ThreadPoolThread.run() checks whether “stopped” is true
    - If yes, stop the thread. (Return run().)
    - Otherwise, keep processing tasks. (Keep running the while loop.)
  - Define shutdown() in StaticThreadPool
    - Changes “stopped” from false to true.
      - Use a lock to guard “stopped”
    - Calls interrupt() on all threads to be terminated.
      - in case they are in the waiting state due to await().