Starvation and Fairness

If a thread is not granted CPU time because other threads grab it all, it is called "starvation". The thread is "starved to death" because other threads are allowed the CPU time instead of it. The solution to starvation is called "fairness" - that all threads are fairly granted a chance to execute.

Causes of Starvation in Java

The following three common causes can lead to starvation of threads in Java:

1. Threads with high priority swallow all CPU time from threads with lower priority.
2. Threads are blocked indefinately waiting to enter a synchronized block, because other threads are constantly allowed access before it.
3. Threads waiting on an object (called wait() on it) remain waiting indefinitely because other threads are constantly awakened instead of it.

A Simple Lock

Let's start out by looking at a synchronized block of Java code:

```
public class Counter{

  private int count = 0;

  public int inc(){
    synchronized(this){
      return ++count;
    }
  }
}
```

Notice the `synchronized(this)` block in the `inc()` method. This block makes sure that only one thread can execute the `return ++count` at a time. The code in the synchronized block could have been more advanced, but the simple `++count` suffices to get the point across.

```
public class Counter{

  private Lock lock = new Lock();
  private int count = 0;

  public int inc(){
    lock.lock();
    int newCount = ++count;
    lock.unlock();
    return newCount;
  }
}
```

Here is a simple `Lock` implementation:

```
public class Lock{

  private boolean isLocked = false;

  public synchronized void lock()
  throws InterruptedException{
    while(isLocked){
      wait();
    }
    isLocked = true;
  }

  public synchronized void unlock(){
    isLocked = false;
    notify();
  }
}
```

Notice the while(isLocked) loop, which is also called a "spin lock"

While isLocked is true, the thread calling lock() is parked waiting in the wait() call.

Spurious Wakeup) :without having received a notify()

Lock Reentrance

If a thread already holds the lock on a monitor object, it has access to all blocks synchronized on the same monitor object. This is called reentrance. The thread can reenter any block of code for which it already holds the lock.

Notice how both outer() and inner() are declared synchronized, which in Java is equivalent to a synchronized(this) block.

```
public class Reentrant{

  public synchronized outer(){
    inner();
  }

  public synchronized inner(){
    //do something
  }
}
```

```
public class Lock{

  boolean isLocked = false;

  public synchronized void lock()
  throws InterruptedException{
    while(isLocked){
      wait();
    }
    isLocked = true;
  }

  ...
}
```

```
public class Reentrant2{

  Lock lock = new Lock();

  public outer(){
    lock.lock();
    inner();
    lock.unlock();
  }

  public synchronized inner(){
    lock.lock();
    //do something
    lock.unlock();
  }
}
```

To make the `Lock` class reentrant we need to make a small change:

```
public class Lock{

  boolean isLocked = false;
  Thread  lockedBy = null;
  int     lockedCount = 0;

  public synchronized void lock()
  throws InterruptedException{
    Thread callingThread = Thread.currentThread();
    while(isLocked && lockedBy != callingThread){
      wait();
    }
    isLocked = true;
    lockedCount++;
    lockedBy = callingThread;
  }


  public synchronized void unlock(){
    if(Thread.curentThread() == this.lockedBy){
      lockedCount--;

      if(lockedCount == 0){
        isLocked = false;
        notify();
      }
    }
  }

  ...
}
```

Additionally, we need to count the number of times the lock has been locked by the same thread. Otherwise, a single call to unlock() will unlock the lock, even if the lock has been locked multiple times

Calling unlock() From a finally-clause

```
lock.lock();
try{
  //do critical section code, which may throw exception
} finally {
  lock.unlock();
}
```

fianlly guarantee that lok will be unlocked ncase of exception also..

When guarding a critical section with a Lock, and the critical section may throw exceptions, it is important to call the unlock() method from inside a finally-clause. Doing so makes sure that the Lock is unlocked so other threads can lock it. Here is an example:

This little construct makes sure that the Lock is unlocked in case an exception is thrown from the code in the critical section. If unlock() was not called from inside a finally-clause, and an exception was thrown from the critical section, the Lock would remain locked forever, causing all threads calling lock() on that Lock instance to halt indefinately.

Lock Fairness

startvation and fairness is internal to when we use synchronized blck as it makes no guarantee that about the sequencing of requesting objects..

Read / Write Locks in Java

Read / Write Lock Java Implementation

Read Access	If no threads are writing, and no threads have requested write access.
Write Access	If no threads are reading or writing.

```
public class ReadWriteLock{

  private int readers       = 0;
  private int writers       = 0;
  private int writeRequests = 0;

  public synchronized void lockRead() throws InterruptedException{
    while(writers > 0 || writeRequests > 0){
      wait();
    }
    readers++;
  }

  public synchronized void unlockRead(){
    readers--;
    notifyAll();
  }

  public synchronized void lockWrite() throws InterruptedException{
    writeRequests++;

    while(readers > 0 || writers > 0){
      wait();
    }
    writeRequests--;
    writers++;
  }

  public synchronized void unlockWrite() throws InterruptedException{
    writers--;
    notifyAll();
  }
```

Java 5 comes with read / write lock implementations in the java.util.concurrent package.

and we did not up-prioritize writes, starvation could occur.

Calling notifyAll() also has another advantage. If multiple threads are waiting for read access and none for write access, and unlockWrite() is called, all threads waiting for read access are granted read access at once - not one by one.

Read / Write Lock Reentrance

1. Avoid writing code that reenters locks
2. Use reentrant locks

If a thread calls `lock()` twice without calling `unlock()` in between, the second call to `lock()` will block. A reentrance lockout has occurred.

```
public class Lock{

  private boolean isLocked = false;

  public synchronized void lock()
  throws InterruptedException{
    while(isLocked){
      wait();
    }
    isLocked = true;
  }

  public synchronized void unlock(){
    isLocked = false;
    notify();
  }
}
```

```
public class Reentrant{

  public synchronized outer(){
    inner();
  }

  public synchronized inner(){
    //do something
  }
}
```

Semaphores

Java 5 comes with semaphore implementations in the java.util.concurrent package so you don't have to implement your own semaphores

A Semaphore is a thread synchronization construct that can be used either to send signals between threads to avoid missed signals, or to guard a critical section like you would with a lock.

```
public class Semaphore {
  private boolean signal = false;

  public synchronized void take() {
    this.signal = true;
    this.notify();
  }

  public synchronized void release() throws InterruptedException{
    while(!this.signal) wait();
    this.signal = false;
  }

}
```

The take() method sends a signal which is stored internally in the Semaphore. The release() method waits for a signal. When received the signal flag is cleared again, and the release() method exited.

Using a semaphore like this you can avoid missed signals. You will call take() instead of notify() and release() instead of wait().

If the call to take() happens before the call to release() the thread calling release() will still know that take() was called, because the signal is stored internally in the signal variable.

This is not the case with wait() and notify().

Using Semaphores for Signaling

```
Semaphore semaphore = new Semaphore();

SendingThread sender = new SendingThread(semaphore);

ReceivingThread receiver = new ReceivingThread(semaphore);

receiver.start();
sender.start();
```

```
public class SendingThread {
  Semaphore semaphore = null;

  public SendingThread(Semaphore semaphore){
    this.semaphore = semaphore;
  }

  public void run(){
    while(true){
      //do something, then signal
      this.semaphore.take();

    }
  }
}
```

```
public class RecevingThread {
  Semaphore semaphore = null;

  public ReceivingThread(Semaphore semaphore){
    this.semaphore = semaphore;
  }

  public void run(){
    while(true){
      this.semaphore.release();
      //receive signal, then do something...
    }
  }
}
```

Counting Semaphore

```
public class CountingSemaphore {
  private int signals = 0;

  public synchronized void take() {
    this.signals++;
    this.notify();
  }

  public synchronized void release() throws InterruptedException{
    while(this.signals == 0) wait();
    this.signals--;
  }

}
```

The Semaphore implementation in the previous section does not count the number of signals sent to it by take() method calls. We can change the Semaphore to do so. This is called a counting semaphore

Bounded Semaphore

The `CoutingSemaphore` has no upper bound on how many signals it can store. We can change the semaphore implementation to have an upper bound, like this:

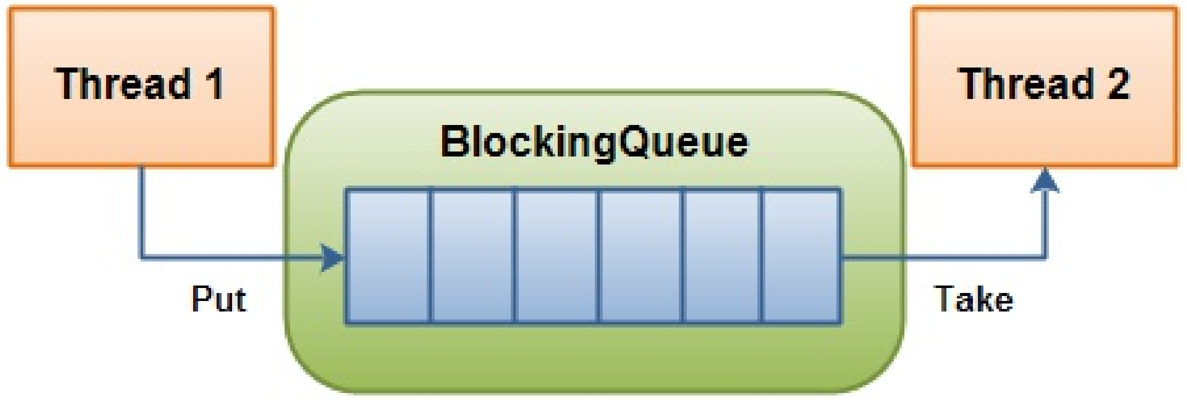
```
public class BoundedSemaphore {
  private int signals = 0;
  private int bound   = 0;

  public BoundedSemaphore(int upperBound){
    this.bound = upperBound;
  }

  public synchronized void take() throws InterruptedException{
    while(this.signals == bound) wait();
    this.signals++;
    this.notify();
  }

  public synchronized void release() throws InterruptedException{
    while(this.signals == 0) wait();
    this.signals--;
    this.notify();
  }
}
```

Using Semaphores as Locks

```
BoundedSemaphore semaphore = new BoundedSemaphore(1);

...

semaphore.take();

try{
  //critical section
} finally {
  semaphore.release();
}
```

You can also use a bounded semaphore to limit the number of threads allowed into a section of code. For instance, in the example above, what would happen if you set the limit of the `BoundedSemaphore` to 5? 5 threads would be allowed to enter the critical section at a time. You would have to make sure though, that the thread operations do not conflict for these 5 threads, or you application will fail.

Blocking Queues



A blocking queue is a queue that blocks when you try to dequeue from it and the queue is empty, or if you try to enqueue items to it and the queue is already full.

A thread trying to dequeue from an empty queue 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.

The implementation of a blocking queue looks similar to a **Bounded Semaphore**. Here is a simple implementation of a blocking queue:

```
public class BlockingQueue {

  private List queue = new LinkedList();
  private int  limit = 10;

  public BlockingQueue(int limit){
    this.limit = limit;
  }


  public synchronized void enqueue(Object item)
  throws InterruptedException  {
    while(this.queue.size() == this.limit) {
      wait();
    }
    if(this.queue.size() == 0) {
      notifyAll();
    }
    this.queue.add(item);
  }
```

```
  public synchronized Object dequeue()
  throws InterruptedException{
    while(this.queue.size() == 0){
      wait();
    }
    if(this.queue.size() == this.limit){
      notifyAll();
    }

    return this.queue.remove(0);
  }

}
```

Notice how notifyAll() is only called from enqueue() and dequeue() if the queue size is equal to the size bounds (0 or limit). If the queue size is not equal to either bound when enqueue() or dequeue() is called, there can be no threads waiting to either enqueue or dequeue items.

Java Synchronized Example

If the two threads had referenced two separate `Counter` instances, there would have been no problems calling the add() methods simultaneously. The calls would have been to different objects, so the methods called would also be synchronized on different objects (the object owning the method). Therefore the calls would not block. Here is how that could look:

```
public class Example {

  public static void main(String[] args){
    Counter counterA = new Counter();
    Counter counterB = new Counter();
    Thread  threadA = new CounterThread(counterA);
    Thread  threadB = new CounterThread(counterB);

    threadA.start();
    threadB.start();
  }
}
```