**Threads--- more info**

**Liveness**

A concurrent application's ability to execute in a timely manner is known as its *liveness*. This section describes the most common kind of liveness problem, [deadlock](http://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html), and goes on to briefly describe two other liveness problems, [starvation and livelock](http://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html).

/\*\*

\* SimpleDeadLock.java

\* Copyright (c) 2008 by Dr. Herong Yang, http://www.herongyang.com/

\*/

import java.util.\*;

public class SimpleDeadLock extends Thread {

public static Object l1 = new Object();

public static Object l2 = new Object();

private int index;

public static void main(String[] a) {

Thread t1 = new Thread1();

Thread t2 = new Thread2();

t1.start();

t2.start();

}

private static class Thread1 extends Thread {

public void run() {

synchronized (l1) {

System.out.println("Thread 1: Holding lock 1...");

try { Thread.sleep(10); }

catch (InterruptedException e) {}

System.out.println("Thread 1: Waiting for lock 2...");

synchronized (l2) {

System.out.println("Thread 2: Holding lock 1 & 2...");

}

}

}

}

private static class Thread2 extends Thread {

public void run() {

synchronized (l2) {

System.out.println("Thread 2: Holding lock 2...");

try { Thread.sleep(10); }

catch (InterruptedException e) {}

System.out.println("Thread 2: Waiting for lock 1...");

synchronized (l1) {

System.out.println("Thread 2: Holding lock 2 & 1...");

}

}

}

}

}

# Starvation and Livelock

Starvation and livelock are much less common a problem than deadlock, but are still problems that every designer of concurrent software is likely to encounter.

## Starvation

*Starvation* describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

## Livelock

A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work. This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left. They're still blocking each other, so...

**Guarded Blocks**

Threads often have to coordinate their actions. The most common coordination idiom is the *guarded block*. Such a block begins by polling a condition that must be true before the block can proceed. There are a number of steps to follow in order to do this correctly.

Suppose, for example guardedJoy is a method that must not proceed until a shared variable joy has been set by another thread. Such a method could, in theory, simply loop until the condition is satisfied, but that loop is wasteful, since it executes continuously while waiting.

public void guardedJoy() {

// Simple loop guard. Wastes

// processor time. Don't do this!

while(!joy) {}

System.out.println("Joy has been achieved!");

}

A more efficient guard invokes [Object.wait](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#wait()) to suspend the current thread. The invocation of wait does not return until another thread has issued a notification that some special event may have occurred — though not necessarily the event this thread is waiting for:

public synchronized guardedJoy() {

// This guard only loops once for each special event, which may not

// be the event we're waiting for.

while(!joy) {

try {

wait();

} catch (InterruptedException e) {}

}

System.out.println("Joy and efficiency have been achieved!");

}

**Note:** Always invoke wait inside a loop that tests for the condition being waited for. Don't assume that the interrupt was for the particular condition you were waiting for, or that the condition is still true.

Like many methods that suspend execution, wait can throw InterruptedException. In this example, we can just ignore that exception — we only care about the value of joy.

Why is this version of guardedJoy synchronized? Suppose d is the object we're using to invoke wait. When a thread invokes d.wait, it must own the intrinsic lock for d — otherwise an error is thrown. Invoking wait inside a synchronized method is a simple way to acquire the intrinsic lock.

When wait is invoked, the thread releases the lock and suspends execution. At some future time, another thread will acquire the same lock and invoke [Object.notifyAll](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#notifyAll()), informing all threads waiting on that lock that something important has happened:

public synchronized notifyJoy() {

joy = true;

notifyAll();

}

Some time after the second thread has released the lock, the first thread reacquires the lock and resumes by returning from the invocation of wait.

**Note:** There is a second notification method, notify, which wakes up a single thread. Because notify doesn't allow you to specify the thread that is woken up, it is useful only in massively parallel applications — that is, programs with a large number of threads, all doing similar chores. In such an application, you don't care which thread gets woken up.

Let's use guarded blocks to create a *Producer-Consumer* application. This kind of application shares data between two threads: the *producer*, that creates the data, and the *consumer*, that does something with it. The two threads communicate using a shared object. Coordination is essential: the consumer thread must not attempt to retrieve the data before the producer thread has delivered it, and the producer thread must not attempt to deliver new data if the consumer hasn't retrieved the old data.

In this example, the data is a series of text messages, which are shared through an object of type [Drop](http://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Drop.java):

public class Drop {

// Message sent from producer

// to consumer.

private String message;

// True if consumer should wait

// for producer to send message,

// false if producer should wait for

// consumer to retrieve message.

private boolean empty = true;

public synchronized String take() {

// Wait until message is

// available.

while (empty) {

try {

wait();

} catch (InterruptedException e) {}

}

// Toggle status.

empty = true;

// Notify producer that

// status has changed.

notifyAll();

return message;

}

public synchronized void put(String message) {

// Wait until message has

// been retrieved.

while (!empty) {

try {

wait();

} catch (InterruptedException e) {}

}

// Toggle status.

empty = false;

// Store message.

this.message = message;

// Notify consumer that status

// has changed.

notifyAll();

}

}

The producer thread, defined in [Producer](http://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Producer.java), sends a series of familiar messages. The string "DONE" indicates that all messages have been sent. To simulate the unpredictable nature of real-world applications, the producer thread pauses for random intervals between messages.

import java.util.Random;

public class Producer implements Runnable {

private Drop drop;

public Producer(Drop drop) {

this.drop = drop;

}

public void run() {

String importantInfo[] = {

"Mares eat oats",

"Does eat oats",

"Little lambs eat ivy",

"A kid will eat ivy too"

};

Random random = new Random();

for (int i = 0;

i < importantInfo.length;

i++) {

drop.put(importantInfo[i]);

try {

Thread.sleep(random.nextInt(5000));

} catch (InterruptedException e) {}

}

drop.put("DONE");

}

}

The consumer thread, defined in [Consumer](http://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Consumer.java), simply retrieves the messages and prints them out, until it retrieves the "DONE" string. This thread also pauses for random intervals.

import java.util.Random;

public class Consumer implements Runnable {

private Drop drop;

public Consumer(Drop drop) {

this.drop = drop;

}

public void run() {

Random random = new Random();

for (String message = drop.take();

! message.equals("DONE");

message = drop.take()) {

System.out.format("MESSAGE RECEIVED: %s%n", message);

try {

Thread.sleep(random.nextInt(5000));

} catch (InterruptedException e) {}

}

}

}

Finally, here is the main thread, defined in [ProducerConsumerExample](http://docs.oracle.com/javase/tutorial/essential/concurrency/examples/ProducerConsumerExample.java), that launches the producer and consumer threads.

public class ProducerConsumerExample {

public static void main(String[] args) {

Drop drop = new Drop();

(new Thread(new Producer(drop))).start();

(new Thread(new Consumer(drop))).start();

}

}

# Lock Objects

Synchronized code relies on a simple kind of reentrant lock. This kind of lock is easy to use, but has many limitations. More sophisticated locking idioms are supported by the [java.util.concurrent.locks](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/package-summary.html) package. We won't examine this package in detail, but instead will focus on its most basic interface, [Lock](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html).

Lock objects work very much like the implicit locks used by synchronized code. As with implicit locks, only one thread can own a Lock object at a time. Lock objects also support a wait/notify mechanism, through their associated [Condition](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Condition.html) objects.

The biggest advantage of Lock objects over implicit locks is their ability to back out of an attempt to acquire a lock. The tryLock method backs out if the lock is not available immediately or before a timeout expires (if specified). The lockInterruptibly method backs out if another thread sends an interrupt before the lock is acquired.

Let's use Lock objects to solve the deadlock problem we saw in [Liveness](http://docs.oracle.com/javase/tutorial/essential/concurrency/liveness.html). Alphonse and Gaston have trained themselves to notice when a friend is about to bow. We model this improvement by requiring that our Friend objects must acquire locks for *both* participants before proceeding with the bow. Here is the source code for the improved model, [Safelock](http://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Safelock.java). To demonstrate the versatility of this idiom, we assume that Alphonse and Gaston are so infatuated with their newfound ability to bow safely that they can't stop bowing to each other:

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

import java.util.Random;

public class Safelock {

static class Friend {

private final String name;

private final Lock lock = new ReentrantLock();

public Friend(String name) {

this.name = name;

}

public String getName() {

return this.name;

}

public boolean impendingBow(Friend bower) {

Boolean myLock = false;

Boolean yourLock = false;

try {

myLock = lock.tryLock();

yourLock = bower.lock.tryLock();

} finally {

if (! (myLock && yourLock)) {

if (myLock) {

lock.unlock();

}

if (yourLock) {

bower.lock.unlock();

}

}

}

return myLock && yourLock;

}

public void bow(Friend bower) {

if (impendingBow(bower)) {

try {

System.out.format("%s: %s has"

+ " bowed to me!%n",

this.name, bower.getName());

bower.bowBack(this);

} finally {

lock.unlock();

bower.lock.unlock();

}

} else {

System.out.format("%s: %s started"

+ " to bow to me, but saw that"

+ " I was already bowing to"

+ " him.%n",

this.name, bower.getName());

}

}

public void bowBack(Friend bower) {

System.out.format("%s: %s has" +

" bowed back to me!%n",

this.name, bower.getName());

}

}

static class BowLoop implements Runnable {

private Friend bower;

private Friend bowee;

public BowLoop(Friend bower, Friend bowee) {

this.bower = bower;

this.bowee = bowee;

}

public void run() {

Random random = new Random();

for (;;) {

try {

Thread.sleep(random.nextInt(10));

} catch (InterruptedException e) {}

bowee.bow(bower);

}

}

}

public static void main(String[] args) {

final Friend alphonse =

new Friend("Alphonse");

final Friend gaston =

new Friend("Gaston");

new Thread(new BowLoop(alphonse, gaston)).start();

new Thread(new BowLoop(gaston, alphonse)).start();

}

}

**Executors**

In all of the previous examples, there's a close connection between the task being done by a new thread, as defined by its Runnable object, and the thread itself, as defined by a Thread object. This works well for small applications, but in large-scale applications, it makes sense to separate thread management and creation from the rest of the application. Objects that encapsulate these functions are known as *executors*. The following subsections describe executors in detail.

* [Executor Interfaces](http://docs.oracle.com/javase/tutorial/essential/concurrency/exinter.html) define the three executor object types.
* [Thread Pools](http://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html) are the most common kind of executor implementation.
* [Fork/Join](http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html) is a framework (new in JDK 7) for taking advantage of multiple processors

# Executor Interfaces

The java.util.concurrent package defines three executor interfaces:

* Executor, a simple interface that supports launching new tasks.
* ExecutorService, a subinterface of Executor, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself.
* ScheduledExecutorService, a subinterface of ExecutorService, supports future and/or periodic execution of tasks.

Typically, variables that refer to executor objects are declared as one of these three interface types, not with an executor class type.

## The Executor Interface

The [Executor](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html) interface provides a single method, execute, designed to be a drop-in replacement for a common thread-creation idiom. If r is a Runnable object, and e is an Executor object you can replace

(new Thread(r)).start();

with

e.execute(r);

However, the definition of execute is less specific. The low-level idiom creates a new thread and launches it immediately. Depending on the Executor implementation, execute may do the same thing, but is more likely to use an existing worker thread to run r, or to place r in a queue to wait for a worker thread to become available. (We'll describe worker threads in the section on [Thread Pools](http://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html).)

The executor implementations in java.util.concurrent are designed to make full use of the more advanced ExecutorService and ScheduledExecutorService interfaces, although they also work with the base Executor interface.

## The ExecutorService Interface

The [ExecutorService](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html) interface supplements execute with a similar, but more versatile submit method. Like execute, submit accepts Runnable objects, but also accepts [Callable](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Callable.html) objects, which allow the task to return a value. The submit method returns a [Future](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html) object, which is used to retrieve the Callable return value and to manage the status of both Callable and Runnable tasks.

ExecutorService also provides methods for submitting large collections of Callable objects. Finally, ExecutorService provides a number of methods for managing the shutdown of the executor. To support immediate shutdown, tasks should handle [interrupts](http://docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html) correctly.

## The ScheduledExecutorService Interface

The [ScheduledExecutorService](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledExecutorService.html) interface supplements the methods of its parent ExecutorService with schedule, which executes a Runnable or Callable task after a specified delay. In addition, the interface defines scheduleAtFixedRate and scheduleWithFixedDelay, which executes specified tasks repeatedly, at defined intervals.

**Thread Pools**

Most of the executor implementations in java.util.concurrent use *thread pools*, which consist of *worker threads*. This kind of thread exists separately from the Runnable and Callable tasks it executes and is often used to execute multiple tasks.

Using worker threads minimizes the overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead.

One common type of thread pool is the *fixed thread pool*. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread. Tasks are submitted to the pool via an internal queue, which holds extra tasks whenever there are more active tasks than threads.

An important advantage of the fixed thread pool is that applications using it *degrade gracefully*. To understand this, consider a web server application where each HTTP request is handled by a separate thread. If the application simply creates a new thread for every new HTTP request, and the system receives more requests than it can handle immediately, the application will suddenly stop responding to *all* requests when the overhead of all those threads exceed the capacity of the system. With a limit on the number of the threads that can be created, the application will not be servicing HTTP requests as quickly as they come in, but it will be servicing them as quickly as the system can sustain.

A simple way to create an executor that uses a fixed thread pool is to invoke the [newFixedThreadPool](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html#newFixedThreadPool(int)) factory method in [java.util.concurrent.Executors](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html) This class also provides the following factory methods:

* The [newCachedThreadPool](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html#newCachedThreadPool(int)) method creates an executor with an expandable thread pool. This executor is suitable for applications that launch many short-lived tasks.
* The [newSingleThreadExecutor](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html#newSingleThreadExecutor(int)) method creates an executor that executes a single task at a time.
* Several factory methods are ScheduledExecutorService versions of the above executors.

If none of the executors provided by the above factory methods meet your needs, constructing instances of [java.util.concurrent.ThreadPoolExecutor](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html) or [java.util.concurrent.ScheduledThreadPoolExecutor](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledThreadPoolExecutor.html) will give you additional options.

# Fork/Join

New in the Java SE 7 release, the fork/join framework is an implementation of the ExecutorService interface that helps you take advantage of multiple processors. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application.

As with any ExecutorService, the fork/join framework distributes tasks to worker threads in a thread pool. The fork/join framework is distinct because it uses a *work-stealing* algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy.

The center of the fork/join framework is the [ForkJoinPool](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinPool.html) class, an extension of AbstractExecutorService. ForkJoinPool implements the core work-stealing algorithm and can execute [ForkJoinTask](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinTask.html)s.

## Basic Use

Using the fork/join framework is simple. The first step is to write some code that performs a segment of the work. Your code should look similar to this:

if (my portion of the work is small enough)

do the work directly

else

split my work into two pieces

invoke the two pieces and wait for the results

Wrap this code as a ForkJoinTask subclass, typically as one of its more specialized types [RecursiveTask](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/RecursiveTask.html)(which can return a result) or [RecursiveAction](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/RecursiveAction.html).

After your ForkJoinTask is ready, create one that represents all the work to be done and pass it to the invoke() method of a ForkJoinPool instance.

## Blurring for Clarity

To help you understand how the fork/join framework works, consider a simple example. Suppose you want to perform a simple blur on an image. The original *source* image is represented by an array of integers, where each integer contains the color values for a single pixel. The blurred *destination* image is also represented by an integer array with the same size as the source.

Performing the blur is accomplished by working through the source array one pixel at a time. Each pixel is averaged with its surrounding pixels (the red, green, and blue components are averaged), and the result is placed in the destination array. Here is one possible implementation:

public class ForkBlur extends RecursiveAction {

private int[] mSource;

private int mStart;

private int mLength;

private int[] mDestination;

// Processing window size, should be odd.

private int mBlurWidth = 15;

public ForkBlur(int[] src, int start, int length, int[] dst) {

mSource = src;

mStart = start;

mLength = length;

mDestination = dst;

}

protected void computeDirectly() {

int sidePixels = (mBlurWidth - 1) / 2;

for (int index = mStart; index < mStart + mLength; index++) {

// Calculate average.

float rt = 0, gt = 0, bt = 0;

for (int mi = -sidePixels; mi <= sidePixels; mi++) {

int mindex = Math.min(Math.max(mi + index, 0),

mSource.length - 1);

int pixel = mSource[mindex];

rt += (float)((pixel & 0x00ff0000) >> 16)

/ mBlurWidth;

gt += (float)((pixel & 0x0000ff00) >> 8)

/ mBlurWidth;

bt += (float)((pixel & 0x000000ff) >> 0)

/ mBlurWidth;

}

// Re-assemble destination pixel.

int dpixel = (0xff000000 ) |

(((int)rt) << 16) |

(((int)gt) << 8) |

(((int)bt) << 0);

mDestination[index] = dpixel;

}

}

...

Now you implement the abstract compute() method, which either performs the blur directly or splits it into two smaller tasks. A simple array length threshold helps determine whether the work is performed or split.

protected static int sThreshold = 100000;

protected void compute() {

if (mLength < sThreshold) {

computeDirectly();

return;

}

int split = mLength / 2;

invokeAll(new ForkBlur(mSource, mStart, split, mDestination),

new ForkBlur(mSource, mStart + split, mLength - split,

mDestination));

}

If the previous methods are in a subclass of the RecursiveAction class, setting it up to run in a ForkJoinPool is straightforward.

Create a task that represents all of the work to be done.

// source image pixels are in src

// destination image pixels are in dst

ForkBlur fb = new ForkBlur(src, 0, src.length, dst);

Create the ForkJoinPool that will run the task.

ForkJoinPool pool = new ForkJoinPool();

Run the task.

pool.invoke(fb);

For the full source code, including some extra code that shows the source and destination images in windows, see the [ForkBlur](http://docs.oracle.com/javase/tutorial/essential/concurrency/examples/ForkBlur.java) class.

# Atomic Variables

The [java.util.concurrent.atomic](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/package-summary.html) package defines classes that support atomic operations on single variables. All classes have get and set methods that work like reads and writes on volatile variables. That is, a set has a happens-before relationship with any subsequent get on the same variable. The atomic compareAndSet method also has these memory consistency features, as do the simple atomic arithmetic methods that apply to integer atomic variables.

To see how this package might be used, let's return to the [Counter](http://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Counter.java) class we originally used to demonstrate thread interference:

class Counter {

private int c = 0;

public void increment() {

c++;

}

public void decrement() {

c--;

}

public int value() {

return c;

}

}

One way to make Counter safe from thread interference is to make its methods synchronized, as in [SynchronizedCounter](http://docs.oracle.com/javase/tutorial/essential/concurrency/examples/SynchronizedCounter.java):

class SynchronizedCounter {

private int c = 0;

public synchronized void increment() {

c++;

}

public synchronized void decrement() {

c--;

}

public synchronized int value() {

return c;

}

}

For this simple class, synchronization is an acceptable solution. But for a more complicated class, we might want to avoid the liveness impact of unnecessary synchronization. Replacing the int field with an AtomicInteger allows us to prevent thread interference without resorting to synchronization, as in [AtomicCounter](http://docs.oracle.com/javase/tutorial/essential/concurrency/examples/AtomicCounter.java):

import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {

private AtomicInteger c = new AtomicInteger(0);

public void increment() {

c.incrementAndGet();

}

public void decrement() {

c.decrementAndGet();

}

public int value() {

return c.get();

}

}