**Thread Pool**

8.1. Implicit Couplings Between Tasks and Execution Policies (GTK)

We thought that : Executor framework which decouples task submission from task execution

Can be used in any condition and solves all thread related issues then, its not completely true (over-statement)

Issues with Executors

1. Dependend task can not run in a thead pool
2. Single-threaded executors can not be made to run in a concurrent way.
3. Thread local can not be used in Thread pool – why ?
4. Thread pools can have responsiveness problems if tasks can block for extended periods of time, even if deadlock is not a possibility,

So Thread pools work best when tasks are homogeneous and independent.

Thread Starvation Deadlock

If tasks that depend on other tasks execute in a thread pool, they can deadlock.

In a single-threaded executor, a task that submits another task to the same executor and waits for its result will always deadlock. The second task sits on the work queue until the first task completes, but the first will not complete because it is waiting for the result of the second task.

Render-PageTask submits two additional tasks to the Executor to fetch the page header and footer, renders the page body, waits for the results of the header and footer tasks, and then combines the header, body, and footer into the finished page. With a single-threaded executor, THReadDeadlock will always deadlock

public class ThreadDeadlock {

ExecutorService exec = Executors.newSingleThreadExecutor();

public class RenderPageTask implements Callable<String> {

public String call() throws Exception {

Future<String> header, footer;

header = exec.submit(new LoadFileTask("header.html"));

footer = exec.submit(new LoadFileTask("footer.html"));

String page = renderBody();

// Will deadlock -- task waiting for result of subtask

return header.get() + page + footer.get();

}

}}

Eg Project : D:\workspace\Thread\Thread

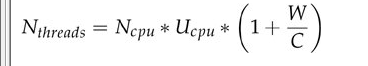
Class : TwoDependentTask

Sizing Thread Pools

Thread pool should not be “too big" and "too small". If a thread pool is too big, then threads compete for scarce CPU and memory resources, resulting in higher memory usage and possible resource exhaustion. If it is too small, throughput suffers as processors go unused despite available work.

For compute-intensive tasks, an Ncpu-processor system usually achieves optimum utilization with a thread pool of Ncpu +1 threads

For tasks that also include I/O or other blocking operations, you want a larger pool, since not all of the threads will be schedulable at all times.



U = utilization 0<u<1

W = waiting time (IO\jdbc \waiting and other waiting period)

C= Computation time

int N\_CPUS = Runtime.getRuntime().availableProcessors();

Configuring ThreadPoolExecutor

1. ThreadPoolExecutor provides the base implementation for the executors returned by the newCachedThreadPool, newFixedThreadPool, and newScheduled-ThreadExecutor factories in Executors. ThreadPoolExecutor.
2. Basic Constructor for thread pool

public ThreadPoolExecutor(intcorePoolSize,

intmaximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable>workQueue,

ThreadFactorythreadFactory,

RejectedExecutionHandler handler)

1. The core pool size, maximum pool size, and keep-alive time govern thread creation and teardown
2. The core size is the target size; the implementation attempts to maintain the pool at this size even when there are no tasks to execute,[[2]](mk:@MSITStore:E:\Books\Java\Java%20Concurrency%20In%20Practice%20%5bDark%20Demon%5d%20%5bh33t%5d.chm::/0321349601/ch08lev1sec3.html#ch08fn02) and will not create more threads than this unless the work queue is full.
3. WhenaThreadPoolExecutor is initially created, the core threads are not started immediately but instead as tasks are submitted

Q. What if we put the core pool size 0 , means no threads are there in the waiting, and will be created on demenad

Ans : .THReadPoolExecutor creates a new thread only if the work queue is full. So tasks submitted to a thread pool with a work queue that has any capacity and a core size of zero will not execute until the queue fills up, which is usually not what is desired. In Java 6

1. The maximum pool size is the upper bound on how many pool threads can be active at once
2. A thread that has been idle for longer than the keep-alive time becomes a candidate for reaping and can be terminated if the current pool size exceeds the core size
3. The newFixedThreadPool factory sets both the core pool size and the maximum pool size to the requested pool size, creating the effect of infinite timeout;
4. the newCachedThreadPool factory sets the maximum pool size to Integer.MAX\_VALUE and the core pool size to zero with a timeout of one minute, creating the effect of an infinitely expandable thread pool that will contract again when demand decreases.

It uses SynchronousQueue for its work queue to maintain zero pool size.

Queuing in Thread Pool:

We us thread pool to get away with the problem of unbouned thread creation.ButIf the arrival rate for new requests exceeds the rate at which they can be handled, requests will still queue up.With a thread pool, they wait in a queue of Runnables managed by the Executor instead of queueing up as threads contending for the CPU.

ThreadPoolExecutor allows you to supply a BlockingQueue to hold tasks awaiting execution. There are three basic approaches to task queueing: unbounded queue, bounded queue, and synchronous handoff. The choice of queue interacts with other configuration parameters such as pool size

1. The default for newFixedThreadPool and newSingleThreadExecutor is to use an unbounded LinkedBlockingQueue.
2. Bounded queue Vs unbounded queue: A more stable resource management strategy is to use a bounded queue, such as an ArrayBlockingQueue or a bounded LinkedBlockingQueue or Priority-BlockingQueue. Bounded queues help prevent resource exhaustion but introduce the question of what to do with new tasks when the queue is full. There are saturation polices to deal that.
3. newCachedThreadPool uses **SynchronousQueue.**
4. A SynchronousQueue (added in Java 1.6) is not really a queue at all, but a mechanism for managing handoffs between threads. In order to put an element on a SynchronousQueue, another thread must already be waiting to accept the handoff. If no thread is waiting but the current pool size is less than the maximum, Thread-PoolExecutor creates a new thread; otherwise the task is rejected according to the saturation policy.
5. Using a direct handoff is more efficient because the task can be handed right to the thread that will execute it, rather than first placing it on a queue and then having the worker thread fetch it from the queue. SynchronousQueue is a practical choice only if the pool is unbounded or if rejecting excess tasks is acceptable
6. The newCachedThreadPool factory is a good default choice for an Executor, providing better queuing performance than a fixed thread pool.[[5]](mk:@MSITStore:E:\Books\Java\Java%20Concurrency%20In%20Practice%20%5bDark%20Demon%5d%20%5bh33t%5d.chm::/0321349601/ch08lev1sec3.html#ch08fn05) A fixed size thread pool is a good choice when you need to limit the number of concurrent tasks for resource-management purposes, as in a server application that accepts requests from network clients and would otherwise be vulnerable to overload.

Saturation Policies :

1. When a bounded work queue fills up, the saturation policy comes into play.

executor.setRejectedExecutionHandler(

new ThreadPoolExecutor.CallerRunsPolicy());

1. A num all available policy are : AbortPolicy, CallerRunsPolicy, DiscardPolicy, and DiscardOldestPolicy
2. The default policy abort, causes execute to throw the unchecked Rejected-ExecutionException; the caller can catch this exception and implement its own overflow handling.
3. The discard policy silently discards the newly submitted task if it cannot be queued for execution; the discard-oldest policy discards the task that would otherwise be executed next and tries to resubmit the new task.
4. caller-runs policy implements a form of throttling that neither discards tasks nor throws an exception, but instead tries to slow down the flow of new tasks by pushing some of the work back to the caller,It executes the newly submitted task not in a pool thread, but in the thread that calls execute
5. Since this would probably take some time, the main thread\calling thread cannot submit any more tasks for at least a little while, giving the worker threads some time to catch up on the backlog. The main thread would also not be calling accept during this time, so incoming requests will queue up in the TCP layer instead of in the application

It’s a good one , read more from the book, if u like

There is no predefined saturation policy to make execute block when the work queue is full.

Think how this can done and implement it.

How thread pool works ?

How thread are created ? How threads are resused.

Concept of worker thread :

A worker class, contain a thread T, which when started made to poll the workqueue.

Quite like this :

Thread t = worker.thread.CreatNewThred(runnableTask);

task = queue.take();

publicvoid run() {

while (task != null) {

try {

task().run();

} catch (InterruptedException ex) {

return;

}

}

}

});

t.start();

From jdk 1.6

**publicvoid** run() {

**try** {

hasRun = **true**;

Runnable task = firstTask;

firstTask = **null**;

**while** (task != **null** || (task = getTask()) != **null**) {

runTask(task);

task = **null**;

}

} **finally** {

workerDone(**this**);

}

}

}

In its run method a worker is getting a task from the workqueue inside a while loop.

Extending ThreadPoolExecutor

ThreadPoolExecutor was designed for extension, providing several "hooks" for subclasses to overridebeforeExecute, afterExecute, and terminatethat can be used to extend the behavior of ThreadPoolExecutor.

The beforeExecute and afterExecute hooks are called in the thread that executes the task, and can be used for adding logging, timing, monitoring, or statistics gathering. The afterExecute hook is called whether the task completes by returning normally from run or by throwing an Exception.

Parallelizing Recursive Algorithms

Try to run in parallel list of files in a dir

**Chapter 9. GUI Applications**

Why are GUIs Single-threaded

1. there have been many attempts to write multithreaded GUI frameworks, but because of persistent problems with race conditions and deadlock, they all eventually arrived at the single-threaded event queue model in which a dedicated thread fetches events off a queue and dispatches them to applicationdefined event handlers.
2. AWT originally tried to support a greater degree of multithreaded access, and the decision to make Swing single-threaded was based largely on experience with AWT.
3. Multithreaded GUI frameworks tend to be particularly susceptible to deadlock, partially because of the unfortunate interaction between input event processing and any sensible object-oriented modeling of GUI components. Inconsistent ordering of locks.
4. Another factor is mvc model - inconsistent lock ordering controller can call the model and view, model can also notify view
5. The Swing single-thread rule: Swing components and models should be created, modified, and queried only from the event-dispatching thread.

**Thread Confinement in Swing**

1. All Swing components (such as JButton and JTable) and data model objects (such as TableModel and TReeModel) are confined to the event thread, so any code that accesses these objects must run in the event thread
2. GUI objects are kept consistent not by synchronization, but by thread confinement
3. The upside is that tasks that run in the event thread need not worry about synchronization when accessing presentation objects
4. the downside is that you cannot access presentation objects from outside the event thread at all.

The Swing single-thread rule: Swing components and models should be created, modified, and queried only from the event-dispatching thread.

Wondering what can be the issue

<https://bitguru.wordpress.com/2007/03/21/will-the-real-swing-single-threading-rule-please-stand-up/>

 They create JFrames and other JComponents in the main thread, either directly in main() or indirectly. Swing is not thread-safe (by design) so we have always had to be careful.

If it executes in the main thread or some other thread it’s likely to work just fine, but there are no guarantees and it is possible that bad things (such as deadlock) may happen. Even if it seems to work on your development machine, it may fail intermittently, or it may even deterministically fail on some platform on which you haven’t tested or on some future release of the JDK.

More on this :

<http://stackoverflow.com/questions/2484425/how-does-the-event-dispatch-thread-work>

How the swing component must b created :

 publicstaticvoidmain(String[] args) {

        //Schedule a job for the event-dispatching thread:

        //creating and showing this application's GUI.

        javax.swing.SwingUtilities.invokeLater(newRunnable() {

            publicvoidrun() {

                createAndShowGUI();

            }

        });

    }

As with all rules, there are a few exceptions. A small number of Swing methods may be called safely from any thread; these are clearly identified in the Javadoc as being thread-safe. Other exceptions to the single-thread rule include:

* SwingUtilities.isEventDispatchThread, which determines whether the current thread is the event thread;
* SwingUtilities.invokeLater, which schedules a Runnable for execution on the event thread (callable from any thread);
* SwingUtilities.invokeAndWait, which schedules a Runnable task for execution on the event thread and blocks the current thread until it completes (callable only from a non-GUI thread);

Short-running GUI Tasks

So long as tasks are short-lived and access only GUI objects (or other thread-confined or thread-safe application objects), you can almost totally ignore threading concerns and do everything from the event thread, and the right thing happens.

final Random random = new Random();

finalJButton button = new JButton("Change Color");

...

button.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

button.setBackground(new Color(random.nextInt()));

}

});

Long running task

Which diplaysomthing to user

button.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

button.setEnabled(false);

label.setText("busy");

backgroundExec.execute(new Runnable() {

public void run() {

try {

doBigComputation();

} finally {

GuiExecutor.instance().execute(new Runnable() {

public void run() {

button.setEnabled(true);

label.setText("idle");

}

});

}

}

});

}

});

**Cancellaton**

How to cancel a running task using executor

Future<?>runningTask = null;

When you call cancel on a Future with mayInterruptIfRunning set to true, the Future implementation interrupts the thread that is executing the task if it is currently running.

If your task is written to be responsive to interruption, it can return early if it is cancelled.

cancelButton.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent event) {

if (runningTask != null)

runningTask.cancel(true);

}});

runningTask = backgroundExec.submit(new Runnable() {

public void run() {

while (moreWork()) {

if (Thread.currentThread().isInterrupted()) {

cleanUpPartialWork();

break;

}

doSomeWork();

}

}

});

Task :Swing cancel task with status

**Dead Lock in Java**

How Db detect and **handle** the deadlock

When DB detects that a set of transactions is deadlocked (which it does by searching the is-waiting-for graph for cycles), it picks a victim and aborts that transaction.

When a set of Java threads deadlock, that's the end of the game.

1. A program will be free of lock-ordering deadlocks if all threads acquire the locks they need in a fixed global order
2. DeadLockeg:

public class LeftRightDeadlock {

private final Object left = new Object();

private final Object right = new Object();

public void leftRight() {

synchronized (left) {

synchronized (right) {

doSomething();

}

}

}

public void rightLeft() {

synchronized (right) {

synchronized (left) {

doSomethingElse();

} }}

1. Transfer Money deadLock

public void transferMoney(Account fromAccount,

Account toAccount,

DollarAmount amount)

throwsInsufficientFundsException {

synchronized (fromAccount) {

synchronized (toAccount) {

if (fromAccount.getBalance().compareTo(amount) < 0)

throw new InsufficientFundsException();

else {

fromAccount.debit(amount);

toAccount.credit(amount);

}}}}

A: transferMoney(myAccount, yourAccount, 10);

B: transferMoney(yourAccount, myAccount, 20);

1. One way to induce an ordering on objects is to use System.identityHashCode, which returns the value that would be returned by Object.hashCode

Soluton :

intfromHash = System.identityHashCode(fromAcct);

inttoHash = System.identityHashCode(toAcct);

if (fromHash<toHash) {

synchronized (fromAcct) {

synchronized (toAcct) {

new Helper().transfer();

}

}

} else if (fromHash>toHash) {

synchronized (toAcct) {

synchronized (fromAcct) {

new Helper().transfer();

}

}

} else {

synchronized (tieLock) {

synchronized (fromAcct) {

synchronized (toAcct) {

new Helper().transfer();

}

}

}

}

You may think we're overstating the risk of deadlock because locks are usually held only briefly, but deadlocks are a serious problem in real systems. A production application may perform billions of lock acquire-release cycles per day

**Deadlocks Between Cooperating Objects**

the warning sign is that an alien method is being called while holding a lock.

Since both setLocation and notifyAvailable are synchronized, the thread calling setLocation acquires the Taxi lock and then the Dispatcher lock. Similarly, a thread calling getImage acquires the Dispatcher lock and then each Taxi lock

Class taxi

{

public synchronized void setLocation(Point location) {

this.location = location;

if (location.equals(destination))

dispatcher.notifyAvailable(this);

}

public synchronized Point getLocation() {

return location;

}

}

Image

class Dispatcher {

public synchronized Image getImage() {

Image image = new Image();

for (Taxi t : taxis)

image.drawMarker(t.getLocation());

return image;

}

}

Open Call

1. calling an alien method with a lock held is difficult to analyze and therefore risky
2. Calling a method with no locks held is called an open call,
3. classes that rely on open calls are more well-behaved and composable than classes that make calls with locks held.
4. Open call helps in **avoiding dead lock** especially deadlock in cooperating objects

synchronized (this) {

this.location = location;

reachedDestination = location.equals(destination);

}

if (reachedDestination)

dispatcher.notifyAvailable(this);

}

1. Using open calls to avoid deadlock is analogous to using encapsulation to provide thread safety: while one can certainly construct a thread-safe program without any encapsulation, the thread safety analysis of a program that makes effective use of encapsulation is far easier than that of one that does not

**Avoiding and Diagnosing Deadlock**

1. A program that never acquires more than one lock at a time cannot experience lock-ordering deadlock
2. identify where multiple locks could be acquired and use ordered locking in thoese cases
3. Use open call when calling alien method
4. Use timed tryLock feature of the explicit Lock classes instead of intrinsic locking.

How try lock with time out can help in detecting deadlock :

1. If u r not able to get lock after time out that means either it’s a deadlock or some thing fishy , like infinite loop,but we know soe thing not write
2. So we can record the activity
3. We can give up all the locks held and try again after certain period
4. and try again, possibly clearing the deadlock condition and allowing the program to recover

**Detect : Thread dumps**

1. How to get tread dumps : Jstack
2. JVM can help identify them when they do happen using thread dumps
3. Thread dumps also include locking information, such as which locks are held by each thread, in which stack frame

In linux : kill -3 pid

How to analyzed them

There are tools like thread dump analyzer, which can filter the thread,

Like which are waitings

<http://stackoverflow.com/questions/3156434/thread-dump-analysis-tool-method>

<http://mchr3k.github.io/javathreaddumpanalyser/>

GTK

**Starvation**

Starvation occurs when a thread is perpetually denied access to resources it needs in order to make progress;

Often caused by inappropriate use of thread priorities

It can also be caused by executing nonterminating constructs (infinite loops or resource waits that do not terminate) with a lock held, since other threads that need that lock will never be able to acquire it.

**Poor Responsiveness**

Eg : event thread an background thread in UI application.

Solition :Thrade priority of backgrounf should be lower.

Livelock

Livelock is a form of liveness failure in which a thread, while not blocked, still cannot make progress because it keeps retrying an operation that will always fail.

11 Performance and Scalability

GTK

1. When the performance of an activity is limited by availability of a particular resource, we say it is bound by that resource: CPU-bound, database-bound, etc.
2. Improving performance means doing more work with fewer resources
3. using multiple threads always introduces some performance costs compared to the single-threaded approach. These includes : coordinating between threads (locking, signaling, and memory synchronization), increased context switching, thread creation and teardown, and scheduling overhead

Concurrency and Performace

What we mean by performace in multi- threading :

1. Utilize the processing resources we have more effectively, and able our program to exploit additional processing resources if they become available. this means we are looking to keep the CPUs as busy as possible
2. If the program is compute-bound, then we may be able to increase its capacity by adding more processors; if it can't even keep the processors we have busy, adding more won't help

Performance Versus Scalability

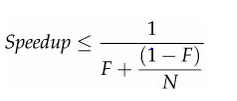
1. Scalability describes the ability to improve throughput or capacity when additional computing resources (such as additional CPUs, memory, storage, or I/O bandwidth) are added.
2. How much – scalability and how fast is performance (service time, latency)
3. Scalability often comes at the expense of performance

Example : A monolithic application vs a multi tier application Where presentation, business logic, and persistence are separated and may be handled by different systems.

1. monolithic application provides better performance thendistribted for limited scalability . It would not have the network latency inherent in handing off tasks between tiers, nor would it have to pay the costs inherent in separating a computational process into distinct abstracted layers (such as queuing overhead, coordination overhead, and data copying).
2. Avoid premature optimization. First make it right, then make it fastif it is not already fast enough
3. The quest for performance is probably the single greatest source of concurrency bugs
4. The belief that synchronization was "too slow" led to many clever-looking but dangerous idioms for reducing synchronization (such as double-checked locking

Amdahl's Law

Amdahl's law describes how much a program can theoretically be speed up by additional computing resources, based on the proportion of parallelizable and serial components. If F is the fraction of the calculation that must be executed serially, then Amdahl's law says that on a machine with N processors, we can achieve a speedup of at most:



As N approaches infinity, the maximum speedup converges to 1/F, meaning that a program in which fifty percent of the processing must be executed serially can be speed up only by a factor of two .

Program Below :

public class WorkerThread extends Thread {

private final BlockingQueue<Runnable> queue;

publicWorkerThread(BlockingQueue<Runnable> queue) {

this.queue = queue;

}

public void run() {

while (true) {

try {

Runnable task = queue.take();

task.run();

} catch (InterruptedException e) {

break; /\* Allow thread to exit \*/

}

}

}

}

Can it be made parallel ?by adding more threads which are taking task from the queue,

What can be the isssues

1. Threads have to call queue.takeserially , as queue is single resource.
2. The work queue is shared by all the worker threads, and it will require some amount of synchronization to maintain its integrity in the face of concurrent access.

**Costs Introduced by Threads**

When using multithading , we have to compare cost introduce by multiple threads in preserving consistency of data structure

**Context switching**

If the main thread is the only schedulable thread, it will almost never be scheduled out. On the other hand, if there are more runnable threads than CPUs, eventually the OS will preempt one thread so that another can use the CPU. This causes a context switch, which requires saving the execution context of the currently running thread and restoring the execution context of the newly scheduled thread.

A program that does more blocking (blocking I/O, waiting for contended locks, or waiting on condition variables) incurs more context switches than one that is CPU-bound, increasing scheduling overhead and reducing throughput

The vmstat command on Unix systems and the **perfmon** tool on Windows systems report the number of context switches and the percentage of time spent in the kernel

**Memory Synchronization**

The visibility guarantees provided by synchronized and volatile may entail using special instructions called memory barriers that can flush or invalidate caches

Memory barriers may also have indirect performance consequences because they inhibit other compiler optimizations; most operations cannot be reordered with memory barriers.

Synchronzation optimization by JVM (GTK)

Modern JVMs can reduce the cost of incidental synchronization by optimizing away locking that can be proven never to contend.

1. Don’t do this

synchronized (new Object()) {

// do something

}

1. More sophisticated JVMs can use escape analysis to identify when a local object reference is never published to the heap and is therefore thread-local.
2. In getStoogeNames the only reference to the List is the local variable stooges, and stack-confined variables are automatically thread-local

public String getStoogeNames() {

List<String> stooges = new Vector<String>();

stooges.add("Moe");

stooges.add("Larry");

stooges.add("Curly");

returnstooges.toString();

}

1. A naive execution of getStoogeNames would acquire and release the lock on the Vector four times, once for each call to add or toString.
2. However, a smart runtime compiler can inline these calls and then see that stooges and its internal state never escape, and therefore that all four lock acquisitions can be eliminated

**Blocking**

Suspending a thread because it could not get a lock, or because it blocked on a condition wait or blocking I/O operation, entails two additional context switches

the blocked thread is switched out before its quantum has expired, and is then switched back in later after the lock or other resource becomes available

**Reducing Lock Contention**

1. Narrowing Lock Scope ("Get in, Get Out")

@GuardedBy("this") private final Map<String, String>

attributes = new HashMap<String, String>();

public **synchronized** booleanuserLocationMatches(String name,

String regexp) {

String key = "users." + name + ".location";

String location = attributes.get(key);

if (location == null)

return false;

else

returnPattern.matches(regexp, location);

}

}

VS

Public Boolean userLocationMatches(String name, String regexp) {

String key = "users." + name + ".location";

String location;

**synchronized (this)** {

location = attributes.get(key);

}

if (location == null)

return false;

else

returnPattern.matches(regexp, location);

}

#### Reducing Lock Granularity

The other way to reduce the fraction of time that a lock is held (and therefore the likelihood that it will be contended) is to have threads ask for it less often. This can be accomplished by lock splitting and lock striping

**lock splitting**

One lock for the whole application :

1. thread will compete\contend for the same lock
2. Execution of sychroniztion will be serialized

If a lock guards more than one independent state variable, you may be able to improve scalability by splitting it into multiple locks that each guard different variables. This results in each lock being requested less often.

|  |  |
| --- | --- |
| One lock | Many Lock |
| public class ServerStatus {  @GuardedBy("this") public final Set<String> users;  @GuardedBy("this") public final Set<String> queries;  ...  public synchronized void addUser(String u) { users.add(u); }  public synchronized void addQuery(String q) { queries.add(q); }  public synchronized void removeUser(String u) {  users.remove(u);  }  public synchronized void removeQuery(String q) {  queries.remove(q);  }  } | @ThreadSafe  public class ServerStatus {  @GuardedBy("users") public final Set<String> users;  @GuardedBy("queries") public final Set<String> queries;  ...  public void addUser(String u) {  **synchronized (users)** {  users.add(u);  }  }  public void addQuery(String q) {  **synchronized (queries)** {  queries.add(q);  }  } |

1. Splitting a lock into two offers the greatest possibility for improvement when the lock is experiencing moderate but not heavy contention
2. it still does not dramatically improve prospects for concurrency on a system with many processors.
3. Lock splitting can sometimes be extended to partition locking on a variablesized set of independent objects, in which case it is called lock striping.

Example

1. implementation of ConcurrentHashMap uses an array of 16 locks, each of which guards 1/16 of the hash buckets; bucket N is guarded by lock N mod 16 (hash%16)
2. this should reduce the demand for any given lock by approximately a factor of 16. It is this technique that enables ConcurrentHashMap to support up to 16 concurrent writers
3. One of the downsides of lock striping is that locking the collection for exclusive access is more difficult and costly than with a single lock. Usually an operation can be performed by acquiring at most one lock, but occasionally you need to lock the entire collection, as when ConcurrentHashMap needs to expand the map and rehash the values into a larger set of buckets. This is typically done by acquiring all of the locks in the stripe se

public class StripedMap {

// Synchronization policy: buckets[n] guarded by locks[n%N\_LOCKS]

private static final int N\_LOCKS = 16;

private final Node[] buckets;

private final Object[] locks;

private static class Node { ... }

publicStripedMap(intnumBuckets) {

buckets = new Node[numBuckets];

locks = new Object[N\_LOCKS];

for (int i = 0; i < N\_LOCKS; i++)

locks[i] = new Object();

}

private final int hash(Object key) {

returnMath.abs(key.hashCode() % buckets.length);

}

public Object get(Object key) {

int hash = hash(key);

synchronized (locks[hash % N\_LOCKS]) {

for (Node m = buckets[hash]; m != null; m = m.next)

if (m.key.equals(key))

returnm.value;

}

return null;

}

public void clear() {

for (int i = 0; i <buckets.length; i++) {

synchronized (locks[i % N\_LOCKS]) {

buckets[i] = null;

}

}

}

1. Clearing Map in Concurrent Map will not be atomic operation, there is not necessarily a time when the Striped-Map is actually empty
2. making the operation atomic would require acquiring all the locks at once. However, for concurrent collections that clients typically cannot lock for exclusive access
3. the result of methods like size or isEmpty may be out of date by the time they return anyway, so this behavior, while perhaps somewhat surprising, is usually acceptable.

**Avoiding Hot Fields**

Counter in HashMap which counts num of elements can be consider a hot filed

1. One way to get th size of the Map is to count in size another is to keep a counter and update on add\remove. Which improve performance from O(n) to O(1).
2. Keeping a separate count to speed up operations like size and isEmpty works fine for a single-threaded or fully synchronized implementation, but makes it much harder to improve the scalability of the implementation because every operation that modifies the map must now update the shared counter.
3. if you use lock striping for the hash chains, synchronizing access to the counter reintroduces the scalability problems of exclusive locking.
4. What looked like a performance optimization caching the results of the size operation has turned into a scalability liability.
5. ConcurrentHashMap avoids this problem by having size enumerate the stripes and add up the number of elements in each stripe, instead of maintaining a global count. To avoid enumerating every element, ConcurrentHashMap maintains a separate count field for each stripe, also guarded by the stripe lock

Alternatives to Exclusive Locks

third technique for mitigating the effect of lock contention is to forego the use of exclusive locks in favor of a more concurrency-friendly means of managing shared state. These include using the concurrent collections, read-write locks, immutable objects and atomic variables

**Monitoring CPU Utilization**

When testing for scalability, the goal is usually to keep the processors fully utilized. Tools like vmstat and mpstat on Unix systems or perfmon on Windows systems can tell you just how "hot" the processors are running.

If the CPUs are asymmetrically utilized (some CPUs are running hot but others are not) your first goal should be to find increased parallelism in your program. Asymmetric utilization indicates that most of the computation is going on in a small set of threads, and your application will not be able to take advantage of additional processors.

If your application is keeping the CPUs sufficiently hot, you can use monitoring tools to infer whether it would benefit from additional CPUs.

A program with only four threads may be able to keep a 4-way system fully utilized, but is unlikely to see a performance boost if moved to an 8-way system since there would need to be waiting runnable threads to take advantage of the additional processors

One of the columns reported by **vmstat** is the number of threads that are runnable but not currently running because a CPU is not available; if CPU utilization is high and there are always runnable threads waiting for a CPU, your application would probably benefit from more processors.

**Just Say No to Object Pooling**

In early JVM versions, object allocation and garbage collection were slow,but their performance has improved substantially since then. In fact, allocation in Java is now faster than malloc is in C

To work around "slow" object lifecycles, many developers turned to object pooling, where objects are recycled instead of being garbage collected and allocated anew when needed. Even taking into account its reduced garbage collection overhead, object pooling has been shown to be a performance loss[[14]](mk:@MSITStore:E:\\Books\\Java\\Java%20Concurrency%20In%20Practice%20%5bDark%20Demon%5d%20%5bh33t%5d.chm::/0321349601/ch11lev1sec4.html" \l "ch11fn14) for all but the most expensive objects in single threaded program.

In addition to being a loss in terms of CPU cycles, object pooling has a number of other problems, among them the challenge of setting pool sizes correctly (too small, and pooling has no effect; too large, and it puts pressure on the garbage collector, retaining memory that could be used more effectively for something else); the risk that an object will not be properly reset to its newly allocated state, introducing subtle bugs; the risk that a thread will return an object to the pool but continue using it; and that it makes more work for generational garbage collectors by encouraging a pattern of old-to-young references.

In concurrent applications, pooling fares even worse. When threads allocate new objects, very little inter-thread coordination is required, as allocators typically use thread-local allocation blocks to eliminate most synchronization on heap data structures. But if those threads instead request an object from a pool, some synchronization is necessary to coordinate access to the pool data structure, creating the possibility that a thread will block. Because blocking a thread due to lock contention is hundreds of times more expensive than an allocation, even a small amount of pool-induced contention would be a scalability bottleneck

Comparing Map Performance

Concurrent Hash Map fares much better then synchronized hash Map uder multi thread env.

The major scalability impediment for the synchronized Map implementations is that there is a single lock for the entire map, so only one thread can access the map at a time.

On the other hand, ConcurrentHashMap does no locking for most successful read operations, and uses lock striping for write operations and those few read operations that do require locking.

As a result, multiple threads can access the Map concurrently without blocking.

**Testing Concurrent Programs**

1. The same techniques for testing correctness and performance in sequential programs can be applied to concurrent programs, but with concurrent programs the space of things that can go wrong is much larger.
2. The major challenge in constructing tests for concurrent programs is that potential failures may be rare probabalistic occurrences rather than deterministic ones; tests that disclose such failures must be more extensive and run for longer than typical sequential tests.

Most tests of concurrent classes fall into one or both of the classic categories of safety and liveness. We defined safety as "nothing bad ever happens" and liveness as "something good eventually happens".

1. Tests of safety, which verify that a class's behavior conforms to its specification, usually take the form of testing invariants. Eg size of list
2. Liveness tests include tests of progress and nonprogress
3. how do you verify that a method is blocking and not merely running slowly?
4. how do you test that an algorithm does not deadlock? How long should you wait before you declare it to have failed?

|  |
| --- |
| Related to liveness tests are performance tests. Performance can be measured in a number of ways, including:   * Throughput: the rate at which a set of concurrent tasks is completed; * Responsiveness: the delay between a request for and completion of some action (also called latency); or * Scalability: the improvement in throughput (or lack thereof) as more resources (usually CPUs) are made available.   **Testing for Correctness**  Bounded Buffer Using Semaphore is the sample program for testing    **Testing Blocking Operations** |

1. If a method is supposed to block under certain conditions, then a test for that behavior should succeed only if the thread does not proceed.
2. Testing that a method blocks is similar to testing that a method throws an exception; if the method returns normally, the test has failed.
3. once the method successfully blocks, you have to convince it somehow to unblock. The obvious way to do this is via interruption.
4. start a blocking activity in a separate thread, wait until the thread blocks, interrupt it, and then assert that the blocking operation completed
5. The test runner thread starts the **taker** thread (below), waits a long time, and then interrupts it. If the taker thread has correctly blocked in the take operation, it will throw InterruptedException, and the catch block for this exception treats this as success and lets the thread exit

voidtestTakeBlocksWhenEmpty() {

finalBoundedBuffer<Integer> bb = new BoundedBuffer<Integer>(10);

Thread taker = new Thread() {

public void run() {

try {

int unused = bb.take();

fail(); // if we get here, it's an error

} catch (InterruptedException success) { }

}};

try {

taker.start();

Thread.sleep(LOCKUP\_DETECT\_TIMEOUT);

taker.interrupt();

taker.join(LOCKUP\_DETECT\_TIMEOUT);

assertFalse(taker.isAlive());

} catch (Exception unexpected) {

fail();

}

}

1. It is tempting to use Thread.getState to verify that the thread is actually blocked on a condition wait, but this approach is not reliable.
2. There is nothing that requires a blocked thread ever to enter the WAITING or TIMED\_WAITING states, since the JVM can choose to implement blocking by spin-waiting instead.
3. Similarly, because spurious wakeups from Object.wait or Condition.await are permitted , a thread in the WAITING or TIMED\_WAITING state may temporarily transition to RUNNABLE even if the condition for which it is waiting is not yet true

**Spurious wakeups**

<http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#spurious-wakeups>

<http://stackoverflow.com/questions/1050592/do-spurious-wakeups-actually-happen>

1. For inexplicable reasons it is possible for threads to wake up even if notify() and notifyAll() has not been called. This is known as spurious wakeups. Wakeups without any reason
2. To guard against spurious wakeups the signal member variable is checked inside a while loop instead of inside an if-statement

|  |  |
| --- | --- |
|  |  |
| **while(!wasSignalled)**{  try{  myMonitorObject.wait();  } catch(InterruptedException e){...}  }…  public void doNotify(){  synchronized(myMonitorObject){  wasSignalled = true;  myMonitorObject.notify();  }  } | synchronized(myMonitorObject){  **if(!wasSignalled){**  try{  myMonitorObject.wait();  } catch(InterruptedException e){...}  } |

1. Notice how the wait() call is now nested inside a while loop instead of an if-statement. If the waiting thread wakes up without having received a signal, the wasSignalled member will still be false, and the while loop will execute once more, causing the awakened thread to go back to waiting.

Testing Safety

The challenge to constructing effective safety tests for concurrent classes is identifying easily checked properties that will, with high probability, fail if something goes wrong, while at the same time not letting the failureauditing code limit concurrency artificially

1. One approach that works well with classes used in producer-consumer designs (like BoundedBuffer) is to check that everything put into a queue or buffer comes out of it, and that nothing else does.

Testing Resource management - Memory

A secondary aspect to test is that it does not do things it is not supposed to do, such as leak resources. Any object that holds or manages other objects should not continue to maintain references to those objects longer than necessary.

Undesirable memory retention can be easily tested with heap-inspection tools that measure application memory usage; a variety of commercial and open-source heap-profiling tools can do this

voidtestLeak() throws InterruptedException {

BoundedBuffer<Big> bb = new BoundedBuffer<Big>(CAPACITY);

int heapSize1 = /\* snapshot heap \*/ ;

for (int i = 0; i < CAPACITY; i++)

bb.put(new Big());

for (int i = 0; i < CAPACITY; i++)

bb.take();

int heapSize2 = /\* snapshot heap \*/ ;

assertTrue(Math.abs(heapSize1-heapSize2) < THRESHOLD);

Generating More Interleavings

Since many of the potential failures in concurrent code are low-probability events, testing for concurrency errors is a numbers game, but there are some things you can do to improve your chances.

We've already mentioned how running on multiprocessor systems with fewer processors than active threads can generate more interleavings than either a single-processor system or one with many processors.

A useful trick for increasing the number of interleavings, and therefore more effectively exploring the state space of your programs, is to use THRead.yield to encourage more context switches during operations that access shared state

public synchronized void transferCredits(Account from,

Account to,

int amount) {

from.setBalance(from.getBalance() - amount);

if (random.nextInt(1000) > THRESHOLD)

Thread.yield();

to.setBalance(to.getBalance() + amount);

}

Testing for Performance

Performance tests are often extended versions of functionality tests.

While there is definitely overlap between performance and functionality tests, they have different goals. Performance tests seek to measure end-to-end performance metrics for representative use cases.

Measure time taken for operations

Building Custom Synchronizers

1. state-dependent classes : FutureTask, Semaphore, and BlockingQueue.
2. For example, you cannot remove an item from an empty queue or retrieve the result of a task that has not yet finished; before these operations can proceed, you must wait until the queue enters the "nonempty" state or the task enters the "completed" state.
3. in a concurrent program, state-based conditions can change through the actions of other threads: a pool that was empty a few instructions ago can become nonempty because another thread returned an element..
4. blocking state-dependent action takes the form shown in below
5. The pattern of locking is somewhat unusual in that the lock is released and reacquired in the middle of the operation.
6. The state variables that make up the precondition must be guarded by the object's lock, so that they can remain constant while the precondition is tested.
7. But if the precondition does not hold, the lock must be released so another thread can modify the object stateotherwise the precondition will never become true. The lock must then be reacquired before testing the precondition again.

voidblockingAction() throws InterruptedException {

acquire lock on object state

while (precondition does not hold) {

release lock

wait until precondition might hold

optionally fail if interrupted or timeout expires

reacquire lock

}

perform action

}

1. State dependent operations can deal with precondition failure by :
2. throwing an exception or
3. returning an error status (making it the caller's problem), or
4. by blocking until the object transitions to the right state.



public abstract class BaseBoundedBuffer<V> {

protected synchronized final void doPut(V v) {

buf[tail] = v;

if (++tail == buf.length)

tail = 0;

++count;

}

protected synchronized final V doTake() {

V v = buf[head];

buf[head] = null;

if (++head == buf.length)

head = 0;

--count;

return v;

}

public synchronized final boolean**isFull**() {

return count == buf.length;

}

public synchronized final boolean**isEmpty**() {

return count == 0;

}

}

1. 1st attempt : Propagating Precondition Failure to Callers
2. The put and take methods are synchronized to ensure exclusive access to the buffer state, since both employ check-then-act logic in accessing the buffer
3. While this approach is easy enough to implement, it is annoying to use
4. Exceptions are supposed to be for exceptional conditions .
5. "Buffer is full" is not an exceptional condition for a bounded buffer any more than "red" is an exceptional condition for a traffic signal

public class GrumpyBoundedBuffer<V> extends BaseBoundedBuffer<V> {

publicGrumpyBoundedBuffer(int size) { super(size); }

public**synchronized** void put(V v) throws BufferFullException {

if (isFull())

**throw new BufferFullException();**

doPut(v);

}

public synchronized V take() throws BufferEmptyException {

if (isEmpty())

throw new BufferEmptyException();

returndoTake();

}

}

Client Code :

1. The client code below is not the only way to implement the retry logic.
2. The caller could retry the take immediately, without sleeping an approach known as busy waiting or spin waiting. This could consume quite a lot of CPU time if the buffer state does not change for a while.
3. On the other hand, if the caller decides to sleep so as not to consume so much CPU time, it could easily "oversleep" if the buffer state changes shortly after the call to sleep.
4. So the client code is left with the choice between the poor CPU usage of spinning and the poor responsiveness of sleeping.
5. Somewhere between busy waiting and sleeping would be calling Thread.yield in each iteration, which is a hint to the scheduler that this would be a reasonable time to let another thread run.

Client Logic for Calling GrumpyBoundedBuffer

while (true) {

try {

V item = buffer.take();

// use item

break;

} catch (BufferEmptyException e) {

Thread.sleep(SLEEP\_GRANULARITY);

}}

**Bounded Buffer Using Crude Blocking**

SleepyBoundedBuffer attempts to spare callers the inconvenience of implementing the retry logic on each call by encapsulating the same crude "poll and sleep" retry mechanism within the put and take operations.

If the buffer is empty, take sleeps until another thread puts some data into the buffer; if the buffer is full, put sleeps until another thread makes room by removing some data. This approach encapsulates precondition management and simplifies using the buffered finitely a step in the right direction.

public class SleepyBoundedBuffer<V> extends BaseBoundedBuffer<V> {

publicSleepyBoundedBuffer(int size) { super(size); }

public void put(V v) throws InterruptedException {

while (true) {

synchronized (this) {

if (!isFull()) {

doPut(v);

return;

}

}

Thread.sleep(SLEEP\_GRANULARITY);

}

}

public V take() throws InterruptedException {

while (true) {

synchronized (this) {

if (!isEmpty())

returndoTake();

}

Thread.sleep(SLEEP\_GRANULARITY);

}

}

}

From the perspective of the caller, this works nicelyif the operation can proceed immediately, it does, and otherwise it blocksand the caller need not deal with the mechanics of failure and retry. Choosing the sleep granularity is a tradeoff between responsiveness and CPU usage; the smaller the sleep granularity, the more responsive, but also the more CPU resources consumed.

**Condition Queues** to the Rescue (Waiit, notify andnotifyAll)

A condition queue gets its name because it gives a group of threads called the wait set a way to wait for a specific condition to become true. Unlike typical queues in which the elements are data items, the elements of a condition queue are the threads waiting for the condition.

Good Analogy:

Condition queues are like the "toast is ready" bell on your toaster. If you are listening for it, you are notified promptly when your toast is ready and can drop what you are doing (or not, maybe you want to finish the newspaper first) and get your toast. If you are not listening for it (perhaps you went outside to get the newspaper), you could miss the notification, but on return to the kitchen you can observe the state of the toaster and either retrieve the toast if it is finished or start listening for the bell again if it is not.

Wait\Notify in synchronized block

1. Just as each Java object can act as a lock, each object can also act as a condition queue, and the wait, notify, and notifyAll methods in Object constitute the API for intrinsic condition queues.
2. An object's intrinsic lock and its intrinsic condition queue are related: in order to call any of the condition queue methods on object X, you must hold the lock on X.
3. This is because the mechanism for waiting for state-based conditions is necessarily tightly bound to the mechanism for preserving state consistency: you cannot wait for a condition unless you can examine the state, and you cannot release another thread from a condition wait unless you can modify the state.

Object.wait atomically releases the lock and asks the OS to suspend the current thread, allowing other threads to acquire the lock and therefore modify the object state. Upon waking, it reacquires the lock before returning.

Intuitively, calling wait means "I want to go to sleep, but wake me when something interesting happens", and calling the notification methods means "something interesting happened".

ThreadSafe

public class BoundedBuffer<V> extends BaseBoundedBuffer<V> {

// CONDITION PREDICATE: not-full (!isFull())

// CONDITION PREDICATE: not-empty (!isEmpty())

publicBoundedBuffer(int size) { super(size); }

// BLOCKS-UNTIL: not-full

public synchronized void put(V v) throws InterruptedException {

while (isFull())

wait();

doPut(v);

notifyAll();

}

// BLOCKS-UNTIL: not-empty

public synchronized V take() throws InterruptedException {

while (isEmpty())

wait();

V v = doTake();

notifyAll();

return v;

}

}

BoundedBuffer in [Listing 14.6](mk:@MSITStore:E:\Books\Java\Java%20Concurrency%20In%20Practice%20%5bDark%20Demon%5d%20%5bh33t%5d.chm::/0321349601/ch14lev1sec1.html#ch14list06) implements a bounded buffer using wait and notifyAll. This is simpler than the sleeping version, and is both more efficient (waking up less frequently if the buffer state does not change) and more responsive (waking up promptly when an interesting state change happens). This is a big improvement

**Using Condition Queues**

Condition queues make it easier to build efficient and responsive state-dependent classes, but they are still easy to use incorrectly; there are a lot of rules regarding their proper use that are not enforced by the compiler or platform.

**The Condition Predicate**

The key to using condition queues correctly is identifying the condition predicates that the object may wait for

condition predicate is the precondition that makes an operation state-dependent in the first place. In a bounded buffer, take can proceed only if the buffer is not empty; otherwise it must wait. For take, the condition predicate is "the buffer is not empty", which take must test for before proceeding.

There is an important three-way relationship in a condition wait involving locking, the wait method, and a condition predicate. The condition predicate involves state variables, and the state variables are guarded by a lock, so before testing the condition predicate, we must hold that lock. The lock object and the condition queue object (the object on which wait and notify are invoked) must also be the same object.

Waking Up Too Soon

1. wait returns does not necessarily mean that the condition predicate the thread is waiting for has become true.
2. When your thread is awakened because someone called notifyAll, that doesn't mean that the condition predicate you were waiting for is now true.
3. This is like having your toaster and coffee maker share a single bell; when it rings, you still have to look to see which device raised the signal

When control re-enters the code calling wait, it has reacquired the lock associated with the condition queue. Is the condition predicate now true?

1. Maybe
2. It might have been true at the time the notifying thread called notifyAll, but could have become false again by the time you reacquire the lock.
3. Other threads may have acquired the lock and changed the object's state between when your thread was awakened and when wait reacquired the lock. Or maybe it hasn't been true at all since you called wait.
4. You don't know why another thread called notify or notifyAll; maybe it was because another condition predicate associated with the same condition queue became true

For all these reasons, when you wake up from wait you must test the condition predicate again, and go back to waiting (or fail) if it is not yet true. Since you can wake up repeatedly without your condition predicate being true, you must therefore always call wait from within a loop, testing the condition predicate in each iteration

When using condition waits (Object.wait or Condition.await):

* Always have a condition predicatesome test of object state that must hold before proceeding;
* Always test the condition predicate before calling wait, and again after returning from wait;
* Always call wait in a loop;
* Ensure that the state variables making up the condition predicate are guarded by the lock associated with the condition queue;
* Hold the lock associated with the the condition queue when calling wait, notify, or notifyAll; and
* Do not release the lock after checking the condition predicate but before acting on it.

Missed Signals

While loop will prevent missed single

A missed signal occurs when a thread must wait for a specific condition that is already true, but fails to check the condition predicate before waiting. Now the thread is waiting to be notified of an event that has already occurred.

Toast analogy : you are outside and toast is ready , you missed the beep. Now u r waiting for the beep.

**Notification**

1. There are two notification methods in the condition queue APInotify and notifyAll. To call either, you must hold the lock associated with the condition queue object
2. Calling notify causes the JVM to select one thread waiting on that condition queue to wake up; calling notifyAll wakes up all the threads waiting on that condition queue
3. Because you must hold the lock on the condition queue object when calling notify or notifyAll, and waiting threads cannot return from wait without reacquiring the lock, the notifying thread should release the lock quickly to ensure that the waiting threads are unblocked as soon as possible.
4. Because multiple threads could be waiting on the same condition queue for different condition predicates, using notify instead of notifyAll can be dangerous, primarily because single notification is prone to a problem akin to missed signals
5. **Use NotifyAll then notify**

Single notify can be used instead of notifyAll only when both of the following conditions hold:

Uniform waiters. Only one condition predicate is associated with the condition queue, and each thread executes the same logic upon returning from wait; and

One-in, one-out. A notification on the condition variable enables at most one thread to proceed.

Most classes don't meet these requirements, so the prevailing wisdom is to use notifyAll in preference to single notify. While this may be inefficient, it is much easier to ensure that your classes behave correctly when using notifyAll instead of notify

Disadvantage of using notify All

1. Using notifyAll when only one thread can make progress is inefficientsometimes a little
2. If ten threads are waiting on a condition queue, calling notifyAll causes each of them to wake up and contend for the lock; then most or all of them will go right back to sleep
3. This means a lot of context switches and a lot of contended lock acquisitions for each event that enables (maybe) a single thread to make progress.

Improving notification

Use conditional notification

public synchronized void put(V v) throws InterruptedException {

while (isFull())

wait();

booleanwasEmpty = isEmpty();

doPut(v);

if (wasEmpty)

notifyAll();

}

Subclass Safety Issues

**A state-dependent class should either fully expose (and document) its waiting and notification protocols to subclasses, or prevent subclasses from participating in them at all.**

At the very least, designing a state-dependent class for inheritance requires exposing the condition queues and locks and documenting the condition predicates and synchronization policy; it may also require exposing the underlying state variables

**Explicit Condition Objects**

Issue with intrinsic condition queue :

Intrinsic condition queues have several drawbacks. Each intrinsic lock can have only one associated condition queue, which means that in classes like BoundedBuffer multiple threads might wait on the same condition queue for different condition predicates, and the most common pattern for locking involves exposing the condition queue object.

public interface Condition {

void await() throws InterruptedException;

boolean await(long time, TimeUnit unit)

throwsInterruptedException;

longawaitNanos(long nanosTimeout) throws InterruptedException;

voidawaitUninterruptibly();

booleanawaitUntil(Date deadline) throws InterruptedException;

void signal();

voidsignalAll();

}

Unlike intrinsic condition queues, you can have as many Condition objects per Lock as you want.

The equivalents of wait, notify, and notifyAll for Condition objects are await, signal, and signalAll. However, Condition extends Object, which means that it also has wait and notify methods. Be sure to use the proper versionsawait and signalinstead!

Choose between using explicit Conditions and intrinsic condition queues in the same way as you would choose between ReentrantLock and synchronized: use Condition if you need its advanced features such as fair queueing or multiple wait sets per lock, and otherwise prefer intrinsic condition queues

protected final Lock lock = new ReentrantLock();

private final Condition notFull = lock.newCondition();

private final Condition notEmpty = lock.newCondition();

public void put(T x) throws InterruptedException {

lock.lock();

try {

while (count == items.length)

notFull.await();

items[tail] = x;

if (++tail == items.length)

tail = 0;

++count;

notEmpty.signal();

} finally {

lock.unlock();

}

}

// BLOCKS-UNTIL: notEmpty

public T take() throws InterruptedException {

lock.lock();

try {

while (count == 0)

notEmpty.await();

T x = items[head];

items[head] = null;

if (++head == items.length)

head = 0;

--count;

notFull.signal();

return x;

} finally {

lock.unlock();

}

}

Anatomy of a Synchronizer

Lock and Semphore has much similarity like lock and acquire and try lock and try acquiring.

We can implement lock using Semaphore or Semaphore using lock.

But they are both implemented using a common base class, Abstract-QueuedSynchronizer (AQS)as are many other synchronizers

all the synchronizers in java.util.concurrent that are built with AQS benefit from this.

AbstractQueuedSynchronizer

Nonblocking algorithms are used extensively in operating systems and JVMs for thread and process scheduling, garbage collection, and to implement locks and other concurrent data structures.

Nonblocking algorithms are considerably more complicated to design and implement than lock-based alternatives, but they can offer significant scalability and liveness advantages.

Atomic variables can also be used as "better volatile variables" even if you are not developing nonblocking algorithms

**Disadvantages of Locking**

Thread Suspension due to unavailability of lock : Modern JVMs can optimize uncontended lock acquisition and release fairly effectively, but if multiple threads request the lock at the same time the JVM enlists the help of the operating system. If it gets to this point, some unfortunate thread will be suspended and have to be resumed later.[[1]](mk:@MSITStore:E:\books\Java\Java%20Concurrency%20In%20Practice%20%5bDark%20Demon%5d%20%5bh33t%5d.chm::/0321349601/ch15lev1sec1.html#ch15fn01) When that thread is resumed, it may have to wait for other threads to finish their scheduling quanta before it is actually scheduled. Suspending and resuming a thread has a lot of overhead and generally entails a lengthy interruption

Volatile variables are a lighter-weight synchronization mechanism than locking because they do not involve context switches or thread scheduling. However, volatile variables have some limitations compared to locking: while they provide similar visibility guarantees, they cannot be used to construct atomic compound actions.

For example, while the increment operation (++i) may look like an atomic operation, it is actually three distinct operationsfetch the current value of the variable, add one to it, and then write the updated value back. In order to not lose an update, the entire read-modify-write operation must be atomic

Locking has a few other disadvantages. When a thread is waiting for a lock, it cannot do anything else. If a thread holding a lock is delayed (due to a page fault, scheduling delay, or the like), then no thread that needs that lock can make progress. This can be a serious problem if the blocked thread is a high-priority thread but the thread holding the lock is a lower-priority threada performance hazard known as priority inversion.

Hardware Support for Concurrency

Exclusive locking is a pessimistictechniqueit assumes the worst

optimistic approach, whereby you proceed with an update, hopeful that you can complete it without interference. This approach relies on collision detection to determine if there has been interference from other parties during the update, in which case the operation fails and can be retried (or not).

The optimistic approach is like the old saying, "It is easier to obtain forgiveness than permission", where "easier" here means "more efficient".

**Non-blocking Concurrency Algorithms**

A non-blocking concurrency algorithm is an algorithm which either:

* A: Performs the action requested by the thread - OR
* B: Notifies the requesting thread that the action could not be performed

Java contains several non-blocking data structures too. The [**AtomicBoolean**](http://tutorials.jenkov.com/java-util-concurrent/atomicboolean.html), [**AtomicInteger**](http://tutorials.jenkov.com/java-util-concurrent/atomicinteger.html),**[AtomicLong](http://tutorials.jenkov.com/java-util-concurrent/atomiclong.html)** and [**AtomicReference**](http://tutorials.jenkov.com/java-util-concurrent/atomicreference.html) are all examples of non-blocking data structures

From :<http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html>

**Compare and Swap**

The approach taken by most processor architectures, including IA32 and Sparc, is to implement a compare-and-swap (CAS) instruction

CAS means "I think V should have the value A; if it does, put B there, otherwise don't change it but tell me I was wrong." CAS is an optimistic techniqueit proceeds with the update in the hope of success, and can detect failure if another thread has updated the variable since it was last examined

if a CAS-like instruction is not available the JVM uses a spin lock. This low-level JVMsupport is used by the atomic variable classes (AtomicXxx in java.util.concurrent. atomic) to provide an efficient CAS operation on numeric and reference types;

ThreadSafe

public class SimulatedCAS {

@GuardedBy("this") private int value;

public synchronized int get() { return value; }

public synchronized intcompareAndSwap(intexpectedValue,

intnewValue) {

intoldValue = value;

if (oldValue == expectedValue)

value = newValue;

returnoldValue;

}

public synchronized booleancompareAndSet(intexpectedValue,

intnewValue) {

return (expectedValue

== compareAndSwap(expectedValue, newValue));

}

}

@ThreadSafe

public class CasCounter {

privateSimulatedCAS value;

publicintgetValue() {

returnvalue.get();

}

publicint increment() {

int v;

do {

v = value.get();

}

while (v != value.compareAndSwap(v, v + 1));

return v + 1;

}

}

Atomic Variable Classes

AtomicInteger bears a superficial resemblance to an extended Counter class, but offers far greater scalability under contention because it can directly exploit underlying hardware support for concurrency

The atomic array classes (available in Integer, Long, and Reference versions) are arrays whose elements can be updated atomically. The atomic array classes provide volatile access semantics to the elements of the array, a feature not available for ordinary arraysavolatile array has volatile semantics only for the array reference, not for its elements.

they do not extend the primitive wrapper classes such as Integer or Long. In fact, they cannot: the primitive wrapper classes are immutable whereas the atomic variable classes are mutable

Atomics as "Better Volatiles

private final AtomicReference<IntPair> values =

newAtomicReference<IntPair>(new IntPair(0, 0));

publicintgetLower() { return values.get().lower; }

publicintgetUpper() { return values.get().upper; }

public void setLower(int i) {

while (true) {

IntPairoldv = values.get();

if (i >oldv.upper)

throw new IllegalArgumentException(

"Can't set lower to " + i + " > upper");

IntPairnewv = new IntPair(i, oldv.upper);

if (values.compareAndSet(oldv, newv))

return;

}

}

Nonblocking Algorithms

Lock-based algorithms are at risk for a number of liveness failures. If a thread holding a lock is delayed due to blocking I/O, page fault, or other delay, it is possible that no thread will make progress.

An algorithm is called nonblocking if failure or suspension of any thread cannot cause failure or suspension of another thread; an algorithm is called lock-free if, at each step, some thread can make progress.

Stacks are the simplest linked data structure: each element refers to only one other element and each element is referred to by only one object reference. ConcurrentStack shows how to construct a stack using atomic references.

A Nonblocking Stack

@ThreadSafe

public class ConcurrentStack<E> {

AtomicReference<Node<E>> top = new AtomicReference<Node<E>>();

public void push(E item) {

Node<E>newHead = new Node<E>(item);

Node<E>oldHead;

do {

oldHead = top.get();

newHead.next = oldHead;

} while (!top.compareAndSet(oldHead, newHead));

}

public E pop() {

Node<E>oldHead;

Node<E>newHead;

do {

oldHead = top.get();

if (oldHead == null)

return null;

newHead = oldHead.next;

} while (!top.compareAndSet(oldHead, newHead));

returnoldHead.item;

}

private static class Node <E> {

public final E item;

public Node<E> next;

public Node(E item) {

this.item = item;

}

}

}

From :<http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html>

public class AtomicCounter {

privateAtomicLong count = new AtomicLong(0);

public void inc() {

boolean updated = false;

while(!updated){

longprevCount = this.count.get();

updated = this.count.compareAndSet(prevCount, prevCount + 1);

}

}

public long count() {

returnthis.count.get();

}

}

These lines are not an atomic operation. That means, that it is possible for two different threads to call theinc() method and execute the long prevCount = this.count.get() statement, and thus both obtain the previous count for the counter. Yet, the above code does not contain any race conditions.

### Why is it Called Optimistic Locking?

The code shown in the previous section is called *optimistic locking*. Optimistic locking is different from traditional locking, sometimes also called pessimistic locking. Traditional locking blocks the access to the shared memory with a synchronized block or a lock of some kind. A synchronized block or lock may result in threads being suspended.

Optimistic locking allows all threads to create a copy of the shared memory without any blocking. The threads may then make modifications to their copy, and attempt to write their modified version back into the shared memory.

In addition to Java's built-in non-blocking data structures there are also some open source non-blocking data structures you can use. For instance, the LMAX Disrupter (a queue-like data structure), and the non-blocking HashMap from Cliff Click

**Ch16 What is a Memory Model, and Why would I Want One**

1. A memory model addresses the question "Under what conditions does a thread that reads aVariable see the value 3?"

aVariable = 3;

1. Compilers may generate instructions in a different order than the "obvious" one suggested by the source code, or store variables in registers instead of in memory;
2. These factors can prevent a thread from seeing the most up-to-date value for a variable and can cause memory actions in other threads to appear to happen out of order.
3. The JMM specifies the minimal guarantees the JVM must make about when writes to variables become visible to other threads
4. An architecture's memory model tells programs what guarantees they can expect from the memory system, and specifies the special instructions required In order to shield the Java developer from the differences between memory models across architectures, Java provides its own memory model,

Happens Before

Even though actions are only partially ordered, synchronization actionslock acquisition and release, and reads and writes of volatilevariablesare totally ordered. This makes it sensible to describe happens-before in terms of "subsequent" lock acquisitions and reads of volatile variables.

When two threads synchronize on different locks, we can't say anything about the ordering of actions between themthere is no happens-before relation between the actions in the two threads

Unsafe Publication in Lazy intialiation

@NotThreadSafe

public class UnsafeLazyInitialization {

private static Resource resource;

public static Resource getInstance() {

if (resource == null)

resource = new Resource(); // unsafe publication

return resource;

}

}

another thread could observe a reference to a partially constructed Resource.

Suppose thread A is the first to invoke getInstance. It sees that resource is null, instantiates a new Resource, and sets resource to reference it. When thread B later calls getInstance, it might see that resource already has a non-null value and just use the already constructed Resource

So even though A initialized the Resource before setting resource to reference it, B could see the write to resource as occurring before the writes to the fields of the Resource. B could thus see a partially constructed Resource that may well be in an invalid stateand whose state may unexpectedly change later.

Safe Publication

@ThreadSafe

public class SafeLazyInitialization {

private static Resource resource;

public synchronized static Resource getInstance() {

if (resource == null)

resource = new Resource();

return resource;

}

}

@ThreadSafe

public class EagerInitialization {

private static Resource resource = new Resource();

public static Resource getResource() { return resource; }

}

Why no synch is need at the time of initialization

The treatment of static fields with initializers is somewhat special and offers additional thread-safety guarantees.

Static initializers are run by the JVM at class initialization time, after class loading but before the class is used by any thread. Because the JVM acquires a lock during initialization and this lock is acquired by each thread at least once to ensure that the class has been loaded, memory writes made during static initialization are automatically visible to all threads. Thus statically initialized objects require no explicit synchronization either during construction or when being referenced

Double-checked Locking

The common code pathfetching a reference to an already constructed Resourcedoesn't use synchronization. And that's where the problem is.

The real problem with DCL is the assumption that the worst thing that can happen when reading a shared object reference without synchronization is to erroneously see a stale value (in this case, null);

|  |
| --- |
|  |

But the worst case is actually considerably worseit is possible to see a current value of the reference but stale values for the object's state, meaning that the object could be seen to be in an invalid or incorrect state.

@NotThreadSafe

public class DoubleCheckedLocking {

private static Resource resource;

public static Resource getInstance() {

if (resource == null) {

synchronized (DoubleCheckedLocking.class) {

if (resource == null)

resource = new Resource();

}

}

return resource;

}

}

Initialization Safety

Initialization safety means that SafeStates in [Listing 16.8](mk:@MSITStore:E:\books\Java\Java%20Concurrency%20In%20Practice%20%5bDark%20Demon%5d%20%5bh33t%5d.chm::/0321349601/ch16lev1sec3.html#ch16list08) could be safely published even through unsafe lazy initialization or stashing a reference to a SafeStates in a public static field with no synchronization, even though it uses no synchronization and relies on the non-thread-safe HashSet

@ThreadSafe

public class SafeStates {

private final Map<String, String> states;

publicSafeStates() {

states = new HashMap<String, String>();

states.put("alaska", "AK");

states.put("alabama", "AL");

...

states.put("wyoming", "WY");

}

public String getAbbreviation(String s) {

returnstates.get(s);

}

}