Best programming approach/methodology to assure thread safety

Asked 16 years, 1 month ago Modified 9 years, 11 months ago

Viewed 2k times

8

When I was learning Java coming from a background of some 20 years of procedural programming with basic, Pascal, COBOL and C, I thought at the time that the hardest thing about it was wrapping my head around the OOP jargon and concepts. Now with about 8 years of solid Java under my belt, I have come to the conclusion that the single hardest thing about programming in Java and similar languages like C# is the multithreaded/concurrent aspects.

Coding reliable and scalable multi-threaded applications is just plain hard! And with the trend for processors to grow "wider" rather than faster, it is rapidly becoming just plain critical.

The hardest area is, of course, controlling interactions between threads and the resulting bugs: deadlocks, race conditions, stale data and latency.

So my question to you is this: what approach or methodology do *you* employ for producing safe concurrent code while mitigating the potential for deadlocks, latency, and other problems? I have come up with an approach which is a little unconventional but has worked very well in several large applications, which I will share in a detailed answer to this question.

multithreading multicore concurrency

Share

Improve this question

Follow

edited Nov 6, 2008 at 19:41

asked Nov 2, 2008 at 4:45

L. Cornelius Dol
63.9k ● 26 ● 141 ● 189

15 Answers

Sorted by: Highest score (default)

7

This not only applies to Java but to threaded programming in general. I find myself avoiding most of the concurrency and latency problems just by following these guidelines:

1/ Let each thread run its own lifetime (i.e., decide when to die). It can be prompted from outside (say a flag variable) but it in entirely responsible.

2/ Have **all** threads allocate and free their resources in the same order - this guarantees that deadlock will not happen.

3/ Lock resources for the shortest time possible.

4/ Pass responsibility for data with the data itself - once you notify a thread that the data is its to process, **leave it alone** until the responsibility is given back to you.

Share Improve this answer Follow

answered Nov 2, 2008 at 5:10

paxdiablo

880k ● 241 ● 1.6k ● 2k

6

There are a number of techniques which are coming into the public consciousness just now (as in: the last few years). A big one would be actors. This is something that Erlang first brought to the grid iron but which has been carried forward by newer languages like Scala (actors on the JVM). While it is true that actors don't solve every problem, they do make it *much* easier to reason about your code and identify trouble spots. They also make it much simpler to design parallel algorithms because of the way they force you to use continuation passing over shared mutable state.

Fork/Join is something you should look at, especially if you're on the JVM. Doug Lea wrote the seminal paper on the topic, but many researchers have discussed it over the years. As I understand it, Doug Lea's reference framework is scheduled for inclusion into Java 7.

On a slightly less-invasive level, often the only steps necessary to simplify a multi-threaded application are just to reduce the complexity of the locking. Fine-grained locking (in the Java 5 style) is great for throughput, but very very difficult to get right. One alternative approach to locking which is gaining some traction through Clojure would be software-transactional memory (STM). This is essentially the opposite of conventional locking in that it is optimistic rather than pessimistic. You start out by assuming that you won't have any collisions, and then allow the framework to fix the problems if and when they occur. Databases often work this way. It's great for throughput on systems with low collision rates, but the big win is in the logical componentization of your algorithms. Rather than arbitrarily associating a lock (or a series of locks) with some data, you just wrap the dangerous code in a transaction and let the framework figure out the rest. You can even get a fair bit of compile-time checking out of decent STM implementations like GHC's STM monad or my experimental Scala STM.

There are a lot of new options for building concurrent applications, which one you pick depends greatly on your expertise, your language and what sort of problem you're trying to model. As a general rule, I think actors coupled with persistent, immutable data structures are a solid bet, but as I said, STM is a little less invasive and can sometimes yield more immediate improvements.

Share Improve this answer Follow

answered Nov 2, 2008 at 6:23

Daniel Spiewak
55.1k ● 14 ● 111 ● 120

5

1. Avoid sharing data between threads where possible (copy everything).
2. Never have locks on method calls to external objects, where possible.
3. Keep locks for the shortest amount of time possible.

Share Improve this answer Follow

answered Nov 2, 2008 at 4:57

jonnii
28.3k ● 8 ● 82 ● 110

I would change the parenthetical remark in point 1. from (copy everything) to (copy or transfer everything).
– Stephen C. Steel Jun 25, 2010 at 21:08

5

There is no **One True Answer** for thread safety in Java. However, there is at least one really great book: Java Concurrency in Practice. I refer to it regularly (especially the online Safari version when I'm on travel).

I strongly recommend that you peruse this book in depth. You may find that the costs and benefits of your unconventional approach are examined in depth.

Share Improve this answer Follow

answered Nov 2, 2008 at 4:58

Bob Cross
22.3k ● 12 ● 62 ● 95

4

I typically follow an Erlang style approach. I use the Active Object Pattern. It works as follows.

Divide your application into very coarse grained units. In one of my current applications (400.000 LOC) I have appr. 8 of these coarse grained units. **These units share no data at all. Every unit keeps its own local data. Every unit runs on its own thread** (= Active Object Pattern) and hence is single threaded. You don't need any locks within the units. When the units need to send messages to other units they do it by posting a message to a queue of the other units. The other unit picks the message from the queue and reacts on that message. This might trigger other messages to other units. Consequently the only locks in this type of application are around the queues (one queue and lock per unit). **This architecture is deadlock free by definition!**

This architecture scales extremely well and is very easy to implement and extend as soon as you understood the basic principle. It like to think of it as a SOA within an application.

By dividing your app into the units remember. The optimum number of long running threads per CPU core is 1.

Share Improve this answer Follow

answered Nov 2, 2008 at 8:59

community wiki
Karl

"By dividing your app into the units remember. The optimum number of long running threads per CPU core is 1." pls eloborate – Tatvamasi Aug 6, 2012 at 4:28

3

I recommend flow-based programming, aka dataflow programming. It *uses* OOP and threads, I feel it like a natural step forward, like OOP was to procedural. Have to say, dataflow programming can't be used for everything, it is not generic.

Wikipedia has good articeles on the topic:

http://en.wikipedia.org/wiki/Dataflow_programming

http://en.wikipedia.org/wiki/Flow-based_programming

Also, it has several advantages, as the incredible flexibile configuration, layering; the programmer (Component programmer) has not to program the business logic, it's done in another stage (putting the processing network together).

Did you know, *make* is a dataflow system? See **make -j**, especially if you have multi-core processor.

Share Improve this answer Follow

answered Jun 25, 2010 at 20:25
ern0
3,162 ● 29 ● 42

0

Writing all the code in a multi-threaded application very... carefully! I don't know any better answer than that. (This involves stuff like jonnii mentioned).

I've heard people argue (and agree with them) that the traditional threading model really won't work going into the future, so we're going to have to develop a different set of paradigms / languages to really use these newfangled multi-cores effectively. Languages like Haskell, whose programs are easily parallelizable since any function that has side effects must be explicitly marked that way, and Erlang, which I unfortunately don't know that much about.

0

I suggest the actor model.

Share Improve this answer Follow

answered Nov 2, 2008 at 7:22

Signal9
93 ● 3

0

The actor model is what you are using and it is by far the simplest (and efficient way) for multithreading stuff. Basically each thread has a (synchronized) queue (it can be OS dependent or not) and other threads generate messages and put them in the queue of the thread that will handle the message.

Basic example:

```
thread1_proc() {

msg = get_queue1_msg(); // block until message is
put to queue1
threat1_msg(msg);

}

thread2_proc() {
msg = create_msg_for_thread1();
send_to_queue1(msg);
}
```

It is a tipical example of producer consumer problem.

Share Improve this answer

Follow

answered Nov 2, 2008 at 8:53

INS

10.8k ● 8 ● 61 ● 93

0

It is clearly a difficult problem. Apart from the obvious need for carefulness, I believe that the very first step is to define precisely what threads you need and why.

Design threads as you would design classes : making sure you know what makes them consistent : their contents and their interactions with other threads.

Share Improve this answer

Follow

answered Nov 2, 2008 at 9:04

Benoît

17k ● 8 ● 48 ● 67

0

I recall being somewhat shocked to discover that Java's synchronizedList class wasn't fully thread-safe, but only conditionally thread-safe. I could still get burned if I didn't wrap my accesses (iterators, setters, etc.) in a synchronized block. This means that I might've assured my team and my management that my code was thread safe, but I might've been wrong. Another way I can assure thread safety is for a tool to analyse the code and have it pass. STP, Actor model, Erlang, etc are some ways of getting the latter form of assurance. Being able to assure properties of a program reliably is/will be a huge step forward in programming.

Share Improve this answer

Follow

answered Nov 3, 2008 at 1:56

ja.
4,249 ● 22 ● 22

0

Looks like your IOC is somewhat FBP-like :-) It would be fantastic if the JavaFBP code could get a thorough vetting from someone like yourself versed in the art of writing thread-safe code... It's on SVN in SourceForge.

Share Improve this answer

Follow

answered Jan 11, 2009 at 19:53

Paul Morrison
1,724 ● 3 ● 21 ● 35

@Paul: "Well versed"... only if multiple bite-marks on my hiney qualifies as well versed. But seriously, the main problem will be finding band-width. I definitely plan to review the code, but when, how closely and how critically, I am not sure. – L. Cornelius Dol Jan 15, 2009 at 8:28

0

Some experts feel the answer to your question is to avoid threads altogether, because it's almost impossible to avoid unforseen problems. To quote The Problem with Threads:

We developed a process that included a code maturity rating system (with four levels, red, yellow, green, and blue), design reviews, code reviews, nightly builds, regression tests, and automated code coverage metrics. The portion of the kernel that ensured a consistent view of the program structure was written in early 2000, design reviewed to yellow, and code reviewed to green. The reviewers included concurrency experts, not just inexperienced graduate students (Christopher Hylands (now Brooks), Bart Kienhuis, John Reekie, and [Ed Lee] were all reviewers). We wrote regression tests that achieved 100 percent code coverage... The... system itself began to be widely used, and every use of the system exercised this code. No problems were observed until the code deadlocked on April 26, 2004, four years later.

Share Improve this answer Follow

answered Jun 25, 2010 at 20:52

joe snyder
3,649 ● 2 ● 22 ● 13

0

The safest approach to design new applications with multi threading is to adhere to the rule:

No design below the design.

What does that mean?

Imagine you identified major building blocks of your application. Let it be the GUI, some computations engines. Typically, once you have a large enough team size, some people in the team will ask for "libraries" to "share code" between those major building blocks. While it was relatively easy in the start to define the threading and collaboration rules for the major building blocks, all that effort is now in danger as the "code reuse libraries" will be badly designed, designed when needed and littered with locks and mutexes which "feel right". Those ad-hoc libraries are the design below your design and the major risk for your threading architecture.

What to do about it?

- Tell them that you rather have code duplication than shared code across thread boundaries.
- If you think, the project will really benefit from some libraries, establish the rule that they must be state-free and reentrant.
- Your design is evolving and some of that "common code" could be "moved up" in the design to become a new major building block of your application.
- Stay away from the cool-library-on-the-web-mania. Some third party libraries can really save you a lot of time. But there is also a tendency that anyone has their "favorites", which are hardly essential. And with each third party library you add, your risk of running into threading problems increases.

Last not least, consider to have some message based interaction between your major building blocks; see the often mentioned actor model, for example.

Share Improve this answer Follow

answered Jan 20, 2015 at 13:03

BitTickler

11.8k ● 6 ● 36 ● 58

0

The core concerns as I saw them were (a) avoiding deadlocks and (b) exchanging data between threads. A lessor concern (but only slightly lessor) was avoiding bottlenecks. I had already encountered several problems with disparate out of sequence locking causing deadlocks - it's very well to say "always acquire locks in the same order", but in a medium to large system it is practically speaking often impossible to ensure this.

Caveat: When I came up with this solution I had to target Java 1.1 (so the concurrency package was not yet a twinkle in Doug Lea's eye) - the tools at hand were entirely synchronized and wait/notify. I drew on experience writing a complex multi-process communications system using the real-time message based system QNX.

Based on my experience with QNX which had the deadlock concern, but avoided data-concurrency by coping messages from one process's memory space to anothers, I came up with a message-based approach for objects - which I called IOC, for inter-object coordination. At the inception I envisaged I might create *all* my objects like this, but in hindsight it turns out that they are only necessary at the major control points in a large application - the "interstate interchanges", if you will, not appropriate for every single "intersection" in the road system. That turns out to be a major benefit because they are quite un-POJO.

I envisaged a system where objects would not conceptually invoke synchronized methods, but instead would "send messages". Messages could be send/reply, where the sender waits while the message is processed and returns with the reply, or asynchronous where the message is dropped on a queue and dequeued and processed at a later stage. Note that this is a conceptual distinction - the messaging was implemented using synchronized method calls.

The core objects for the messaging system are an IsolatedObject, an IocBinding and an IocTarget.

The IsolatedObject is so called because it has no public methods; it is this that is extended in order to receive and process messages. Using reflection it is further enforced that child object has no public methods, nor any package or protected methods except those inherited from IsolatedObject nearly all of which are final; it looks very strange at first because when you subclass IsolatedObject, you create an object with 1 protected method:

```
Object processIocMessage(Object msgsdr, int
msgidn, Object msgdta)
```

and all the rest of the methods are private methods to handle specific messages.

The IocTarget is a means of abstracting visibility of an IsolatedObject and is very useful for giving another object a self-reference for sending signals back to you, without exposing your actual object reference.

And the IocBinding simply binds a sender object to a message receiver so that validation checks are not incurred for every message sent, and is created using an IocTarget.

All interaction with the isolated objects is through "sending" it messages - the receiver's processIocMessage method is synchronized which ensures that only one message is be handled at a time.

```
Object iocMessage(int mid, Object dta)
void   iocSignal (int mid, Object dta)
```

Having created a situation where all work done by the isolated object is funneled through a single method, I next arranged the objects in a declared hierarchy by means of a "classification" they declare when constructed - simply a string that identifies them as being one of any number of "types of message receiver", which places the object within some predetermined hierarchy. Then I used the message delivery code to ensure that if the sender was itself an IsolatedObject that for synchronous send/reply messages it was one which is lower on the hierarchy. Asynchronous messages (signals) are dispatched to message receivers using separate threads in a thread pool who's entire job deliver signals, therefore signals can be send from any object to any receiver in the system.

Signals can can deliver any message data desired, but not reply is possible.

Because messages can only be delivered in an *upward* direction (and signals are always upward because they are delivered by a separate thread running solely for that purpose) deadlocks are eliminated by design.

Because interactions between threads are accomplished by exchanging messages using Java synchronization, race conditions and issues of stale data are likewise eliminated by design.

Because any given receiver handles only one message at a time, and because it has no other entry points, all considerations of object state are eliminated - effectively, the object is fully synchronized and synchronization cannot accidentally be left off any method; no getters returning stale cached thread data and no setters changing object state while another method is acting on it.

Because only the interactions between major components is funneled through this mechanism, in practice this has scaled very well - those interactions don't happen nearly as often in practice as I theorized.

The entire design becomes one of an orderly collection of *subsystems* interacting in a tightly controlled manner.

Note this is not used for simpler situations where worker threads using more conventional thread pools will suffice (though I will often inject the worker's results back into the main system by sending an IOC message). Nor is it used for situations where a thread goes off and does something completely independent of the rest of the system such as an HTTP server thread. Lastly, it is not used for situations where there is a resource coordinator that itself does not interact with other objects and where internal synchronization will do the job without risk of deadlock.

EDIT: I should have stated that the messages exchanged should generally be immutable objects; if using mutable objects the act of sending it should be considered a hand over and cause the sender to relinquish all control, and preferably retain no references to the data. Personally, I use a lockable data structure which is locked by the IOC code and therefore becomes immutable on sending (the lock flag is volatile).

Share Improve this answer Follow

edited Jan 20, 2015 at 17:38

answered Nov 2, 2008 at 6:11

L. Cornelius Dol
63.9k ● 26 ● 141 ● 189

Sounds a lot like a limited implementation of the actors concept. – Daniel Spiewak Nov 2, 2008 at 6:27