CSP for Java programmers, Part 1

Pitfalls of multithreaded programming on the Java platform

Level: Intermediate

Abhijit Belapurkar (abhijit_belapurkar@infosys.com), Senior Technical Architect, Infosys Technologies Limited

21 Jun 2005

While the constructs of multithreaded application programming in the Java™ language aren't difficult to learn, many developers struggle with applying them correctly. As a result, multithreaded programs are often far more prone to subtle errors than we would like them to be, leading some developers to avoid them at all costs, even when concurrency and parallelism would clearly yield the most elegant design. In this three-part article, regular developerWorks contributor Abhijit Belapurkar sets you on the path to overcoming your fear of multithreaded programming for good, starting here with an overview of the most common issues involved: race hazards, deadlocks, livelocks, resource starvation, and more.

It is is widely acknowledged that multithreaded programming on the Java platform is a daunting proposition. In fact, the general theory seems to be that multithreaded programming is best left to the Java gurus. Sun Microsystems has indirectly furthered this notion by stating (in the EJB specification -- see Resources) the following as one of the goals of the EJB architecture:

> [a]pplication developers will not have to understand low-level transaction and state management details, multi-threading, connection pooling, or other complex low-level APIs.

Given this as a starting point, it is no surprise that many Java developers shy away from designing and developing multithreaded applications. The fact is, however, that many -- if not most --

enterprise problems are best suited to be resolved by some form of multithreading, and EJB and similar frameworks aren't always the easy answer they're set out to be.

In this three-part article, I introduce you to a theory that honors the complexity of concurrent programming without hiding it from you or making it unnecessarily difficult to learn and apply. Communicating Sequential Processes (CSP) is a precise mathematical theory of concurrency that can be used to build multithreaded applications that are guaranteed to be free of the common problems of concurrency and (perhaps more importantly) *can be proven* to be so.

Before I introduce you to the theory of CSP and its Java language-based implementation, the JCSP library, I would like to be sure that we have a common framework for discussion. I start here with a technical overview of concurrent programming on the Java platform, followed by an in-depth overview of the pitfalls of multithreaded application development; namely race hazards, deadlocks, livelocks, and resource starvation. I wrap up by discussing exactly *why* you cannot verify your multithreaded Java applications, much as you would like to, and confirm the ultimate partiality of the existing workarounds.

With these basics in hand, you should fully appreciate the advantages of JCSP, a conceptual and practical solution to the problems of multithreaded programming on the Java platform, which I discuss in Part 2, as well as the more advanced applications of CSP on the Java platform, which I discuss in Part 3.

Note that all three parts of this article are published simultaneously for your convenience. The article assumes that you are generally familiar with concurrent programming in the Java language, although I do provide a brief overview of the topic here. Refer to the Resources section for more detailed information.

Don't miss the rest of the series!

"CSP for Java programmers" is a three-part introduction to Communicating Sequential Processes (CSP), a paradigm for concurrent programming that honors its complexity without abandoning you to it. Read the other parts of the series:

Part 2: Concurrent

Concurrent programming in the Java language

By itself, concurrent programming is a technique that provides for the execution of operations simultaneously, whether on a single system or spread across a number of systems. Such operations are essentially sequences of instructions, such as the subtasks for a single top-level task, that can be executed in parallel, either as threads or processes. The essential difference between threads and processes is that while processes are typically independent

programming with JCSP

Part 3: Advanced topics in JCSP

(separate address spaces, for example) and can therefore interact only through system-provided interprocess communication mechanisms, threads typically share the state information of a single process and can share objects in memory and system resources directly.

You can achieve concurrency through multiple processes in one of two ways. The first way is to run the processes on the same processor, with the OS handling the context switching between them. (Understandably, this switching is slower than the context switching between multiple threads in the same process.) The second way is to build a massively parallel and complex distributed system by running these multiple processes on different physical processors.

In terms of built-in support, the Java language provides for concurrent programming via threads; each JVM can support many threads of execution at once. You can create a thread in the Java language in one of two ways:

- **By subclassing the `java.lang.Thread` class**. In this case, the overridden `run()` method of the subclass must contain the code that implements the thread's run-time behavior. You execute this code by instantiating the subclassed object and then calling the `start()` method on it, thus internally executing the `run()` method.

- **By creating a custom implementation of the `Runnable` interface**. This interface consists of a single method called `run()`, into which you place your application code. You execute this code by instantiating an object of the implementor class, then passing it in as a constructor parameter when creating a new `Thread`. You can then call the newly created thread object's `start()` method to begin executing the new thread of control.

Back to top

Thread safety and synchronization

A method in a Java object is said to be *thread safe* if it can be safely run in a multithreaded environment. To achieve this safety, there must be a mechanism by which multiple threads running the same method can synchronize their operations, such that only one of them is allowed to proceed when accessing the same object or lines of code. This synchronization requires the threads to communicate with each other using objects called *semaphores*.

One specific type of semaphore is called a *mutual exclusion semaphore* or a *mutex*. As the name indicates, ownership of this semaphore object is mutually exclusive, in that only one thread can own the mutex at any given time. Any other thread that tries to acquire ownership will be blocked

and must wait until the owning thread releases the mutex. If multiple threads are waiting in line for the same mutex, only one of them will get it when it is released by the current owner; the others will continue to block.

In the early 1970s, C.A.R. Hoare and others developed a concept known as a *monitor* (see Resources). A *monitor* is a body of code whose access is guarded by a mutex. Any thread wishing to execute this code must acquire the associated mutex at the top of the code block and release it at the bottom. Because only one thread can own a mutex at a given time, this effectively ensures that only the owing thread can execute a monitor block of code. (The guarded code need not be contiguous -- for example, every object in the Java language has a single monitor associated with it.)

Any developer with exposure to thread programming in the Java language will immediately recognize the above as the net effect of what the `synchronized` keyword does. Java code enclosed within a `synchronized` block is guaranteed to be run by a single thread at any given time. Internally, the `synchronized` keyword is translated by the run time into a situation wherein all contending threads are trying to acquire the (single) mutex associated with the object instance on which they (the threads) are operating. The thread that succeeds in acquiring the mutex runs the code and releases the mutex when exiting the `synchronized` block.

Waiting and notification

The construct of `wait`/`notify` also plays an important role in the Java language's interthread communication mechanism. The essential idea is that one thread needs a certain condition that can be brought about by another thread to become true. It therefore *waits* for the condition to be met. Once the condition is true, the causing thread *notifies* the waiting thread to wake up and proceed from where it left off.

The `wait`/`notify` mechanism is much more difficult to understand and reason about than the `synchronized` mechanism. To reason about the behavioral logic of one method that uses `wait`/`notify` requires that you reason about the logic of all the methods using it. Reasoning about one method at a time, in isolation from others, is a sure means to arriving at incorrect conclusions about the overall system behavior. Clearly, the complexity of doing this increases very rapidly as the number of methods to be reasoned about increases.

Back to top

Thread states

I previously mentioned that the `start()` method of a newly created thread must be called to start its execution. However, simply calling the `start()` method need not imply the thread starts running

immediately. This method just changes the state of the thread from *new* to *runnable*. The thread state becomes *running* (from *runnable*) only when it is actually scheduled for execution by the OS.

Typical OSes support two threading models -- cooperative and preemptive. In the *cooperative* model, each thread has the final say on how long it will retain control of the CPU and when it will give it up. In this model, because a rogue thread may never relinquish control, the other threads may never get to run. In the *preemptive* model, the OS itself uses a timer on the clock "ticks" on which it can abruptly transfer control from one thread to another. In this case, the scheduling policy that decides which thread will gain control next may be based on a variety of criteria, such as relative priorities, how long a particular thread has been waiting to execute, etc.

A thread in the *running* state can enter the *blocked* state if it decides to sleep for some reason, needs to wait for a resource (for example, for input data to arrive on a device, or for notification that some condition has been set), or is blocked while trying to acquire a mutex. A blocked thread reenters the *runnable* state when either the sleep period expires, the expected input arrives, or the current owner of the mutex has released it and notified the waiting threads that the mutex is up for grabs again.

A thread terminates when its `run()` method completes, either by returning normally or by throwing an unchecked exception such as `RuntimeException`. At this time, the state of the thread is *dead*. Once a thread is dead, it cannot be restarted by reinvoking its `start()` method, as doing so will throw the `InvalidThreadStateException`.

Back to top

Four common pitfalls

As I've shown, multithreaded programming in the Java language is facilitated by a number of well-designed constructs supported by the language. In addition, a large collection of design patterns and guidelines have been devised to help you steer clear of the many pitfalls of this complex undertaking. In spite of this, it is very easy to inadvertently introduce a subtle bug into your multithreaded code and, more importantly, such problems are just as difficult to analyze and debug. What follows is a list of the most common problems you'll encounter (and likely have encountered) while attempting multithreaded programming in the Java language.

Race conditions

A *race condition* is said to exist in a system when there is contention for a shared resource between multiple threads and the winner determines the behavior of the system. Allen Holub provides a very simple example of a multithreaded program with this bug in his article "Programming Java threads in the real world" (see Resources). An even more insidious consequence of incorrect

synchronization between conflicting access requests is *data corruption*, wherein the shared data structure is partly updated by one thread and partly by another. In this case, instead of the system behaving per the winning thread's intent, it behaves according to neither's intent, so that both threads end up losing.

Deadlocks

A *deadlock* is a condition where a thread is blocked forever because it is waiting for a certain condition to become true (such as a resource being available), but the condition is prevented from becoming true because the thread that would make it true is, in turn, waiting for the first thread to "do something." In this way, both threads are waiting for the other to take the first step and neither is able to do anything. Read Allen Holub's article (see Resources) for examples of how deadlocks can happen in multithreaded Java code.

Livelocks

A *livelock*, unlike a *deadlock*, happens when threads are actually running, but no work gets done. This usually happens when the two threads are working at cross-purposes, so what is done by the first thread is undone by another. A simple example is where each thread already holds one object and needs another that is held by the other thread. Now imagine a situation wherein each thread puts down the object it possesses and picks up the object put down by the other thread. Clearly, these two threads can run forever in lock-step, effectively managing to achieve nothing at all. (A common real world example is when two people approach each other in a narrow corridor. Each person tries to be polite by moving to one side to let the other one pass, but both keep moving to the same side at the same time, thereby ensuring that neither can pass. This continues for some time, with both of them swaying from side to side and no progress being made.)

Resource starvation

Resource starvation, also known as *thread starvation*, is a consequence of the fact that the `wait`/`notify` primitives of the Java language do not guarantee *live-ness*. It is mandatory that these methods hold locks for the objects that they are waiting or notifying. The `wait()` method called on a particular thread releases the monitor lock prior to commencing to wait, and it must be reacquired before returning from the method, post notification. Accordingly, the Java Language Specification (see Resources) describes a *wait set* associated with each object, in addition to the lock itself. Once a thread releases the lock on an object (following the call to `wait`), it is placed in this wait set.

Most JVM implementations place waiting threads in a queue. Therefore, if there are other threads waiting for the monitor when a notification happens, a new thread will be placed at the back of the queue and won't be the next one to acquire the lock. So, by the time the notified thread actually gets the monitor, the condition for which it was notified may no longer be true and it will have to `wait` again. This can continue indefinitely, thereby leading to wasted computational effort (because

of the shunting of threads in and out of the wait sets) and thread starvation.

The fable of the greedy philosopher

The archetypal example to demonstrate this behavior is the "Wot, no chickens?" problem described by Professor Peter Welch (see Resources). In this scenario, the system under consideration is a college consisting of five philosophers, a chef, and a canteen. All the philosophers except one think for a while (three seconds, in the code sample) and then go to the canteen for food. The "greedy" philosopher doesn't want to waste any time on thinking -- instead he returns to the canteen again and again in the hopes of getting a chicken to eat.

The chef cooks chickens in batches of four, replenishing the canteen when each batch is ready. In spite of going to the kitchen continuously, the greedy philosopher always misses out on the food! This is what happens: the first time he gets there, it is too early and the chef isn't done cooking yet. This results in the greedy philosopher being put on hold (by the `wait()` method call). When released (via a `notify()` method call), he is put back on the canteen queue again. However, while he was on hold, his other four colleagues had arrived, so his position in the canteen queue is behind all of them. The colleagues take the whole batch of four just arrived from the kitchen and the greedy philosopher gets put on hold again. Sadly (or perhaps justly), he never gets out of this cycle.

Back to top

The problem of verification

In general, it is very difficult to verify a multithread program written in Java code against formal specifications. Nor can an automated tool be easily developed for complete and foolproof analysis of common concurrency problems such as deadlock, livelock, and resource starvation -- particularly not in arbitrary Java programs and in the absence of a formal model of concurrency.

Worse yet is the fact that concurrency issues are notoriously fickle and hard to track down. Every Java developer has heard about (or written himself) a Java program that undergoes rigorous analysis and runs correctly for extended periods of time without a latent deadlock manifesting itself. Then one day, suddenly, the problem decides to kick in, costing the development team many a sleepless night trying to get to and fix the root cause.

On the one hand, multithreaded Java programs are prone to errors that are far from obvious and can occur at seemingly arbitrary times. On the other hand, it is entirely possible that none of these bugs will ever manifest in your programs. The problem lies in the not knowing. The complex nature of multithreaded programs makes them difficult to effectively verify. There is no cut-and-dried set of rules for ferreting out such problems in your multithreaded code or conclusively proving their absence, which leads some Java developers to shy away completely from designing and developing

multithreaded applications, even where it would make perfect sense to model the system in terms of concurrency and parallelism.

Developers who *do* attempt multithreaded programming usually settle for one or both of the following, at best partial, solutions:

- Test the code long and hard, sort out all the concurrency issues that do crop up, and fervently hope that all such problems have been discovered and fixed by the time the application goes live.
- Make ample use of the design patterns and guidelines established for multithreaded programming. Such guidelines work only if the entire system is designed to their specification, however, and no design rule can cover all types of systems anyway.

While less known, there is a third option to the problem of writing (and then verifying) correct multithreaded applications. Issues like deadlock and livelock are best handled at design time, using the precise mathematical theory of thread synchronization known as Communicating Sequential Processes (CSP). Developed by C.A.R. Hoare in the late 1970s, CSP offers an effective means to prove that a system built using its constructs and tools is free of of the common problems of concurrency.

Back to top

Conclusion to Part 1

In this first part of a comprehensive look at CSP for Java programmers, I focused on the first step to overcoming the common issues in multithreaded application development, which is understanding them. I walked you through the currently supported constructs for multithreaded programming on the Java platform, explained their origins, and discussed the problems that such programs can have. I also explained the difficulty of applying a formal theory to either weed out or prove the absence of these problems (namely race hazards, deadlocks, livelocks, and resource starvation) in arbitrary, large, and complex applications.

In Part 2, with this basic framework in mind, I introduce you to CSP and its Java-based implementation, the JCSP library. As you'll discover, CSP is a complex mathematical theory with numerous powerful applications (I discuss some of the more advanced ones in Part 3), including the resolution of common issues in multithreaded programming.

To learn how the JCSP library distills the essence of CSP into a well-understood framework of Java constructs, jump to "Part 2: Concurrent programming with JCSP" now.

Acknowledgments

I would like to gratefully acknowledge the kind encouragement I received from Professor Peter Welch during the writing of this article series. His busy schedule notwithstanding, he took time to do a very thorough review of a draft version and gave many valuable inputs towards enhancing the quality and accuracy of the series. All remaining errors are mine alone! The examples I have worked with in my articles are based on and/or derived from those documented in the Javadocs for the JCSP library and/or the Powerpoint presentation slides available on the JCSP Web site. Both of these sources offer a wealth of information to be explored.

Back to top

Resources

- Brian Goetz's three-part "Threading lightly" is a smart and methodical approach to resolving synchronization issues on the Java platform (developerWorks, July 2001).
- Brian's *Java theory and practice* column regularly covers the multithreaded programming. "Concurrency made simple (sort of)" (developerWorks, November 2002) is a first look at `util.concurrent`, a widely used open-source package of concurrency utilities for the Java platform.
- Allen Holub has written a great deal about multithreaded programming on the Java platform. His "Programming Java threads in the real world" (*JavaWorld*, October 1998) provides examples of threading problems such as race conditions and deadlocks.
- Also read Holub's famous diatribe, "If I were king: A proposal for fixing the Java programming language's threading problems" (developerWorks, October 2000).
- Professor Peter Welch of the University of Kent at Canterbury, UK, devised the "Wot, no chickens?" problem to illustrate resource starvation in Java programs.
- C.A.R. Hoare's seminal paper "Communicating Sequential Processes" introduced the parallel composition of communicating sequential processes as a fundamental program structuring method (*Communications of the ACM Archive*, 1978).
- Hoare's "Monitors: an operating system structuring concept" (*Communications of the ACM Archive*,1974) first introduced the concept of monitors to the world, via illustrative examples including a single resource scheduler, a bounded buffer, an alarm clock, a buffer pool, a disk head optimizer, and a version of the problem of readers and writers.
- Enterprise JavaBeans is one of several frameworks that attempt to hide the complexities of

multithreaded programming from Java programmers.

- Read the Java Language Specification.
- You'll find articles about every aspect of Java programming in the developerWorks Java technology zone.
- Also see the Java technology zone tutorials page for a complete listing of free Java-focused tutorials from developerWorks.

Back to top

About the author

Abhijit Belapurkar has a B.Tech. degree in computer science from the Indian Institute of Technology (IIT), Delhi, India. He has been working in the areas of architectures and information security for distributed applications for the last 11 years and using the Java platform to build n-tier applications for about six years. He is presently working as a senior technical architect in the J2EE space, with Infosys Technologies Limited, Bangalore, India.

Back to top

Rate this page

Please take a moment to complete this form to help us better serve you.

Did the information help you to achieve your goal? ○ Yes ○ No ○ Don't know

Please provide us with comments to help improve this page:

How useful is the information? (1 = Not at all, 5 = Extremely useful)	○ 1	○ 2	○ 3	○ 4	○ 5

Submit

Back to top