Define a multithreading threshold when building SaaS applications

Encounter multithreading challenges when interfacing COBOL and Java in a SaaS app

Judith M. Myerson
Systems Engineer and Architect

While on-premise COBOL programs have been successfully transformed into Java™-based Software as a Service (SaaS) applications, there are multithreading issues developers should watch out for when interfacing COBOL and Java with one another in a SaaS application. The author illustrates what proactive actions to take in a multithreaded SaaS failure scenario.

Back to Cloud Island: All SaaS users happily get quick responses. Since the only control these users have is to access the application, they don't care whether the application has multithreading routines or not; they won't care how many cores in the cloud are used to speed up the processing of the multiple threads in parallel. The application in question was successfully migrated from on-premise, multithreaded COBOL legacy system.

Of course, one day the SaaS application slows down and continues to run slower and slower; the users howl. They discover too late that:

- The cores, except one, are malfunctioning.
- SaaS subscription is limited to two cores, not the maximum of four cores.
- The SaaS application has been recently updated with multithreading flaws.
- The failover plan has failed.

Meanwhile, on the functional side of the island, a team of developers gather to consider a range of options to get the system up and running again.

This article helps you explore multithreading performance issues, it describes my first encounters with multithreading (briefly; I'll try to cause as little pain as possible). Next it examines the multithreading controls model cloud users have access to and creates a brief on built-in multithreading support in Java and COBOL. The goal of this article is to show you what proactive actions the provider can take to halt the damage, fix the problems, restore the system, and notify clients.
Multithreading performance

The more cores there are in the CPUs, potentially, the better the multiple threads perform in executing instructions from program code, but the number of cores that have been used is NOT the only factor in determining how well multithreaded software performs. Another factor: It is not always possible to accomplish a complete thread parallelization in an algorithm for a task or process. Computation of some threads that have been parallelized in the previous steps may implicitly give a sequential result.

Consider an application split into modules, each one designed to accomplish a task or process. In one module, you have six single threads working in nearly parallel. Let's assume:

a is thread 1, b is thread 2, c is thread 3, d is thread 4, e is thread 5 and f is thread 6

Where each of the results r0, r1, and r2 is computed as a combined parallel thread of two single threads as follows:

\[
\begin{align*}
\text{r0} &= \text{a + b} \\
\text{r1} &= \text{c + d} \\
\text{r2} &= \text{e + f}
\end{align*}
\]

However, adding all three parallel thread results (r0, r1, and r2) gives sequential result g — not a new parallel thread, like this:

\[
\text{g} = \text{r0 + r1 + r2}
\]

All sequential parts have to wait for the threads that have been parallelized in the previous step to signal that they're ready. The more sequential parts there are in a program, the less benefit it will have from multiple cores.

Other factors impacting multithreading algorithms include:

- Limitations on COBOL's THREAD compiler option.
- Limitations on Java's multithreading routine.
- Flaws in decomposing tightly coupled COBOL programs into loosely coupled SaaS applications.
- Cloud user controls.
- Lack of multithreading thresholds.

My first encounters with threads

Decades ago, when I first started to use mainframe COBOL, I considered interfacing it with non-COBOL languages. I discussed this with a professor as a possible dissertation topic on performance impacts of COBOL interfaces. I shared my thoughts on parallelizing threads in program code for a subroutine.

To find out what the performance impacts might be, I experimented with mini-COBOL/Fortran interfaces based on "A Fortran Interface to the CODASYL database task group specifications" (see
Resources). Fortran was a popular language in those days. Then COBOL did not have the THREAD option like we do today. The maximum size of processors was very small compared to the enormous size of processors you see today.

In my experiment, I noted that some COBOL data types did not have Fortran equivalents. When data or objects that the application no longer needed stayed on the disk, I got around the memory limitations by calling subprograms when needed and releasing them when not needed and by automatically deleting data no longer needed.

Each subprogram performed one or more tasks. A couple of threads in some tasks were computed as a parallel thread (r0 = thread a + thread b). All sequential parts waited for the threads that were parallelized in the previous steps to signal that they’re ready. The wait was short.

If we had clouds then, I would have developed a SaaS application with multithreading routines on the Platform as a Service (PaaS) running on multi-core virtual machines.

Model cloud users on multithreading control

The purpose of a multithreading threshold is to set the limits on the number of threads a task can perform in parallel. When the threshold is reached, the threads that have finished their work can take work from the queue of other threads.

The model users' control over a multithreading threshold depends on how much control they can get from a cloud provider when they:

• Access SaaS on demand.
• Build applications with the PaaS.
• Work with Infrastructure as a Service (IaaS) virtual machines.

Let's look at each in more detail.

Access SaaS on demand

The SaaS user has the least control while the provider has the most control.

• **End user control:** The only control end users have is to access the SaaS application from a mobile device or a virtual desktop whether they are private individuals, businesses (small or medium), or government agencies. Examples of SaaS applications include ship arrival and departure schedules, customer relationship management, human resources, and spreadsheets.
• **SaaS provider control:** At a minimum, the provider manages access controls by limiting the number of authorized users who can concurrently access multithreaded application as set forth in the user threshold policy. The provider controls the cores, operating systems, servers, and network infrastructure needed to run the SaaS application.
• **Multithread threshold control:** The SaaS end user does not have control over a multithreading threshold. The provider may limit the multithread threshold to the maximum number of cores the application can use to process threads in parallel. The maximum number the provider sets depends on the subscription rates the SaaS end users choose.
Build applications with PaaS

The PaaS developer has more control while the provider has less control.

- **Developer control:** The developer controls and protects all applications found in a full business life cycle created with the PaaS. The developer sets the multithreading threshold to the number of cores an application can use to process threads in parallel when building an application. The developer may set the user and virtual desktop threshold levels.
- **PaaS provider control:** At a minimum, the provider controls the cores, operating systems, servers, and network infrastructure needed to run SaaS applications, develop new applications, or test run the multithreading and scalability of the applications in the cloud. The provider sets resource, data requests, social media, and load balancing threshold levels.
- **Multithreading threshold control:** The developer sets the multithreading threshold depending on the complexity of thread algorithms in the application. The provider limits the threshold the developer sets to the maximum number of cores the developer may use in executing the threads.

Work with IaaS virtual machines

The IaaS infrastructure or network specialist has the most control.

- **Network specialist control:** The infrastructure or network specialist controls the cores, operating systems, network equipment, and deployed multithreaded applications at the virtual machine level. The infrastructure specialist can scale up or down virtual servers or blocks of storage area and the specialist uses social media tools to communicate with other IaaS specialists, PaaS developers, and the provider. The infrastructure specialist may set the user, load balancing, and virtual desktop threshold levels.
- **IaaS provider control:** At a minimum, the provider controls the infrastructure of traditional computing resources underlying virtual machines, the maximum number of cores allocated to the infrastructure or network specialist, and what applications are needed to access the IaaS. The provider sets the user, resource, data requests, social media, and load balancing threshold levels.
- **Multithreading threshold control:** The infrastructure specialist may set the multithreading threshold. The provider may negotiate with the infrastructure specialist on the maximum number of cores for each virtual server.

The Java programming language has built-in multithreading support. Let's look at that.

Java's built-in multithreading support

Today, as an interface with COBOL, Java is the most popular language, followed by C/C++. Fortran, currently a favorite with scientists, has slid down the ladder of popularity in the general developer population.

Java supports multithreading at the language level; much of this support focuses on coordinating access to data shared among multiple threads.
The Java virtual machine (JVM) organizes the data of a running Java application into several runtime data areas:

- One or more Java stacks
- A heap
- A method area

Inside the JVM, each thread is given a Java stack that contains data no other thread can access. The data includes the local variables, parameters, and return values of each method the thread has invoked.

All threads share one heap. The method area contains all the class (or static) variables used by the program. Unlike the stack, however, the class variables in the method area are shared by all threads.

I/O and GUI programming is one example in which multithreading is required to provide a seamless experience for the user. When writing multithreaded programs, you must ensure that no one thread stops the work of any other thread.

The JVM can manage thread movement from a ready queue onto the multi-core processor; it's there that the thread can begin executing its program code. This is accomplished through both cooperative and preemptive models:

- Cooperative threading allows the threads to decide when they should give up the processor to other waiting threads. All running threads share execution time.
- Preemptive threading allows the OS to interrupt threads at any time, after a period of time. All running threads shared resources.

To create a thread using the Java language, you instantiate an object of type `Thread` (or a subclass) and send it the `start()` command. A thread will continue running until `run()` returns to the main program. At this point, the thread dies.

Most applications require threads to communicate and synchronize their behaviors to one another. At the very least to accomplish this task in a Java program, you use locks. To prevent multiple accesses to the same thread, threads can acquire and release a lock before using resources.

Threads that attempt to acquire a lock in use go to sleep until the thread holding the lock releases it. After the lock is freed, the sleeping thread moves to the ready-to-run queue.

The use of locks can bring multithreading problems with it, including *deadlocking* (two or more competing actions wait for the other to finish so neither ever finishes. In computer science, a deadlock that involves exactly two competing actions is called a *deadly embrace*). Work cannot be completed because different threads are waiting for locks that will never be released.

Local references cannot be passed from one thread to another. They are valid only in the thread in which you create them. If you pass them, you will not be able to free the local references when no
longer needed. You should always convert local references to global references whenever there is a possibility that multiple threads might use the same reference.

Another problem is that the calling Java program with well-designed multithreading routines will fail if the called COBOL program encounters problems with the THREAD compiler option during execution.

**COBOL’S THREAD compiler option**

Another story from the Dark Ages: When I first used COBOL decades ago, the THREAD compiler option did not exist. Today, running COBOL programs in multiple threads requires that you compile all COBOL programs with the THREAD compiler option. The same program can have separate threads — for example, one thread for a task of the program, a second thread for the second task of the same program, and so on.

When you write COBOL programs with this compiler option, choose appropriate linkage statements and language elements (including statements, special registers, and clauses) that make up the logic of a COBOL program (see Resources). Watch out for the language elements that do not work with the compiler option. If you compile the program containing unwanted elements with the compiler option, they are flagged. Be mindful that some COBOL applications depend on subsystems or other applications that may contain unwanted language elements.

Another way of handling limitations on multithreading is to work within the scopes of (acceptable) language elements. Because your COBOL programs can run as separate threads within a process, a language element can be interpreted in two different scopes: Run-unit scope or program invocation instance scope. These two types of scope are important in determining where an item can be referenced and how long the item persists in storage.

In a multithreaded environment, a COBOL run unit is the portion of the process that includes threads that have been actively executing COBOL programs. While the COBOL run unit runs, the language element persists and is available to other programs (Java) within this thread. It continues until no COBOL program is active in the execution stack for any of the threads.

Within a thread, control is transferred between separate COBOL and non-COBOL programs. For example, a COBOL program can call another COBOL program or a Java program. Each separately called program is a program invocation instance. Those instances of a particular program can exist in multiple threads within a given process. The language element persists only within a particular program invocation.

One problem is that the calling COBOL program with the THREAD option will fail if the called Java program encounters problems with its multithreading routines during execution.

**Multithreaded SaaS failure scenario**

A company successfully built on premise collaboration applications with no multithreading (deadlocking) issues and then migrated them to an external SaaS/provider hosting data center.

Define a multithreading threshold when building SaaS applications
regions in the United States. For a few months the application ran smoothly until a network glitch at a data center brought down the system for a couple of days.

An upsurge in complaints rolled in (as you would expect).

The provider groaned since the company was unable to get the SaaS collaboration applications back in service within two minutes after the failure, as guaranteed in a SLA negotiated with the SaaS users. All application threads were in deadlocked mode.

It took a couple of days for the provider to get the SaaS applications working. The provider was not able to provide effective failover that would unlock thread deadlocking.

If the provider had taken the following proactive actions, the provider may be able to halt the damage, fix the problems, restore the system, and notify clients.

Halt the damage
To halt damage, the provider should plan ahead by preparing SaaS applications as instances for automated failover. The SLA as negotiated between the SaaS subscriber and the provider should be in place.

Meanwhile SaaS clients are notified that the service is continued at another data center while the provider is fixing the problems.

Just remember, keeping customers in the dark about a legitimate situation will always work to your disadvantage.

Fix the problem
The provider can plan ahead for the failure:

• Test whether the SaaS application is free of deadlocking and other multithreading problems.
• Set a multithreading threshold for each language interface (COBOL, Java).
• Install instances of the multithreaded SaaS application to allow failover from one data center to another.
• Check periodically if backup tapes are working properly and free of defects.

Restore the system
The next step is to restore:

• The system at the data center where the system was brought down.
• The multithreaded SaaS application to the restored system.

After this, test the restored system's resilience to ensure relatively smooth failover to another data center. Back up a copy of the restored system before moving it to a production environment.

Notify clients
As soon as the SaaS applications are restored in the data center that was brought down, the provider notifies its clients of the following:
• The restoration was complete.
• The SaaS application was moved to the production environment.
• The terms of the SLA (free credit, reimbursements, an opportunity to terminate) as negotiated with the SaaS subscribers are enforced.

Conclusion

In planning for building a SaaS application, consider best practices for resolving multithreading issues. Multithread thresholds should be considered for use in resolving multithreading issues. You need to build a team of developers, managers, business analysts, system engineers and make it easier for them to their job of resolving multithreading issues when building SaaS applications.
Resources

Learn

- For more information pertaining to the topic in this article:
  - Read the author’s article, "Get proactive with limitations in COBOL multithreading" to get more information on handling threads.
  - Read the author's article, "Got performance impacts of interfacing COBOL with other languages? Reduce them!" on interfacing with Java, Fortran and C/C++.
  - Learn how to build MicroFocus COBOL applications on AIX in IBM DB2 Universal Database Application Development Guide: Building and Running Applications.
  - The Work with Web services in enterprise-wide SOA series by Judith M. Myerson offers information on how to work with Web services in enterprise-wide SOAs.
  - Browse the Judith M. Myerson's series, Use SLAs in a Web services context, has details on service-level agreement.
  - Bring your organization into the future with RFID in the Supply Chain, which explains business processes, operational and implementation problems, risks, vulnerabilities, and security and privacy. Find out how RFID is to interact real objects to virtual ones.
  - Read Judith M. Myerson's The Complete Book of Middleware, which focuses on the essential principles and priorities of system design and emphasizes the new requirements brought forward by the rise of e-commerce and distributed integrated systems.
  - Get the business insight and the technical know-how to ensure successful systems integration by reading Enterprise Systems Integration, Second Edition.
  - Learn more about cloud computing technologies at cloud at developerWorks.
  - Follow developerWorks on Twitter.
  - Watch developerWorks demos ranging from product installation and setup demos for beginners, to advanced functionality for experienced developers.

Get products and technologies

- Access IBM SmartCloud Enterprise.
- Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement Service Oriented Architecture efficiently.

Discuss

- Get involved in the developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.
About the author

Judith M. Myerson

Judith M. Myerson is a systems architect and engineer. Her areas of interest include enterprise-wide systems, middleware technologies, database technologies, cloud computing, threshold policies, industries, network management, security, RFID technologies, presentation management, and project management.

© Copyright IBM Corporation 2013
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)