SHS Web of Conferences **25**, 02003 (2016) DOI: 10.1051/shsconf/20162502003
ICITCE 2015

The discuss of the dead-lock issue in event-driven system

Ling Tang*
Department of Information Science and Technology, East China University of Political Science and Law, Shanghai, China

ABSTRACT: This paper analyzes the dead-lock issue triggered by unseemliness usage of synchronization mechanism in event-driven system. The author brings out some important ways to solve the dead-lock issue. These methods can offer principles to the system developers to avoid dead-lock risk during system design and implementation.

Keywords: event-driver; dead-lock; event-queue; threads pool

1 INTRODUCTION

In the view of the construction of system, Service-Oriented Architecture (SOA) [1] divides the whole system into a series of independent service. These services can be easily re-used to construct various cross-platform applications. But because SOA mainly focuses on the static information (the services' composition of a system), it does not match with dynamic system transactions.

In order to resolve the above-mentioned problem, the Event-Driven mechanism is introduced. It enables the system to perceive and quickly (and asynchronously) respond the events in transactions. Event-Driven Architecture (EDA) [2] is a widely used software architecture pattern. In this architecture, an event is triggered when an operation needs performing, and then the executer of the operation finishes the operation by receiving and processing the event. This architecture adapts to the design and implementation of a loosely-coupled system very much. Different system component communicates with each other only by sending event. It is obviously that there is a natural affinity between EDA and SOA. By introducing EDA into SOA, Event-Driven SOA [3] architecture is derived. An event can trigger to call one or several services, and services can also generate some events to call themselves or other services. The typical Event-Driver SOA based software systems includes Financial Systems [4], Database Transaction Process [5], etc..

*Corresponding author: ausflug163@163.com

In any system based on Event-Driven SOA, events and their processing are the essential elements. Generally, events are triggered because of some requests; the processing of the events is performed by services. The processing ability of a service has upper limit, so it can only handle more or less limited events at the same time. Those events which haven't been handled in time are buffered in an explicit or implicit Event-Queue. Besides, the event processing provided by service has to be performed in an executing context. Normally, such context is provided by threads (or processes) in the major operation systems. Nowadays, the events are processed by the threads one by one circularly. Hence, beyond any concrete system implementation, an Event-Driven SOA based system can be abstract as Figure 1.

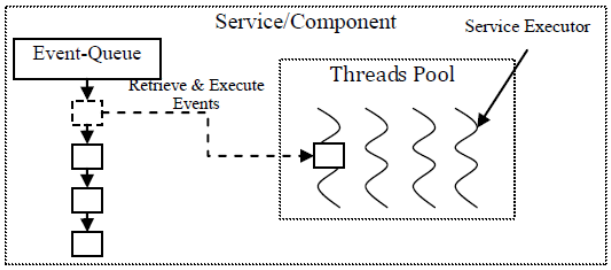

Figure 1. Event-Driven Architecture Abstraction

A group of threads (threads pool) process various events in the Event-Queue. Each thread obtains one event from the Event-Queue and then processes it. After finishing the thread, it continues to obtain an-

other event from the queue and processes it. The process keeps on circulating until all events in the queue are completed. Then the threads turn into waiting status until another new event comes into the queue.

When multiple tasks are executed concurrently in the system, it is unavoidable to use various synchronization mechanisms to protect the concurrent operations during the event processing. However, when designing and implementing an Event-Driven SOA based system; the synchronization mechanisms must be used very carefully; inappropriate usage will cause dead-lock. Based on the analysis of the dead-lock problem by using synchronization mechanisms in Event-Drive SOA architecture, this paper describes several design and implementation techniques to resolve the dead-lock problem in detail.

2 THE DEAD-LOCK PROBLEM IN EVENT-DRIVEN SOA ARCHITECTURE

Taking mutex for example, assuming some transaction includes handling two events: Event1 and Event2, a mutex M is held when handling Event1, and then M is released when handling Event2. Furthermore, assuming there are three same transactions A, B and C are executed concurrently. There are two threads in the threads pool: T1 and T2. This is shown in Figure 2.

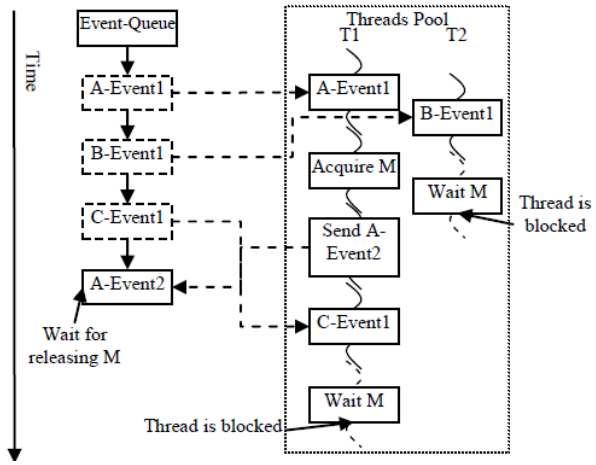

Figure 2. Dead-Lock Problem Schematic

First, when A, B and C are started, all of them send Event1 to Event-Queue. Then T1 obtains A-Event1 to process, and T2 obtains B-Event1 to process. When T1 handles A-Event1, it holds M successfully. Then when T2 handles B-Event1, it can't hold M, so it must wait there.

When T1 finishes processing A-Event1, it sends A-Event2 for triggering the next step of the transaction. And then T1 continues to choose the next event named C-Event1 from the Event-Queue and process it. But because M hasn't been released, T1 can't hold M either when it handles C-Event1. So T1 can do nothing but only wait there. At this time, all threads in the threads pool are waiting for M, but the event A-Event2 which releases M can't be processed by any threads. Accordingly, the system falls into dead-lock state.

Such a dead-lock problem can be similarly promoted to the systems of larger scale: if there are N threads in the threads pool, when the concurrency for acquiring some synchronization object reaches N+1, the dead-lock problem can be triggered. Generally, the reason of dead-lock is because of the confliction of the acquired resources. In the Event-Driven SOA architecture, the thread resource needs acquiring at first before handling events. For the above-mentioned transactions, the acquiring sequence of the resources is shown as Figure 3.

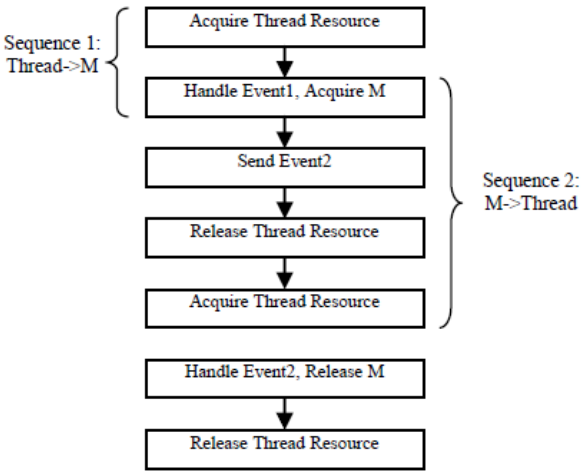

Figure 3. Resource Acquiring Sequence Schematic

It can be found that there are two opposite resource acquiring sequences: 1. Acquiring M after acquiring thread resource. 2. Acquiring thread resource after acquiring M. Therefore, as an implicit resource, the threads which handle the events conflicts with the synchronization object, and this finally causes dead-lock. There are two points of essence leading to this phenomenon:

1. The operations of acquiring and releasing the synchronization object are performed in the process of two different events. In the period between the two events, all service threads available may be blocked on the synchronization object, thereby the event for releasing the synchronization object can't be scheduled by an available thread.

2. A thread can't schedule the other events once it is blocked on a synchronization object.

Taking an Event-Driven SOA based system in reality for example, assuming a company has a Print department (corresponding to a service component); there are two print-operators called A and B (corresponding to a service thread). Each operator can only handle one thing at one time. Each one's job is print-

ing the incoming request (corresponding events) and sending the result papers to the other departments. There is only one printer in the department. Because the printing process is so slow, they decide to use Event-Driven (asynchronously) method to improve the throughput. When A receives a printing request at first, he applies the exclusive access of the printer, and then starts the printing task. When B receives another printing request, he has to wait for A's completion. But when A's first printing task is on-going, A also receives the third printing request. According to the Event-Driven model, A doesn't wait for finishing the first printing request and then handle the new one. Instead, A handles the third printing request immediately. But the printer hasn't been released at the moment, so A has to wait. Then even A's first printing request is finished, there isn't anyone to send out the result papers and release the printer, because both A and B are in the waiting state.

All in all, when designing and implementing the Event-Driven SOA based systems, it must be very cautious for using various synchronization objects, the confliction with service threads must be considered carefully.

3 SOLUTIONS

In general, causing dead-lock must satisfy four necessary conditions [6]: 1. Mutual exclusive; 2. Hold and wait; 3. Non-preemption; 4. Circular wait. To resolve the dead-lock problem, one of the four conditions must be broken. The method to handle dead-lock can be categorized into three types [6]: 1. Prevent dead-lock; 2. Avoid dead-lock; 3. Detect and relieve dead-lock. Among them, the 3rd kind of method needs to relieve the dead-lock by terminating some attending threads when detecting the dead-lock. The programming logic needs special processing to adapt to being suddenly terminated during execution, this brings huge complexity into the programming design, thus it can't be commonly used in various Event-Driven SOA based scenario. So our following solutions mainly focus on the 1st and 2nd kind of methods.

3.1 *Constrain the releasing point of synchronization object*

This method requires that, if a synchronization object is acquired when handling an event; the object must be released in the same event. Apparently, this method can make sure that the thread resource isn't acquired after acquiring the synchronization object, thus the dead-lock could not happen. This belongs to the method of preventing dead-lock.

However, the restriction of acquiring and releasing a synchronization object in one event is too strict, because a complicate transaction may lead to a lot of processing work after acquiring synchronization objects. These processing cannot be able to be implemented in one event. For example, after acquiring a synchronization object, a transaction may want to do a series of asynchronous I/O, and the synchronization object cannot be released only after the I/O is finished. To meet this requirement, the event processing must wait for the I/O's completion, so the thread keeps being occupied and cannot serve other events. This affects the system concurrency and throughput severely. This method actually constrains the asynchronous feature of Event-Driven SOA architecture. Therefore, it can only apply to the simple systems which don't have much throughput requirement.

3.2 *Bind event handler thread*

This method means that, when a thread handles an event, once a synchronization object is acquired, the thread is bound with the transaction which sends the event. Then all afterward events which sent in this transaction must be handled by this thread, until the event which releases the synchronization object is processed. While a thread is bound with some transaction, it cannot handle the other events which belong to other transactions and needs to acquire synchronization objects. In this way, this method also makes sure it won't appear that thread resource is acquired after acquiring synchronization object. Because the thread has been bound with the transaction, the thread is always available after acquiring the synchronization object.

This method is similar to Ⅲ.A. The basic idea of this method is reserving the thread which holds the synchronization object, to make sure that the synchronization object can be released in this thread. But the difference is that, this method doesn't make constraint to how to use the synchronization objects. The resolution is resolved in the system architecture layer, the actual transaction won't see any special processing (i.e., the logic of how the event is handled need not handling specially). Therefore, this method belongs to the method of avoiding dead-lock.

But due to the period of the thread is bound; it can't handle other transactions' events which need to acquire synchronization objects, so this method also constrains the concurrency and throughput of the systems. But comparing with Ⅲ.A, even after binding in this method, the thread can still handle those events which need not to acquire synchronization objects, so its concurrency and throughput are better than Ⅲ.A .

3.3 *Multiple level event-queues*

The above-mentioned two methods focus on ensuring the events of acquiring and releasing synchronization object can be handled in the same thread. But this requirement is too strict. Actually it is only necessary that the event of releasing synchronization object

could be handled by some threads, it is not a requirement that the thread must be as same as the one which acquires the synchronization object.

In order to achieve this, this method defines dedicated Event-Queue and threads pool for the events which acquire synchronization objects. Still considering the example in section 2, because Event1 needs to hold M, Event-Queue 2 and threads pool 2 are defined dedicatedly for handling the events which needs to acquire M. Assuming there is only one thread T3 in threads pool 2. When transaction A, B and C start, they send Event1 to Event-Queue 2. Then T3 processes A-Event1 at first, it can hold M successfully and send A-Event2. Then T3 processes B-Event1, but since M has been held by A-Event1, T3 has to wait. But because A-Event2 doesn't need to acquire M, A-Event2 isn't handled by threads pool 2, but by threads pool 1 instead. So A-Event2 is sent to Event-Queue 1, the thread in threads pool 1 can handle A-Event2, and then M can be released properly. The procedure is shown in Figure 4.

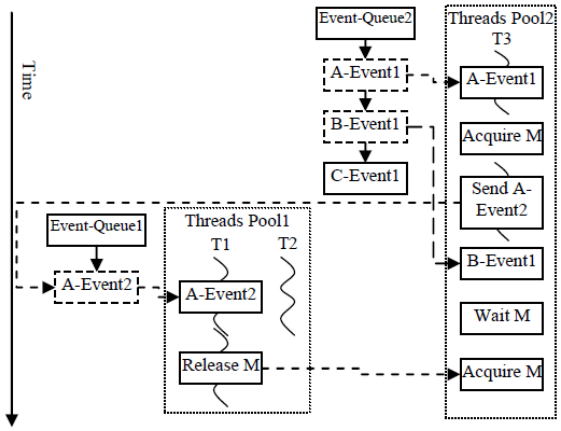

Figure 3. The Principle of Multiple Level Event-Queues

In this way, because all events which acquire M are handled by threads pool 2, the threads in threads pool 1 are never blocked by M; therefore the releasing of event M can always be handled properly.

Ideally, each single synchronization object needs to specify with a corresponding Event-Queue and threads pool. A large scale system may use many synchronization objects, it isn't reasonable to specify Event-Queues and threads pools for every synchronization object. As an optimization, it can be defined according to the categories of the synchronization objects. For example, some transactions acquire the synchronization object M in Event1 and release M in Event2. But some other transactions acquire the synchronization object N in Event1 and release N in Event2. Meanwhile, if there isn't any relationship between M and N, i.e., there isn't any transaction which needs to acquire M and N simultaneously, and then M and N can be considered into the same category, they can deal with the same Event-Queue and threads pool.

Nevertheless, if some transactions acquire N after acquiring N, then M and N should be considered in different levels, they can share the same Event-Queue and threads pool, or lese the dead-lock can be triggered. For example, assuming transaction A and B need to acquire M in Event1, and then acquire N in Event2, and finally release M and N in Event3, as shown in Figure 5.

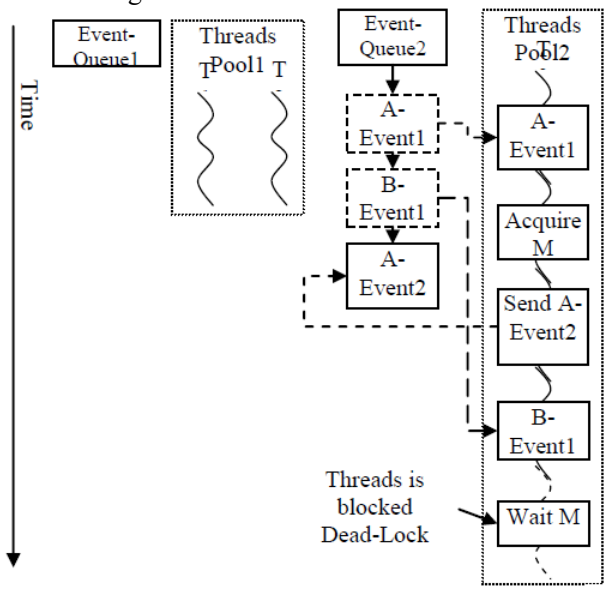

Figure 4. Dead-Lock caused by synchronization object in different level

Here, assuming both Event1 and Event2 are handled in threads pool 2.At first, T3 handles A-Event1 and it holds M. And then T3 sends out A-Event2. But when T3 handles B-Event1 in succession, it is blocked because it can't acquire M. Therefore, A-Event2 can't be handled, furthermore it leads to the fact that A-Event3 can't be sent out for releasing M. So T3 falls into dead-lock scenario.

Thus the categories of synchronization objects are determined by their levels. If a synchronization object belongs to low level in one transaction, but belongs to high level in another transaction, and then the synchronization object should be categorized by the high level. Afterwards, each category is specified with a new Event-Queue and a new threads pool.

As it is shown in

Figure 3, threads pool 2 only corresponds to one mutex M. Only one thread is enough for threads pool 2 in this case, because there is only one event can acquire M at one time. But if the synchronization object belongs to some other types, like RW-Lock or semaphore, they can be acquired by multiple events simultaneously; or if the corresponding category contains multiple synchronization objects, then increasing the thread number of the threads pool can improve the concurrency and throughput of handling those events which acquire the synchronization objects.

Relative to the methods in Ⅲ.A and Ⅲ.B, this method takes more memory resource for defining new

Event-Queues and threads pools. The benefit is that it only affects the concurrency of handling the events which acquire synchronization objects. But the other events are still handled in the original Event-Queue and threads pool, their concurrency and throughput aren't affected by the events which acquire synchronization objects. Because the usage of the synchronization objects isn't constrained in the transactions, this method also belongs to the method of avoiding dead-lock.

3.4 *Improve synchronization mechanism to avoid thread blocking*

According to the description about the essence of this dead-lock problem in section 2, the previous three methods tries to resolve the problem against the 1st point, i.e., making sure the event which releases the synchronization objects can be assigned to an available thread. But if the 2nd point can be resolved, i.e., a thread can process other events even it is blocked (i.e., a thread is never blocked), then the event which releases synchronization objects can always be scheduled by an available thread.

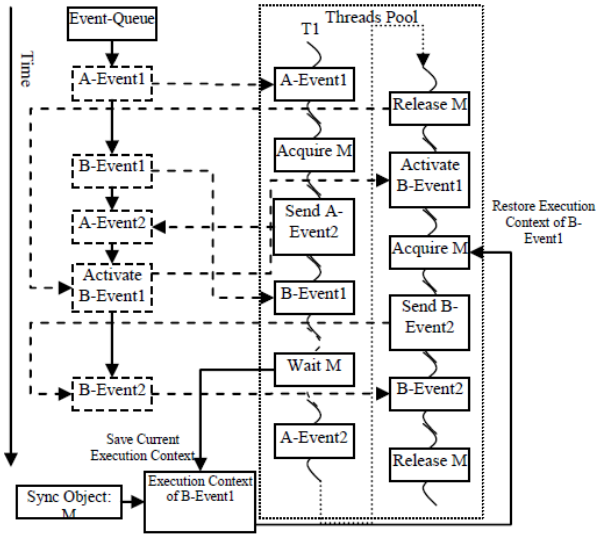

Figure 5. Improve Synchronization Mechanism to Avoid Thread Blocking

In today's popular operating systems, the blocking mechanism of the synchronization objects is the basic system primitive. To make a thread not be blocked, new synchronization objects must be implemented. When a thread needs to blocking on a synchronization object, the current execution context (stacks and registers) of the thread needs to keeping at somewhere else, and then the thread can go to handle other events. And when a synchronization object is released, if there is a waiter on this synchronization object, an event is sent to activate the waiter. The processing for this event obtains the waiter's execution context which is saved when the waiter is blocked, and then the context is loaded into the thread which handles the event, so that the waiter can be restored from the blocked point. For example, assuming transactions A and B acquire mutex M in Event1 and release M in Event2. There is only one service thread in the threads pool. The procedure is shown in Figure 5.

When thread T1 handles B-Event1, because it can't acquire M, it saves its current execution context. Then T1 continues to handle the following events. M is released until T1 handles A-Event2. An event is triggered when M is released, so that the blocked B-Event1 can be continued. Then M can be acquired successfully by B-Event1.

This method tries to improve the system synchronization primitives to make the thread not be blocked. Comparing with the previous 3 methods, this method can provide much more concurrency and throughput. And it doesn't require determining in advance which event or what synchronization objects needs to acquire, just like 3.2 or 3.3. But the implementation of this method is more complicated and needs more memory resource (for saving thread contexts). This is also a method of avoiding dead-lock because it doesn't constrain the usage of synchronization objects. But this method provides a thorough solution from the operation system primitives' layer.

It is worth mention that in the recent years, operating system academic circles promote a Servant/Exe-Flow Model based operating system[7, 8]. Its synchronization mechanism is as similar as the above method. In this operating system, the saving for the threads' contexts is performed by an object named Mini-Port. Because this operating system natively supports the similar synchronization mechanism, the Event-Driven SOA architecture implementation based on this operating system won't cause the dead-lock problem. Besides, this operation system orients the component-based environment, system components are loosely coupled, and they communicate with each other through messages, which is a similar concept as event. Therefore, essentially the operation system itself is an implementation of Event-Driven SOA architecture.

4 CONCLUSIONS

This paper discusses the dead-lock problem when using Event-Driven SOA architecture, and promotes several detailed solution against this problem. Besides the benefit of loosely coupled structure characteristic supported by SOA, Event-Driven SOA architecture provides the asynchronous event processing ability, which increases the automation and throughput of the system. These features enable Event-Driven SOA to be easily used in the complex large systems. But the more complex of the systems, the harder the dead-lock issue described in this paper is perceived. It is even possible that the dead-lock issue is caused by the in-

teraction of multiple system components. So if the dead-lock issue can be considered in the system design phase, and can be eliminated by using various solutions described in this paper, then the stability and robustness of the system can be highly improved.

In the meantime, the idea of Event-Driven SOA architecture can be not only used in software system, but also put into practice in many business processes, enterprise management, etc. The dead-lock problem can be also simulated in these non-software systems (such as the simple reality example in section 2); therefore those solutions can be greatly referred for implementing a robust, efficient process or management system.

ACKNOWLEDGEMENT

This work is financially supported by National Social Science Foundation of China (No.11BFX125); National Social Important Science Foundation of China (No.C-6501-14-06101).

REFERENCES

[1] Ling Xiaodong. 2007. A review of SOA. *Computer Applications and Software*, 4: 122-124.
[2] Brenda M. Michelson. 2006. Event-Driven Architecture Overview [OL]. Patricia Seybold Group, http://dx.doi.org/10.1571/bda2-2-06cc
[3] Levina, O., Stantchev, V. 2009. Realizing event-driven SOA. *Fourth International Conference on Internet and Web Applications and Services*, pp.37-42.
[4] Li Xinyu. 2010. *The Application of Event-Driven Architecture in Financial Trading Clearing System*. Master. Zhejiang University.
[5] Matt Wright, Antony Reynolds. 2009. *Oracle SOA Suite Developer's Guide*. Packt Publishing.
[6] Tang Ziying, Zhe Fengping, Tang Xiaodan. 2000. *Computer Operating System*. Xi'an: Xidian University Press.
[7] Gong Yuchang, Zhang Ye, Li Xi, Chen Xianglan. 2008. The kernel design of a novel component based operating system. *Journal of Chinese Computer Systems*.
[8] Wu Mingqiao, Chen Xianglan, Zhang Ye, Gong Yuchang. 2006. A new operating system construction model based on servant and executive flow. Journal of University of Science and Technology of China, 36(2): 230-236.