The Study of Resource Racing with Message-based Model

Ling Tang*

*Corresponding author: ausflug163@163.com

ABSTRACT: There is a widely used architecture model named as message-based model, which can construct a loosely coupled system and provide primitive support for rapid asynchronous transaction processing. A comprehensive message-based model server for multiple concurrent transactions and synchronization mechanism for acquiring resources are unavoidable, but there are some dead lock problems caused by inappropriate acquiring and releasing of resources among current transactions. This paper analyzed the resource contention problem caused by inappropriate usage of synchronization mechanism in message-based model, which provided several key techniques and solutions to resolve such resource contention problem from different dimensions. These solutions can provide guidelines for the system designers and developers to avoid resource dead lock risk during system design and implementation. Thus the system could be more stable and robust.

Keywords: message-based model; resource racing; message-queue

1 INTRODUCTION

Procedure-based and object-oriented models are the most classical programming models. They divide the whole system into a series of procedures or objects. The procedures and objects can be easily called and reused to complete the whole system flow. However, those models only focus on the static information (system composition), and each component is executed according to pre-defined schedule. This doesn’t well adapt to the dynamical characteristic of the system transactions.

To resolve the above insufficiency, message-based model [1, 2] is brought out and the key is the message and its handling. When executing some operations, a message is triggered. The executor completes the operation by processing the message. Message processing is asynchronously, which allows the system to perceive and respond requests in transactions quickly. Such model is very useful to design and implement a loosely coupled system. Different system components only communicate with each other by sending messages. Instead of caring about the internal flow of others, every component only needs to handle required messages. The whole system processing is composed of handling various messages. The message-based model is widely used in various software systems: various GUIs [3], database transaction processing [4], SOA systems [5], networking [6], and process/flow control [7], etc.

Generally, in any system which is based on message-based model, a message is triggered by some requests; and a message processing is performed by a component. The processing ability of the component has an upper limit, so it can only handle limited messages at the same time. As to those messages which are not handled in time, they are buffered in an explicit or implicit message-queue. Besides, the message processing provided by the component has to be performed in an executing context. Normally, such context is provided by a process (or thread) in existing major operation systems, thus the messages are circularly processed by processes/threads one by one. Beyond any concrete system implementation, a typical EBP system model is shown in Figure 1.

A group of processes/threads (Service-Pool) deals with various messages in the message-queue. Each process obtains one message from the message-queue and then processes it. After the process finishes, it continues to obtain another message from the queue and processes it. The process keeps on circulating until all messages in the queue are completed, and then the processes turn into waiting status until another new message comes into the queue.
When multiple tasks are executed in the system concurrently, it is unavoidable to use various synchronization mechanisms to protect resource contention during processing messages. However, when designing and implementing a message-based model system, the synchronization mechanisms must be used very carefully, while inappropriate usage will cause resource dead lock. Based on the analysis of the resource contention problem by synchronization mechanisms in message-based model, this paper introduced several design and implementation techniques to resolve the problem.

2 THE RESOURCE CONTENTION PROBLEM IN MESSAGE-BASED MODEL

Taking an existing message-based model system for example, assuming a company has a print department (corresponding to a component) and there are two print-operators called A and B (corresponding to Service-Pool). Each operator can only handle one print job at one time and its job is to finish the received request (corresponding to the messages) and send the printed jobs to other departments. There is only one printer (corresponding to the resource) in the department. Because the printing process is very slow, they decide to use EBP model to improve the throughput. When A receives a printing request at first, it applies the exclusive access to the printer, and then starts the printing task. When B receives another printing request, it has to wait for A’s completion. However, when A’s first printing task is on-going; A also receives the third printing request. According to the EBP model, A doesn’t wait for finishing the first printing request and then handle the new one, so A handles the third printing request immediately. However, 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.

With more general description, we assumed that some transactions include handling two messages: Message1 and Message2. A resource R is acquired when handling Message1, and then R is released when handling Message2. Furthermore, assuming there are three same transactions T1, T2 and T3 which are executed concurrently. There are two processes in the Service-Pool: P1, P2, as shown in Figure 2.

First, when T1, T2 and T3 start simultaneously, all of them send Message1 to Message-Queue. Then P1 obtains T1-Message1, and P2 obtains T2-Message1. When P1 handles T1-Message1, it acquires R successfully. While P2 handles T2-Message1, it must wait there because R is occupied.

After P1 finishes processing T1-Message1, it sends T2-Message2 for triggering the next step of the transaction. And then P1 continues to choose the next message named T3-Message1 from the Message-Queue and process it. However, R hasn’t been released, so P1 can’t acquire R either when it handles T3-Message1. Hence P1 can do nothing but only wait there. At this time, all processes in the Service-Pool are waiting for R, but the message T1-Message2 which releases R can’t be processed. Accordingly, the system falls into resource dead lock state.

In general, the reason of resource dead lock is because of the confliction of the acquired resources. In EBP model, the processes in Service-Pool are also a kind of resource, which needs to be acquired at first before handling messages. For the above-mentioned...
transactions, the acquiring sequence of the resources is shown in Figure 3.

It can be found that there are two opposite resource acquiring sequences:
1. Acquiring R after acquiring process resource.
2. Acquiring process resource after acquiring R.

Such a resource contention problem can be similarly promoted to the systems of larger scale: if there are N processes in the Service-Pool, when the concurrency for acquiring some resources reaches N+1, the resource contention problem can be triggered.

Resource deadlock is typically described by resource allocation graph (RAG), which is composed of a set of edges between requester and resources. An edge from a requester to a resource means the requester waits for the resource, and an edge from a resource to a requester means the resource is held by a requester. A loop in RAG is a necessary occurrence condition of dead lock.

The RAG of the above dead lock scenario could be described by Formula (1):

\[
\text{Requests} = \{T_1, T_2, T_3\}; \quad \text{Resources} = \{R, SP\}, SP: \text{Service Pool}
\]

\[
\text{Edge} = \{R \rightarrow T_1, SP \rightarrow T_3, SP \rightarrow T_2, T_3 \rightarrow R, T_2 \rightarrow R, T_1 \rightarrow SP\};
\]

The corresponding graph is shown in Figure 4.

Apparently there are loops in RAG, and SP is referenced by three edges of the loops but it has only 2 elements. So the deadlock is doubtless.

Therefore, as an implicit resource, the processes which handle the messages conflict with the explicit resource R, which finally cause the dead lock of resource. There are two points of essence leading to this phenomenon:
1. The operations of acquiring and releasing the resource are performed in the process of two different messages. In the period between the two messages, all processes in the Service-Pool may be blocked in the process of acquiring resources; thereby the message for releasing the resources can’t be scheduled by an available process.
2. A process can’t schedule the other message once it is blocked while acquiring the resources.

Generally, in the design and implementation of systems based on the event-based programming, it must be very cautious when acquiring exclusive resources and the confliction with service processes must be considered carefully.

3 SOLUTIONS

In general, causing resource deadlock must satisfy four necessary conditions [8]: 1. Mutual exclusion; 2. Hold and wait; 3. Non-preemption; 4. Circular wait. To resolve the resource deadlock problem, one of the four conditions must be broken. The method to handle resource deadlock can be categorized into three types [8]: 1. Prevent dead lock; 2. Avoid dead lock; 3. Detect and relieve dead lock.

Among them, the third method needs to relieve the dead lock by terminating some attending processes after the deadlock is blocked. The transaction logic needs special processing to adapt to be suddenly terminated during execution, and this brings huge complexity into the programming design, thus it can’t be commonly used in various EBP-based systems. So our following solutions mainly focus on the first and second methods. There are mainly four resolutions being proposed.

3.1 Constrain the releasing point of resource

This method requires that, if a resource is acquired when handling a message, the resource must be released in the same message processing step. Apparently, this method can make sure that the process isn’t acquired after acquiring the resource, thus the dead lock could not happen. This belongs to the method of preventing dead lock.

By this method, there won’t be the scenario that service processes are all in the process of acquiring resources, since the resource must be able to be released after being acquired. But the restriction of acquiring and releasing a resource in one message is too strict, because a complicated transaction may have a
large amount of processing work after acquiring a resource. This work may be not suitable to be implemented in one message. For example, after acquiring a resource, a transaction may want to do a series of asynchronous I/O, and the resource cannot be released unless the I/O is finished. To meet this requirement, the message processing must wait for I/O’s completion, so the service process keeps being occupied and cannot serve other messages, which affects the system concurrency and throughput severely. This method actually constrains the asynchronous feature of EBP-based system. Therefore, it can only apply to the simple systems which don’t have high throughput requirement.

3.2 Bind message handler process

This method means that, when a process handles a message, once a resource is acquired, the process is bound with the transaction which sends the message. Then all later messages which are sent in this transaction must be handled by this process, until the message which releases the resource is processed. While a process is bound with some transaction, it cannot handle the other messages which belong to other transactions and need to acquire some resources. In this way, this method also makes sure that process is acquired after acquiring the resource won’t appear. Because the process has been bound with the transaction, it is always available after acquiring the resource. This is shown in Figure 5.

When P2 is blocked in the process of waiting for R, the RAG at that moment is described as Formula (2).

Figure 5. Bind message handler process.

\[
\text{Request} = \{T1, T2, T3\}; \quad \text{Resource} = \{R, SP\};
\]

\[
\text{Edge} = \{R \rightarrow T1, SP \rightarrow T1, SP \rightarrow T2, T2 \rightarrow R, T3 \rightarrow SP\};
\] (2)

Apparently, there isn’t any loop in this RAG, so the deadlock is impossible. The basic idea of this method is reserving the process which holds the resource, to make sure that the resource can be released in this process. This looks similar to section 3.1. But the difference is that, this method doesn’t constrain how to acquire and release the resource. The resolution is resolved in the system architecture layer, and the actual transaction won’t see any special processing (i.e., the logic of how the message is handled need not special processing). Therefore, this method belongs to the method of avoiding deadlock.

However, in the period when the process is bound, it can’t handle other transactions’ messages which need to acquire resources, so this method also constrains the concurrency and throughput of the systems. But comparing with section 3.1, even after binding in this method, actually the process can still handle those messages which don’t need to acquire resources, so its concurrency and throughput are better.

3.3 Multiple level message-queues

The above-mentioned two methods focus on ensuring the messages of acquiring and releasing resources can be handled in the same process. But this requirement is too strict. Actually it is only necessary that the message of releasing resource could be handled by some process, but it is not a requirement that the process must be as same as the one which acquires the resource.
In order to achieve that, this method defines dedicated Message-Queue and Service-Pool for the messages which acquire resources. Still considering the example in section 2, because Message1 needs to acquire R, Message-Queue2 and Service-Pool2 are defined dedicatedly for handling the messages which needs to acquire R, as shown in Figure 6.

Assuming there is only one process P3 in Service-Pool2. When transaction T1, T2 and T3 start, they send Message1 to Message-Queue2. Then P3 processes T1-Message1 at first, which can acquire R successfully and send T1-Message2 and after that it processes T2-Message1. However, since R has been acquired by T1-Message1, P3 has to wait. Whereas T1-Message2 doesn’t need to acquire R, T1-Message2 is handled by Service-Pool2, but by Service-Pool1 instead. So T1-Message2 is sent to Message-Queue1, the process in Service-Pool1 can handle T1-Message2, and then R can be released properly.

When P3 is blocked in the process of waiting for R, the RAG at that moment is described as Formula (3).

\[
\text{Request} = \{T1, T2, T3\}; \quad \text{Resource} = \{R, SP1, SP2\}; \\
\text{Edge} = \{R \to T1, T1 \to SP1, SP2 \to T2, T2 \to R, T3 \to SP1\};
\]

(3)

So there isn’t any loop in this RAG either, and the deadlock won’t happen. In this way, because all messages which acquire R are handled by Service-Pool2, the processes in Service-Pool1 are never blocked by R; therefore the messages for releasing R can always be handled properly.

Ideally, each single resource needs to specify a corresponding Message-Queue and Service-Pool. A large scale system may use many resources, so it isn’t reasonable to specify Message-Queues and Service-Pools for every resource. As an optimization method, it can be defined according to the categories of the resources. For example, some transactions acquire the resource R in Message1 and release R in Message2. But some other transactions acquire the resource S in Message1 and release S in Message2. Meanwhile, if there isn’t any relationship between R and S, i.e., there isn’t any transaction which needs to acquire R and S simultaneously, R and S can be considered into the same category, and they can be handled with the same Message-Queue and Service-Pool.

Nevertheless, if some transactions acquire S after acquiring R, R and S should be considered in different categories, and they can’t share the same Message-Queue and Service-Pool, or the resource deadlock can be triggered. For example, assuming transaction T1 and T2 need to acquire R in Message1, and S in Message2, and finally release R and S in Message3, both Message1 and Message2 are handled in the same Message-Queue and Service-Pool, as shown in Figure 7.

When P3 is blocked in the process of waiting for R, the RAG at that moment is described as Formula (4).

\[
\text{Request} = \{T1, T2\}; \quad \text{Resource} = \{R, SP2\}; \\
\text{Edge} = \{R \to T1, T1 \to SP2, SP2 \to T2, T2 \to R\};
\]

(4)

This RAG has a loop and SP2 is referenced twice but there is only one element in it, which demonstrates the occurrence of deadlock.

The essential problem is that T1-Message1 for acquiring R and T1-Message2 for acquiring S are all handled by P3, so the step for releasing R can’t be scheduled. To resolve such deadlock scenario, R and S should be considered in different categories, thus dif-
different service processes should be used. This could be illustrated in Figure 8.

With this solution, when P3 is blocked in the process of waiting for R, the RAG at that moment is described as formula 5.

\[
\text{Request} = \{T1,T2\}; \quad \text{Resource} = \{R,SP1,SP2\}; \\
\text{Edge} = \{R \rightarrow T1, S \rightarrow T1, T1 \rightarrow SP1, SP2 \rightarrow T2, T2 \rightarrow R\};
\]

(5)

So the loop in RAG is eliminated, the dead lock won’t occur.

Resources are categorized depending on how the resources are acquired. At first, resources level is introduced as the following definitions:

1. Within a transaction, before a resource is released, if there isn’t any other resource being acquired, its level is 1;
2. Within a transaction, before a resource is released, if there are N resources being acquired, its level is N+1;
3. For one specific resource, if different transactions give different levels, the maximum one is chosen as the resource’s level;
4. For the resource in point 3, if its level is changed from X to Y in some transactions, it needs to increase the levels of the resources which have higher level in this transaction by Y-X.
5. So eventually, each level corresponds to one category, and all resources having the same level are divided into the same category. And each category is specified with a unique Message-Queue and Service-Pool.

As shown in Figure 6, Service-Pool2 only corresponds to one resource R. Only one process is enough for Service-Pool2 in this case, because there is only one message can acquire R at one time. But if the resource can be shared by multiple transactions simultaneously; or if the corresponding category contains multiple resources, increasing the process number in Service-Pool can improve concurrency and throughput of handling those messages which acquire the resources.

Relative to the methods in section 3.1 and 3.2, this method takes more memory for defining new Message-Queues and Service-Pools. The benefit is that it only affects concurrency of handling the messages which acquire resources. But the other messages are still handled in the original Message-Queue and Service-Pool, and their concurrency and throughput aren’t affected. Because the usage of the resources isn’t constrained in the transactions, this method also belongs to the method of avoiding dead lock.

3.4 Improve resource synchronization mechanism to avoid process blocking

According to the description of the essence of this resource deadlock problem in section 2, the previous three methods tries to resolve the problem against the first point, i.e., making sure the message which releases the resources can be assigned to an available process. But if the second point can be resolved, i.e., a process can handle other messages even it is waiting for acquiring a resource (i.e., a process is never blocked), the message which releases resources can always be scheduled by an available process.

In today’s popular operating systems, the blocking mechanism of the resources is the basic system synchronization primitive. To make a process not be blocked, new synchronization primitives must be implemented. When a process needs to block on a resource, the current execution context (stacks and registers) of the process needs to be kept at somewhere else, and then the process can go to handle other messages. When a resource is released, if there is any waiter on it, a message is sent to activate the waiter. The process for this message obtains the waiter’s execution context which is saved when the waiter is blocked, and then the context is loaded into the process which handles the message, so that the waiter can be restored from the blocked point. For example, assuming transactions T1 and T2 acquire resource R in Message1 and release R in Message2, there is only one process in the Service-Pool. The procedure is illustrated in Figure 9.

When process P1 handles T2-Message1, because it can’t acquire R, it saves its current execution context.
Then T1 continues to handle the following messages. R isn’t released until P1 handles T1-Message2. A message is triggered when R is released, so that the blocked T2-Message1 can be continued. Then R can be acquired successfully by T2-Message1.

Even though this solution looks complicated, its RAG analysis is extremely simple. When P1 is blocked on waiting for R, the RAG at that moment is described as Formula (6).

\[ \text{Request} = \{T1, T2\}; \quad \text{Resource} = \{R, SP\}; \]
\[ \text{Edge} = \{R \rightarrow T1, T2 \rightarrow R\}; \quad (6) \]

Apparently this RAG is so simple and won’t cause dead lock. This method tries to improve the system synchronization primitives to make the process not be blocked. Comparing with the previous three methods, this one can provide much more concurrency and throughput. It doesn’t require determining in advance which messages need to acquire resources, or what resources need to be acquired, just like section 3.2 or 3.3. But the implementation of this method is more complicated and needs more memory resource for saving process contexts. This is also a method of avoiding dead lock because it doesn’t constrain the usage of resources, but it provides a thorough solution from the operation system primitives’ layer.

It is worth mentioning that in the recent years, academic circles of operating system propose an operating system based on Servant/Exe-Flow Model [9, 10]. Its synchronization mechanism is similar to the above method. In this operating system, the saving for the process’ contexts is performed by an object named Mini-Port. Because this operating system natively supports the similar synchronization mechanism, the EBP architecture implementation based on this operating system won’t cause the resource deadlock problem. Besides, this operation system orients the component-based environment, and system components are loosely coupled. They communicate with each other through messages; therefore in essence the operation system itself is an implementation of EBP-based architecture.

4 CONCLUSIONS

This paper discusses the resource contention problem using Event-Based Programming model, and proposes four detailed solutions against this problem. Table 1 shows the summary of these solutions:

<table>
<thead>
<tr>
<th>Solutions</th>
<th>Complexity</th>
<th>Performance</th>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constrain the releasing point</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Bind message handler process</td>
<td>Middle</td>
<td>Middle</td>
<td>High</td>
</tr>
<tr>
<td>Multiple level message-queues</td>
<td>High</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Improve resource synchronization mechanism</td>
<td>High</td>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>

The first method is very simple, but it has strict limitation on how resources are used. So it can’t adapt to asynchronous scenario, and the performance and applicability are poor. The other three methods have no restriction, so they could be applied to any scenario. The second method binds some processes, which decreases the concurrency, so its performance isn’t as good as the others. The third method needs to categorize the resources, and more memory is required for extra Message-Queues and Service Pools. The fourth method needs to implement new operation system primitives. Even though their implementation is more complex, they could bring best concurrency and asynchronous performance.

Besides the benefit of loosely coupled structure characteristic is supported by Message-Based Model, and this architecture provides the asynchronous message processing ability, which increases the automation and throughput of the system. These features enable Message-Based Model to be easily used in the complex large systems. But the more complex the systems are, the harder the resource contention issue which is described in this paper is perceived. It is even possible that the resource contention issue would be caused by the interaction of multiple system components. So if the resource contention issue can be considered in the system design phase, and can be eliminated by using various solutions described in this paper, the stability and robustness of the system can be highly improved. Depending on the concrete appliance scenario, different solutions above could be chosen.

In the meantime, the idea of Message-Based Model can be not only used in software system, but also put into practice in many business processes, enterprise management, etc.. The resource contention 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 supported by National Social Science Foundation of China (No.11BFX125), National Important Social Science Foundation of China (No. 14ZDB147) and China Law Society 2016 Ministerial research topics (CLS (2016) D18).
REFERENCES