The Scheduling of Resources in Program Architecture

Ling Tang
Dept. Information Science and Technology, Criminalistics School
East China University of Political Science and Law
Shanghai, China, 201620
E-mail: ausflug163@163.com

***Abstract*—As a programmer, there are some difficulty in resource scheduling. Most of them are caused by inappropriate acquiring and releasing resources among concurrent transactions. The author analyzes the resource scheduling problem caused by inappropriate usage of synchronization mechanism, and then provides several methods to resolve this problem from different perspectives. These methods can provide some guidelines for computer programmers and the way to solve resource scheduling problem.**

***Keywords*—Resource Scheduling; Message; DeadLock; Service Buffer**

I. Introduction

Procedure-Based [1] and Object-Oriented [2] Programming are the most classical programming models. They divide the whole system into a series of procedures or objects. The procedures and objects can be called and re-used easily to complete the whole system flow. But such models only focus on the static information (system composition), each component executes according to pre-defined schedule. This doesn't well adapt to the dynamical characteristic of system transactions.

To resolve the above insufficiency, Event-Based Programming (EBP) [3, 4] model is brought out. It is also called Event-Driven Programming. The essential of this model is event and its handling. For understanding conveniently, event is also called as message.

Generally, in any system which based on EBP model, message is triggered by some request; message processing is performed by a component. The processing ability of the component has upper limit, so it can only handle more or less limited messages at the same time. As to those messages which haven't been 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 nowadays major operation systems. Thus the messages are processed by processes/threads one by one circularly. The group of these processes/threads is called as Service-Buffer.

This work is supported by National Social Science Foundation of China (No.11BFX125).
This work is supported by National Important Social Science Foundation of China (No. 14ZDB147)

II. The Resource Allocation Problem

The resource allocation problem is because of the confliction of the acquired resources. In EBP model, the processes in Service-Buffer are also a kind of resource; such resource needs to be acquired at first before handling messages. Considering a set of transactions, each one has two messages handled in sequence. In the first message, a resource is required, and in the second message, the resource is released.

Furthermore, assuming there are three same transactions T1, T2 and T3 are executed concurrently. There are two processes in the Service-Buffer: P1 and P2. This is shown in Fig. 1.

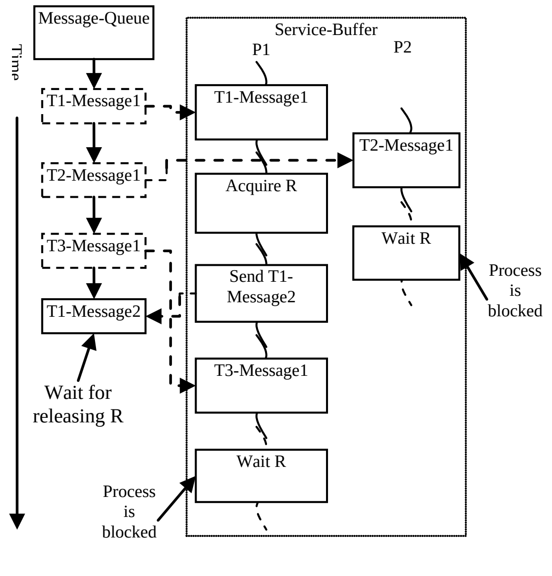

Fig. 1. Resource allocation problem sketch

The execution procedure is described as below:

1) When T1, T2 and T3 are started, all of them send Message1 to Message-Queue;
2) P1 obtains T1-Message1 to process;
3) P2 obtains T2-Message1 to process;
4) When P1 handles T1-Message1, it acquires R successfully;
5) When P2 handles T2-Message1, it can't acquire R, so it must wait there;

Copyright © 2017, the Authors. Published by Atlantis Press.
This is an open access article under the CC BY-NC license (http://creativecommons.org/licenses/by-nc/4.0/).

6) When P1 finishes processing T1-Message1, it sends T1-Message2 for triggering the next step of the transaction;
7) Then P1 continues to choose the next message T3-Message1 from the Message-Queue and process it;
8) But because R hasn't been released, P1 can't acquire R either when it handles T3-Message1. So P1 can do nothing but only wait there;
9) At this time, all processes in the Service-Buffer are waiting for R, but the message T1-Message2 which releases R can't be processed by any process. Accordingly, the system falls into resource dead lock state.

If we treat the processes in Service-Buffer as another kind of resource, for the above transactions, the acquiring sequence of the resources is shown as Fig. 2.

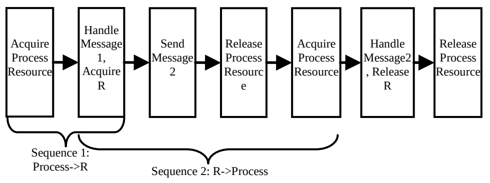

Fig. 2. Resource acquiring sequence schematic

It can be found that there are two opposite resource acquiring sequences:

1) Acquiring R after acquiring process.
2) Acquiring process 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-Buffer, when the concurrency for acquiring some resources reaches N+1, the resource-contention problem can be triggered.

Therefore, as an implicit resource, the processes which handle the messages conflicts with the explicit resource R, and this finally causes resource dead lock. There are two points of essence leading to this phenomenon:

Firstly, the operations of acquiring and releasing resource are performed in the process of two different messages. In the period between the two messages, all processes in the Service-Buffer may be blocked on acquiring resources; thereby the message for releasing the resources can't be scheduled by an available process.

Secondly, a process can't schedule the other message once it is blocked on acquiring resources.

RAG (Resource Allocation Graph) is often used to describe how the resources are acquired by concurrent transactions. A RAG is composed by nodes and edges. Nodes include resources and requesters. Edges represent the relationship between resources and requesters. An edge from resource to requester means the resource is held by the requester; and an edge from requester to resource means the requester is waiting for the resource. So if some edges form a loop in the graph, then it means there is dead lock in the corresponding resource allocation scenario.

In the above procedure, its RAG is represented as formula (1) (SB means the processes in Service-Buffer):

$$\begin{gathered}\text{Request} = \{T1, T2, T3\};\ \text{Resource} = \{R, SB\};\\ \text{Edge} = \{R- > T1, T1- > SB, SB- > T2, T2- > R, SB- > T3, T3- > R\};\end{gathered} \tag{1}$$

Apparently this RAG forms a loop, which implies dead-lock.

All in all, when designing and implementing the Event-Based Programming based systems, it must be very cautious when acquiring exclusive resources, the confliction with service processes must be considered carefully.

III. METHODS

In general, causing resource dead lock must satisfy four necessary conditions [5]: 1.Mutual exclusive; 2.Hold and wait; 3.Non-preemption; 4.Circular wait.

To resolve the resource dead lock problem, one of the four conditions must be broken. The method to handle resource dead lock can be categorized into three types [5]: 1.Prevent dead lock; 2.Avoid dead lock; 3.Detect and relieve dead lock.

The 3rd category of method needs to relieve the dead lock by terminating some attending processes when detecting the dead lock. The transaction 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 system. So our proposed solutions mainly focus on the first and second category of methods. There are mainly three resolutions being promoted as follows:

1) Constrain release point of resource
2) Bind message handler
3) Multiple level message-queues

A. *Constrain Release 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 is a method of preventing dead lock.

By this method, there won't be the scenario that service processes are all blocked on the resource, 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 complicate transaction may have a lot 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. This affects the system concurrency and throughput severely. This method actually constrains the asynchronous feature of EBP based system. Therefore, it can only apply to simple systems which don't have high throughput requirement.

B. Bind Message Handler

This method means when a process handles a message, once a resource is acquired; the process is bound with the transaction which sends the message. Then all afterward 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 it won't appear that process is acquired after acquiring the resource. Because the process has been bound with the transaction, the process is always available after acquiring the resource. This is shown in Fig. 3.

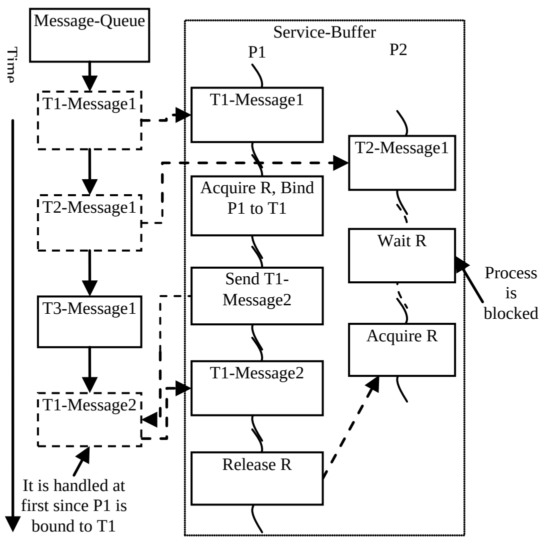

Fig. 3. Bind message handler process

1) When transactions T1, T2, T3 are started, all of them send Message1 to Message-Queue;

2) P1 obtains T1-Message1 to process;

3) P2 obtains T2-Message1 to process;

4) When P1 handles T1-Message1, it acquires R successfully;

5) Once R is acquired, P1 is bound to T1, so P1 can only handle T1's messages;

6) When P2 handles T2-Message1, it can't acquire R, so it must wait there;

7) When P1 finishes processing T1-Message1, it sends T1-Message2 for triggering the next step of the transaction;

8) P1 continues to choose the next message from the Message-Queue, since P1 is bound to T1, so P1 can't choose T3-Message1, but it has to choose T1-Message2;

9) When P1 handles T1-Message2, R could be released;

10) Then P2 could be woken up and acquire R successfully; no dead lock could happen.

When P2 is blocked on waiting for R, the RAG at that moment is described as formula (2).

$$\begin{aligned}&\text{Request} = \{T1, T2, T3\};\text{Resource} = \{R, SB\};\\&\text{Edge} = \{R->T1, SB->T1, SB->T2, T2->R, T3->SP\};\end{aligned} \quad (2)$$

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 A. But the difference is that, this method doesn't make constraint to how to acquire and release the resource. 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 message is handled need not special processing). Therefore, this method belongs to the method of avoiding dead lock.

But 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 A, 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 than A.

C. 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, it is not a requirement that the process must be as same as the one which acquires the resource.

In order to achieve this, this method defines dedicated Message-Queue and Service-Buffer for the messages which acquire resources. Still considering the example in section II, because Message1 needs to acquire R, Message-Queue2 and Service-Buffer2 are defined dedicatedly for handling the messages which needs to acquire R. This is shown in Fig. 4.

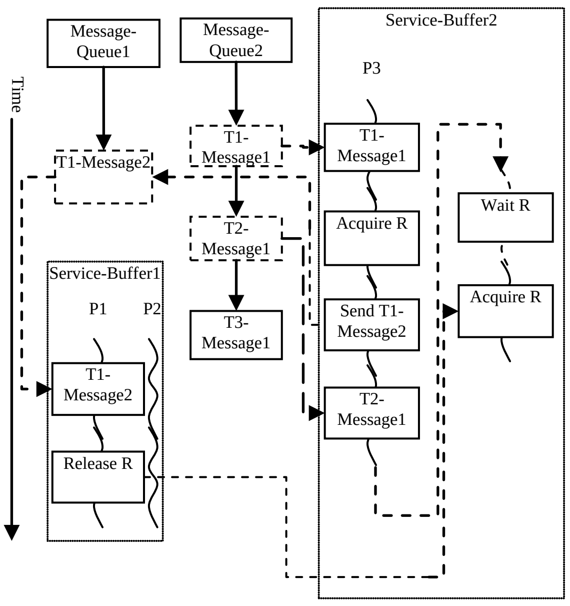

Fig. 4. Principles of multiple level message-queues

1) Assuming there is only one process P3 in Service-Buffer2.
2) When transaction T1, T2 and T3 start, they all send Message1 to Message-Queue2;
3) Then P3 processes T1-Message1 at first, it can acquire R successfully and send T1-Message2;
4) Then P3 processes T2-Message1, but since R has been acquired by T1-Message1, P3 has to wait;
5) But because T1-Message2 doesn't need to acquire R, T1-Message2 isn't handled by Service-Buffer 2, but by Service-Buffer1 instead;
6) So T1-Message2 is sent to Message-Queue1, the process in Service-Buffer1 can handle T1-Message2;
7) Then R can be released properly;
8) Then P3 is woken up and R could be acquired properly.

When P3 is blocked on waiting for R, the RAG at that moment is described as formula (3).

$$\begin{aligned}&Request = \{T1, T2, T3\}; Resource = \{R, SB1, SB2\};\\&Edge = \{R->T1, T1->SB1, SB2->T2, T2->R, T3->SB2\};\end{aligned} \quad (3)$$

So there isn't any loop in this RAG either, the dead lock won't happen. In this way, because all messages which acquire R are handled by Service-Buffer2, the processes in Service-Buffer1 are never blocked by R; therefore the messages for releasing R can always be handled properly.

Ideally, each single resource needs to specify with a corresponding Message-Queue and Service-Buffer. A large scale system may use many resources, it isn't reasonable to specify Message-Queues and Service-Buffers for every resource. As an optimization, 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, then R and S can be considered into the same category, they can be handled with the same Message-Queue and Service-Buffer.

Nevertheless, if some transactions acquire S after acquiring R, then R and S should be considered in different categories, they can't share the same Message-Queue and Service-Buffer, or the resource dead lock can be triggered.

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

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 is 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 transaction, then increase the levels of the resources which have larger level in this transaction by Y-X.

So eventually, each level corresponds to one category, all resources have the same level are divided into one same category. And each category is specified with a unique Message-Queue and Service-Buffer.

It is worth mention that in the recent years, operating system academic circles promote a Servant/Exe-Flow Model based operating system [6]. Its synchronization mechanism is as similar as the above method. In this operating system, the saving for the thread's 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 dead-lock problem.

IV. CONCLUSIONS

This thesis discusses the resource contention problem when using Event-Based Programming model, and promotes three detailed solutions against this problem.

The first method is very simple, but it does strict limitation on how resources are used, so it can't adapt to asynchronous scenario, the performance and applicability are poor. The other two methods require no restriction, so they could be applied to any scenario. The second method binds some processes. This 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 Buffers.

EBP model has the benefit of loosely coupled architecture; this makes it easily be used in a complex and 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 interaction among multiple system components. So if the dead-lock issue can be considered in the system design phase, and can be eliminated by using the solutions described in this paper, then the stability and robustness of the system can be highly improved. Depending on the concrete appliance scenario, different solution described above could be chosen.

REFERENCES

[1] URL:https://en.wikipedia.org/wiki/Procedural_programming
[2] URL:https://en.wikipedia.org/wiki/Object-oriented_programming
[3] Ted Faison, Event-Based Programming, Apress, 2006.
[4] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, et al., "Event-driven Programming for Robust Software", In Proceedings of the 10th workshop on ACM SIGOPS, 2002.
[5] Tang Zi-ying, Zhe Feng-ping, Tang Xiao-dan, Computer Operating System, XiAn: XIDIAN UNIVERSITY PRESS, 2000. (In Chinese)
[6] Gong Yu-chang, Zhang Ye, Li Xi, Chen Xiang-lan, "The Kernel Design of A Novel Component Based Operating System", Journal of Chinese Computer Systems, 2008. (In Chinese)