The Resolution on Dead-Lock Problem in Message Driven Model

Ling Tang
College of Criminal Justice, East China University of Political Science and Law
Shanghai, China, 201620
E-mail: ausflug163@163.com

***Abstract*—this article introduces a new model named Message Driven Model (MDM) .It is a widely used architecture model to construct loosely coupled system Inappropriate acquiring and releasing locks among concurrent procedures always lead to dead-lock problems. So the dead-lock problem caused by inappropriate usage of synchronization mechanism in MDM model is introduced. Then several solutions to resolve such dead-lock problem from different dimensions are brought out. These solutions can provide guidelines for the system designers and developers to avoid dead-lock risk during system design and implementation. In this way, the system could be more stable and reliable.**

Keywords- Message-Driven; Dead-Lock; Event-Queue; Thread-Pool

I. INTRODUCTION

Nowadays ,there are two classical programming models which are Procedure-Based model and Object-Oriented Programming model. The whole system are divided into a series of procedures or objects by these models. The procedures and objects can be called and re-used easily to complete the whole system flow. However, these models only focus on the static information (system composition), each component executes according to pre-defined schedule. It can not l adapt to the dynamical characteristic of system transactions very well.

In order to resolve the above insufficiency, Message Driven Model (MDM) is carried out. It is also called Event-Driven Model. The essential of this model is message and its handling. For understanding conveniently, message is also called as event. When executing some operation, an event is triggered. The executor completes the operation by processing the event. Event handling is asynchronously, this 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 events. Every component does not need to care about the internal flow of others, but only needs to handle required events. The whole system processing is composed of the processing of various events. MDM is widely used in various software systems: various GUIs [1], database transaction processing [2], SOA systems [3], networking [4], and process/flow control [5], etc.

Generally, in any system which based on MDM model, event is triggered by some request; event handling is performed by a component. The processing ability of the component has upper limit, so it can only handle more or less limited events at the same time. As to those events which haven't been handled in time, they are buffered in an explicit or implicit Event-Queue. Besides, the event processing provided by the component has to be performed in an executing context. Normally, such context is provided by a thread (or process) in nowadays major operating systems. Thus the events are processed by threads/processes one by one circularly. Beyond any concrete system implementation, a typical MDM based system can be abstract as Figure 1.

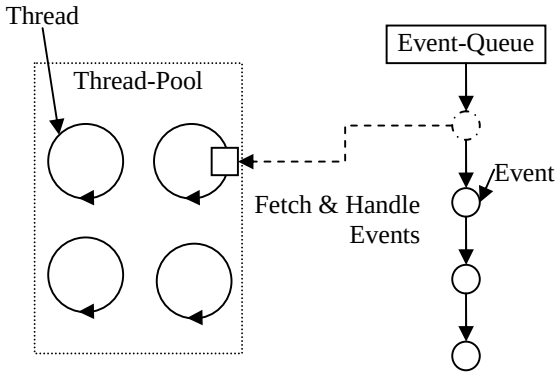

Figure 1 MDM

A group of threads handle various events in the Event-Queue. Each thread obtains one event from the Event-Queue and then processes it. After the process finishes, it continues to obtain another event from the queue and processes it. The thread keeps on circulating until all event in the queue are completed. Then the threads turn into waiting status to wait until another new event comes into the queue.

When multiple tasks execute concurrently in the system, it is unavoidable to use various synchronization mechanisms to protect lock contention during processing events. However, when designing and implementing an MDM based system; the synchronization mechanisms must be used very carefully; inappropriate usage will cause dead-lock. Based on the analysis of dead-lock problem while using synchronization mechanisms in MDM model, this paper describes several design and implementation techniques to resolve the dead-lock problem in detail.

II. THE DEAD-LOCK PROBLEM IN MDM

Taking an MDM based system in reality, for example, assuming a company has a print department (corresponding to a component); there are two print-operators called A and B (corresponding to Thread-Pool). Each operator can only

handle one thing at one time. Each one's job is printing the incoming request (corresponding to the events) and sending the result papers to the other departments. There is only one printer (corresponding to the lock) in the department. Because the printing process is very slow, they decide to use MDM model to improve the throughput. When A receives a printing request at first, it applies the exclusive access of the printer, and then starts the printing task. When B receives another printing request, it 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 MDM 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. 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.

With a more general description, assuming some procedures include handling two events: E1 and E2, a lock L is acquired when handling E1, and then L is released when handling E2. Furthermore, assuming there are three same procedures P1, P2 and P3 are executed concurrently. There are two threads in the Thread-Pool: T1 and T2. This is shown in Figure 2.

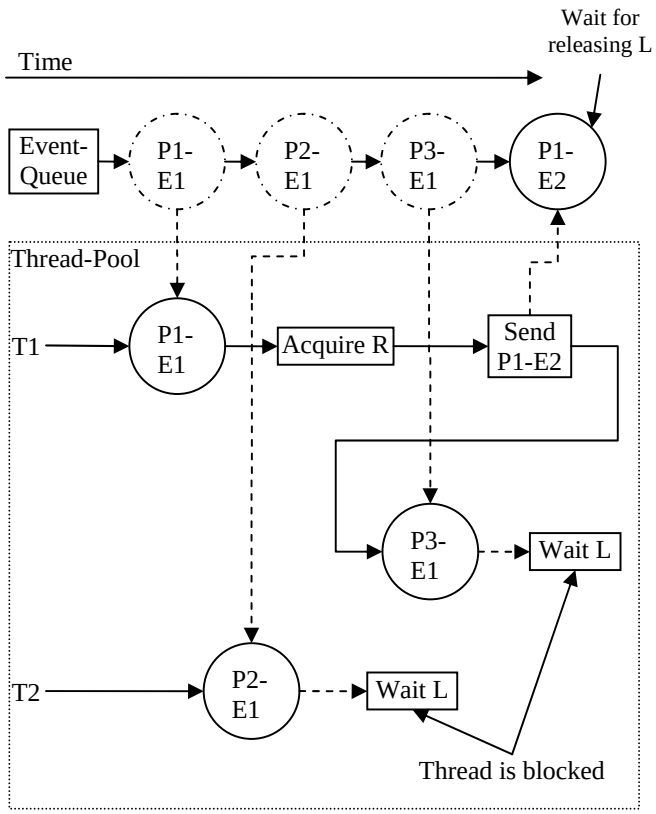

Figure 2 Dead-Lock Problem Schematic

The following procedure is executed:

1. When P1, P2 and P3 are started, all of them send E1 to Event-Queue;
2. T1 obtains P1-E1 to process;
3. T2 obtains T2-E1 to process;
4. When T1 handles P1-E1, it acquires L successfully;
5. When T2 handles P2-E1, it can't acquire L, so it must wait there;
6. When T1 finishes processing P1-E1, it sends P1-E2 for triggering the next step of the procedure;
7. Then T1 continues to choose the next event P3-E1 from the Event-Queue and process it;
8. But because L hasn't been released, T1 can't acquire L either when it handles P3-E1. So T1 can do nothing but only wait there;
9. At this time, all threads in the Thread-Pool are waiting for L, but the event P1-E2 which releases L can't be processed by any thread. 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 Thread-Pool, when the concurrency for acquiring some locks reaches N+1, the dead-lock problem can be triggered. Generally, the reason of dead-lock is because of the conflict of the acquired resources. In MDM, the threads in Thread-Pool are also a kind of resources, such resource need to be acquired before handling events. For the above-mentioned procedures, the resources acquiring sequence is shown as Figure 3.

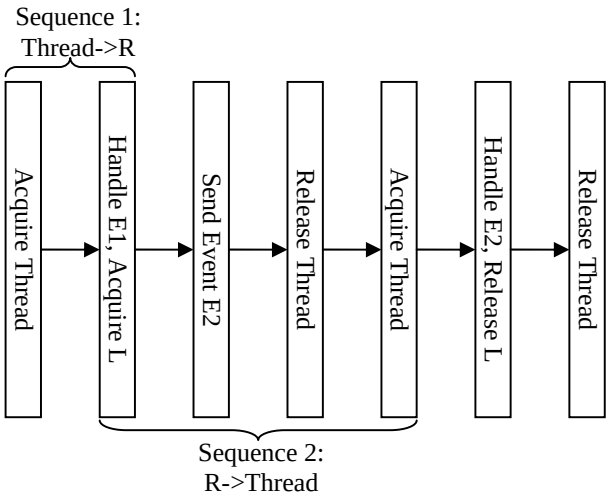

Figure 3 Resource Acquiring Conflict Schematic

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

1. Acquiring L after acquiring thread.
2. Acquiring thread after acquiring L.

Therefore, as an implicit resource, the threads which handle the events conflicts with the explicit resource L, and this finally causes dead-lock. 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 events. In the period between the two events, all threads in the Thread-Pool may be blocked on acquiring locks, thereby the event for releasing the locks can't be scheduled by an available thread.

2. A thread can't schedule the other event once it is blocked on acquiring locks.

All in all, when designing and implementing the MDM based systems, it must be very cautious when acquiring exclusive locks, the conflict with threads must be considered carefully.

III. SOLUTIONS

In general, causing dead-lock must satisfy four necessary conditions [6] as follows:

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 [7]:

1. Prevent dead-lock;
2. Avoid dead-lock;
3. Detect and relieve dead-lock.

Among them, the 3rd method needs to relieve the dead-lock by terminating some attending threads when detecting dead-lock. The procedure 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 MDM based system. So our following solutions mainly focus on the 1st and 2nd methods. There are mainly four resolutions being promoted as follows:

1. Synchronizing lock release
2. Attach event handler thread
3. Dedicate event-queues
4. Prevent thread blocking

A. Synchronizing Lock Release

This method requires that, if a lock is acquired when handling a event; the lock must be released in the same event handling step. Apparently, this method can make sure that the thread isn't acquired after acquiring the lock, thus the dead-lock could not happen. This belongs to the method of preventing dead-lock. It is shown in Figure 4.

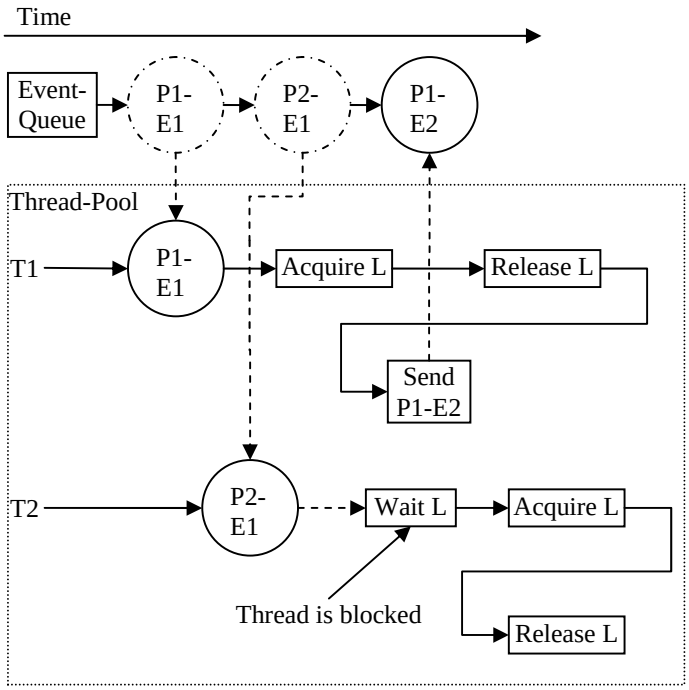

Figure 4 Synchronizing Lock Release

The following procedure is executed:

1. When P1, P2 are started, all of them send E1 to Event-Queue;
2. T1 obtains P1-E1 to process;
3. T2 obtains P2-E1 to process;
4. When T1 handles T1-E1, it acquires L successfully;
5. When T2 handles T2-E1, it can't acquire L, so it must wait there;
6. Before T1 finishes processing T1-E1, it must release L;
7. Then T2 could be woken up and acquire L successfully; no dead-lock could happen.

By this method, there won't be the scenario that threads are all blocked on the lock, since the lock must be able to be released after being acquired. But the restriction of acquiring and releasing a lock in one event is too strict. Because a complicate procedure may lead to a lot of processing work after acquiring a lock. These processing cannot be able to be implemented in one event. For example, after acquiring a lock, a procedure may want to do a series of asynchronous I/O, and the lock cannot be released unless 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 MDM based system. Therefore, it can only apply to the simple systems which don't have high throughput requirement.

B. Attach Event Handler Thread

This method means that, when a thread handles an event, once a lock is acquired, the thread is attached with the procedure which sends the event. Then all afterward events which are sent in this procedure must be handled by this thread, until the event which releases the lock is processed. While a thread is attached with some procedure, it cannot handle the other events which belong to other procedures which need to acquire some locks. By this way, this method also makes sure it won't appear that thread is acquired after acquiring the lock. Because the thread has been attached with the procedure, the thread is always available after acquiring the lock. This is shown in Figure 5.

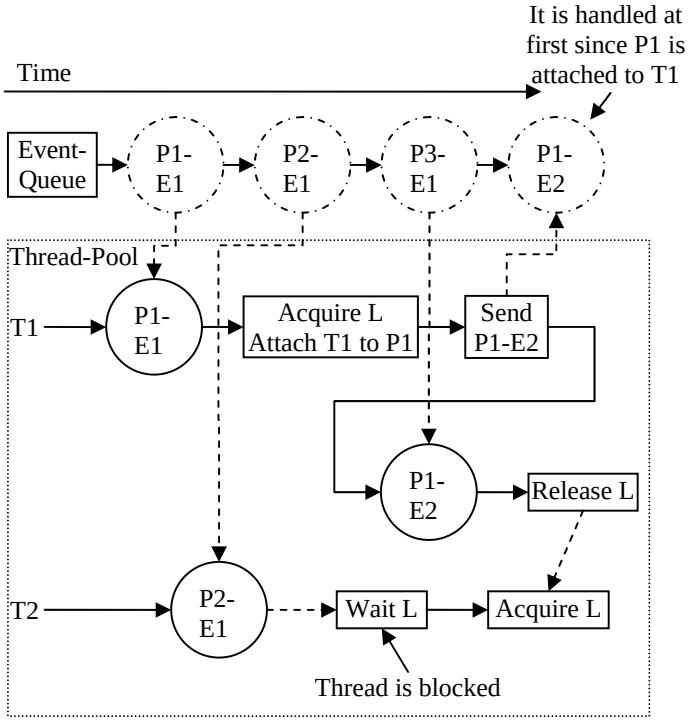

Figure 5 Attach Event Handler Thread

The following procedure is executed:

1. When P1, P2, P3 are started, all of them send E1 to Event-Queue;
2. T1 obtains P1-E1 to process;
3. T2 obtains P2-E1 to process;
4. When T1 handles P1-E1, it acquires L successfully;
5. Once L is acquired, T1 is attached to P1, so T1 can only handle P1's events;
6. When T2 handles P2-E1, it can't acquire L, so it must wait there;
7. When T1 finishes processing P1-E1, it sends P1-E2 for triggering the next step of the procedure;
8. T1 continues to choose the next event from the Event-Queue, since T1 is attached to P1, so T1 can't choose P3-E1, but it has to choose P1-E2;
9. When T1 handles P1-E2, L could be released;
10. Then T2 could be woken up and acquire L successfully; no dead-lock could happen.

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

But in the period when the thread is attached, it can't handle other procedures' events which need to acquire locks, so this method also constrains the concurrency and throughput of the systems. But comparing with III.A, even after attaching in this method, actually the thread can still handle those events which don't need to acquire locks, so its concurrency and throughput are better than III.A.

C. Dedicate Event-Queues

The above-mentioned two methods focus on ensuring the events of acquiring and releasing locks can be handled in the same thread. But this requirement is too strict. Actually it is only necessary that the event of releasing lock could be handled by some thread, it is not a requirement that the thread must be as same as the one which acquires the lock.

In order to achieve this, this method defines dedicated Event-Queue and Thread-Pool for the events which acquire locks. Still considering the example in section II, because event E1 needs to acquire L, Event-Queue2 and Thread-Pool2 are defined dedicatedly for handling the events which needs to acquire L. As shown in Figure 6, the following procedure is executed:

1. When P1, P2 and P3 are started, all of them send out E1;
2. Because E1 needs to acquire L, so these events are sent to Event-Queue2;
3. T3 obtains P1-E1 to process;
4. When T3 handles P1-E1, it acquires L successfully;
5. When T3 finishes processing P1-E1, it sends P1-E2 for triggering the next step of the procedure;
6. Then T3 continues to choose the next event P2-E1 from the Event-Queue2 and process it;
7. But because L hasn't been released, T3 can't acquire L when it handles P2-E1. So T3 can do nothing but only wait there;
8. Because P1-E2 doesn't need to acquire L, so it is handled in Event-Queue1;
9. T1 chooses P1-E2 to execute;
10. When T1 handles P1-E2, L could be released;
11. Then T3 could be woken up and acquire L successfully; no dead-lock could happen.

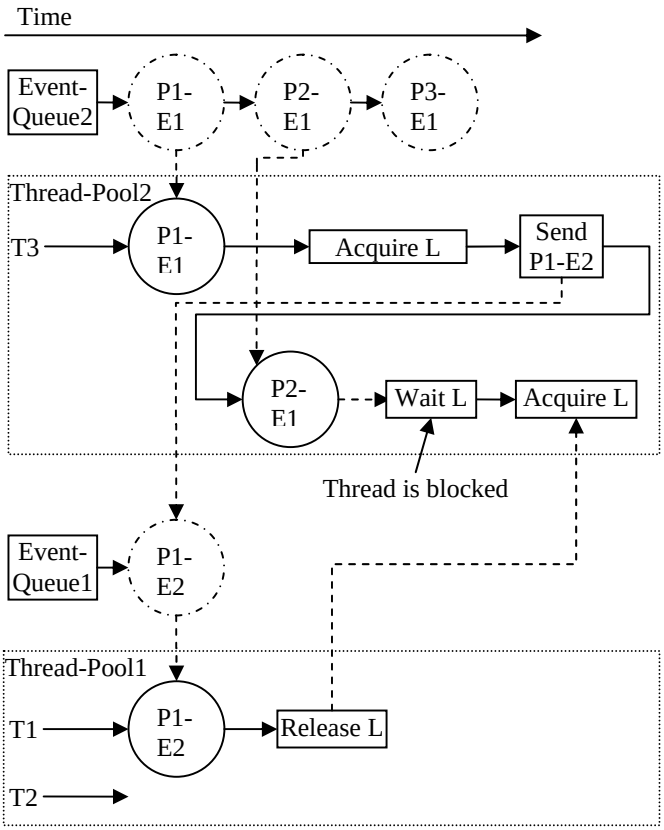

Figure 6 Dedicate Event-Queues

In this way, because all events which acquire L are handled by Thread-Pool2, the threads in Thread-Pool1 are never blocked by L; therefore the events for releasing L can always be handled properly.

Relative to the methods in III.A and III.B, this method takes more memory for defining new Event-Queues and Thread-Pools. The benefit is that it only affects concurrency of handling the events which acquire locks. But the other events are still handled in the original Event-Queue and Thread-Pool, their concurrency and throughput aren't affected. Because the usage of the locks isn't constrained in the procedures, this method also belongs to the method of avoiding dead-lock.

D. Prevent 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 locks can be assigned to an available thread. But if the 2nd point can be resolved, i.e., a thread can handle other events even it is waiting for acquiring a lock (i.e., a thread is never blocked), then the event which releases locks can always be scheduled by an available thread.

In today's popular operating systems, the blocking mechanism of the resources is the basic system synchronization primitive. To make a thread not be blocked, new synchronization primitives must be implemented. When a thread needs to blocking on a lock, the current execution context (stacks and registers) of the thread needs to be kept at somewhere else, and then the thread can go to handle other events. When a lock is released, if there is any waiter on this lock, 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, 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 procedure P1 and P2 acquire lock L in event E1 and release L in E2. There is only one thread in the Thread-Pool. The procedure is illustrated in Figure 7 as following:

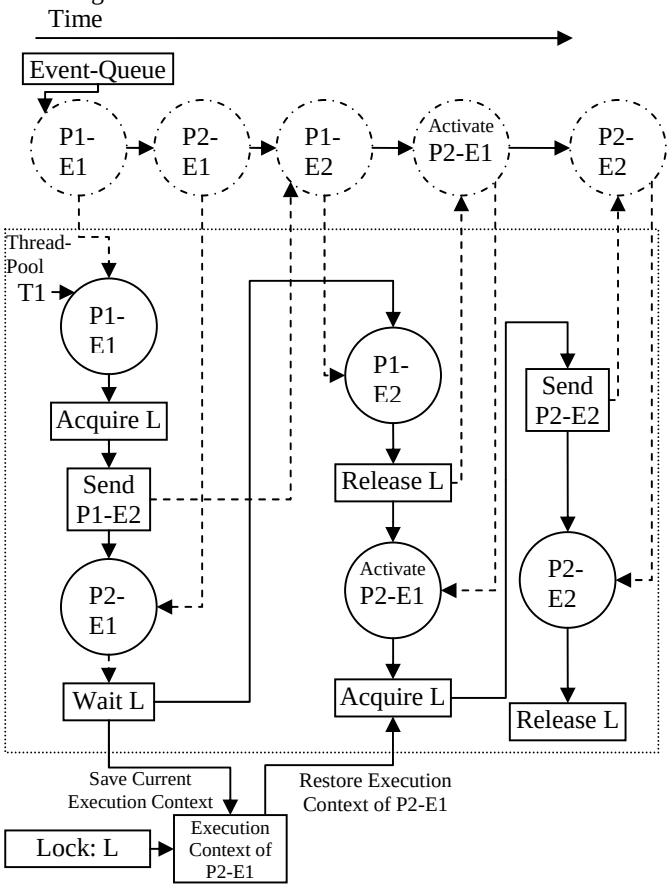

Figure 7 Prevent Thread Blocking

1. When P1, P2 are started, both of them send E1 to Event-Queue;
2. T1 obtains P1-E1 to process;
3. When T1 handles P1-E1, it acquires L successfully;
4. When T1 finishes processing P1-E1, it sends P1-E2 for triggering the next step of the procedure;
5. Then T1 continues to choose the next event P2-E1 from the Event-Queue and process it;

6. Because L hasn't been released, T1 can't acquire L when it handles P2-E1. So T1 can do nothing but only wait there;

7. But T1 isn't really blocked there, it saves the execution context when waiting for L to a list attached to L;

8. Then T1 could continue to handle the next event P1-E2;

9. When T1 handles P1-E2, it releases L;

10. When L is released, it finds there is a waiter (P2-E1) pending there, so it sends a new event Active-P2-E1 to continue blocked P2-E1;

11. Then T1 handles Active-P2-E1, it loads the saved execution context in the list of L, and continue the execution from the blocking point;

12. Now L is in free state, it could be acquired by P2-E1 successfully this time; no dead-lock could happen.

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 events need to acquire locks, or what locks need to be acquired, just like III.B or III.C. 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 locks. 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[8]. 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 MDM 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. They communicate with each other through messages. Therefore, essentially the operation system itself is an implementation of MDM based architecture.

IV. CONCLUSIONS

This paper discusses the dead-lock problem when using Message Driven Model, and promotes four detailed solutions against this problem. The table 1 shows the summary of these solutions:

TABLE 1 COMPARISON OF DIFFERENT SOLUTIONS

Solutions	Complexity	Performance	Applicability
Synchronizing lock release	Low	Low	Low
Attach event handler thread	Middle	Middle	High
Dedicate event-queues	High	High	High
Prevent thread blocking	High	High	High

The first method is very simple, but it does strict limitation on how locks are used, so it can't allow many asynchronous scenario, the performance and applicability are poor. The other three methods require no restriction, so they could be applied to any scenario. The second method attaches threads, this decreases the concurrency, so its performance isn't as good as the others. The third method needs more memory for extra Event-Queues and Thread-Pools. The fourth method needs to implement new operation system primitives. So its implementation is more complex, but they could bring best concurrency and asynchronous performance

Besides the benefit of loosely coupled structure characteristic supported by MDM, this architecture provides the asynchronous event processing ability, which increases the automation and throughput of the system. These features enable MDM model 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 interaction of 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 above could be chosen.

In the meantime, the idea of MDM 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 II); therefore those solutions can be greatly referred for implementing a robust, efficient process or management system.

REFERENCES

[1] Jeff Prosise. 2007. Programming Windows with MFC [M]. Microsoft Press.

[2] Luis Vargas, Jean Bacon, Ken Moody. 2008. Event-Driven Database Information Sharing [J]. Sharing Data, Information and Knowledge. Vol.5071. 113-125.

[3] Levina, O., Stantchev, V. 2009. Realizing Event-Driven SOA [J]. Fourth International Conference on Internet and Web Applications and Services, 2009. 37-42.

[4] Pei, Guanhong, Binoy Ravindran. 2010. "Event-Based System Architecture in Mobile Ad Hoc Networks (MANETs)." Principles and Applications of Distributed Event-Based Systems [M]. IGI Global. 346-368.

[5] K Hauser, HS Sigurdsson, KM Chudoba. 2010. EDSOA: An Event-Driven Service-Oriented Architecture Model For Enterprise Applications [J]. International Journal of Management & Information Systems. Vol 14, No 3.

[6] Tang Zi-ying, Zhe Feng-ping, Tang Xiao-dan. 2000. Computer Operating System [M]. XiAn: XIDIAN UNIVERSITY PRESS.

[7] Gong Yu-chang, Zhang Ye, Li Xi, Chen Xiang-lan. 2008. The Kernel Design of A Novel Component Based Operating System[J]. Journal of Chinese Computer Systems.

[8] Wu Ming-qiao, Chen Xiang-lan, Zhang Ye, Gong Yu-chang. A new operating system construction model based on servant and executive flow [J]. 2006. JOURNAL OF UNIVERSITY OF SCIENCE AND TECHNOLOGY OF CHINA, 2006, 36(2). 230-236.