

A Non-Blocking Buffer Mechanism for Real-Time Event Message Communicatoin

K. H. (Kane) Kim

University of California
Irvine, CA, USA
<http://dream.eng.uci.edu>

Abstract: It is desirable to facilitate data communications among concurrent computation threads without incurring non-essential synchronizations in real-time computing systems. An interaction mechanism, called the *non-blocking writer* (NBW) mechanism and invented by Kopetz, is useful in facilitating state message communication from a producer to a consumer thread in real-time applications. A more widely applicable practical interaction mechanism called the *non-blocking buffer* (NBB) is presented here. The NBB mechanism can be viewed as a significant extension of the NBW mechanism. The NBB mechanism facilitates communication of event messages from a producer to a consumer without causing any party to experience blocking. Therefore, its application scope includes all conceivable producer-consumer situations. The NBB mechanism is not a replacement of but rather a companion to the NBW mechanism since the latter facilitates the most efficient state message communication. The application of NBBs in building middleware supporting real-time objects is discussed as a demonstration of the utility of the NBB mechanism.

Keywords: concurrent programming, distributed computing, thread, synchronization, blocking, real time, NBW, buffer, NBB, monitor, object, TMO, middleware, TMOSM, producer, consumer.

1. Introduction

It has been understood for a long time that the performance of a software system exploiting concurrency is impacted substantially by the approach used for facilitating interactions and synchronizations among concurrent computation threads [Dij65, Bri73, Kop97, Sil02]. In general, it is desirable to make cooperating concurrent threads to interact among themselves without involving non-essential synchronizations. Non-essential synchronizations can be easily found in operating systems (OSs), middleware, and concurrent computing application software of today. They can be found not only in process-structured software systems but also object-oriented software systems. This is largely because the designers did not have the incentives to optimize the performance of the software systems to the extent required in many real-time applications.

Yet as research interests in real-time computing and the rate of developing new real-time computing applications started growing faster in 1990's, the importance of software mechanisms facilitating safe and efficient interactions among concurrent computation threads started receiving a new level of recognition [And97, Ber93, Kim95, Kop89, Kop93, Kop97, Moc99, Pra94, Qaz93, Raj91, Raj95, Zha95]. Non-essential synchronizations are highly undesirable in OSs, middleware, and application software used in many real-time applications. The issue dealt with in this paper is the facilitation of data communications among real-time computation threads without incurring non-essential synchronizations.

An interaction mechanism, called the *non-blocking writer* (NBW) mechanism, was invented by Kopetz [Kop93, Kop97]. It has been found useful in facilitating state message communication from a producer to a consumer thread in real-time applications. Unlike conventional *event messages* each of which must be read by the receiver, the destination of each *state message* is a fixed memory location which is accessed by the receiver in a read-only fashion. The memory location may be called a *state message variable* (SMV). Thus each new state message results in updating the content of the corresponding memory location regardless of whether the reader read the previous content or not. The NBW mechanism

was devised to enable the state message producer / carrier to update the state message variable without experiencing blocking. The reader does not experience blocking either but may experience up to a small number of times its read operations turning into wastes. Therefore, the application of the NBW mechanism was limited to state message communication. For such applications, the NBW mechanism is practical and the most cost-effective approach to the knowledge of this author.

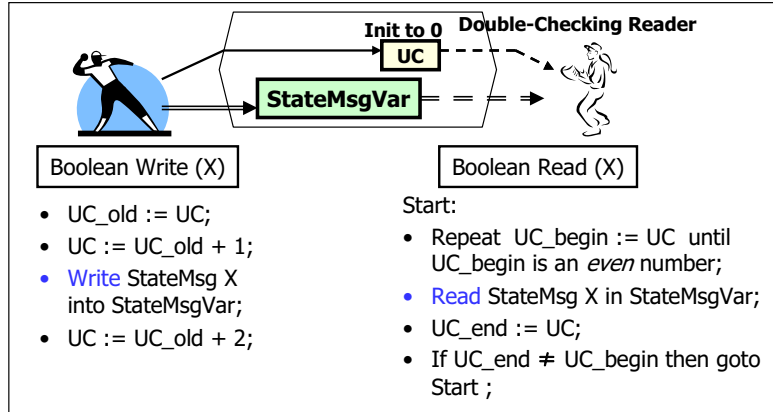


Figure 1. The Non-Blocking Writer (NBW) mechanism (adapted from [Kop97])

In this paper, a more widely applicable practical interaction mechanism called the *non-blocking buffer* (NBB) is presented. The NBB mechanism can be viewed as a major extension of the NBW mechanism. The NBB mechanism facilitates communication of event messages from a producer to a consumer without causing any party to experience blocking. Therefore, its application scope includes all conceivable producer-consumer situations. Experiments involving application of this mechanism in building middleware as well as real-time application software confirmed the utility of the NBB mechanism.

Therefore, the NBB mechanism is not only a major extension of the NBW mechanism but also a great companion to the latter since the latter can be used for state message communication while the former is used for event message communication. It should also be noted that both NBW and NBB mechanisms are interaction mechanisms, more specifically, message communication mechanisms. There are many situations where just efficient synchronizations of threads, not data communication, is required and use of message communication mechanisms for realizing such synchronization is not as efficient as use of established synchronization mechanisms.

In the next section, the NBW mechanism is briefly reviewed and then the NBB mechanism is presented. After discussing the case of a single producer and a single consumer, approaches for adapting the basic NBB mechanism for the cases of more complex interactions are discussed. Then in Section 3, the value of the NBB mechanism in constructing efficient middleware supporting real-time distributed computing objects is discussed. The paper concludes in Section 4.

2. The non-blocking buffer (NBB) mechanism

2.1. The non-blocking writer (NBW) mechanism

The NBW scheme invented by Kopetz is depicted in Figure 1 [Kop93, Kop97]. The *writer thread*, WR, can write a state message into the *state message variable* StateMsgVar at any time without experiencing any blocking. In addition to the state message variable, a counter named the *update counter* (UC) is used. WR increments UC once before updating StateMsgVar and once after updating it. Therefore, UC contains an odd number only during the state-message-write operation and an even number at any other time. Writing a new value into UC must be an *atomic* operation. That is, the value of UC that can be read by the *reader thread*, RD, should always be a valid odd number or a valid even number and never an undefined value. If UC is of a single-word integer type, a basic read or write operation of the counter within a typical commercial computing platform (hardware + OS) is executed as an atomic operation without requiring any special OS mechanism.

When RD reads the state message variable, it follows a protocol that may be called the *double-checking reader* protocol. Basically, after reading the content of StateMsgVar, typically through multiple machine cycles, and before completing the state-message-read procedure, the double-checking RD checks

if WR has updated either UC or UC and StateMsgVar since RD read StateMsgVar recently. If so, the double-checking RD reads StateMsgVar again and performs the check again and this may be repeated until RD finds that its state-message-read has been completed without any interference by the NBW, i.e., WR. The number of read retries is bounded by a number not much larger than two as long as

- (1) the time between state-message-write operations is significantly longer than the duration of a state-message-write or state-message-read operation, and
- (2) RD is not preempted at a disproportionate rate while it is executing the state-message-read procedure.

Therefore, for communication of state messages from WR to RD, the NBW mechanism facilitates the most efficient operation.

2.2. The basic structure and operations of the non-blocking buffer (NBB) mechanism

The NBB scheme is depicted in a highly abstract form in Figure 2. An earlier framework of NBB was discussed in [Kim02a] but since then, the framework has been refined into a concrete optimized mechanism which is presented here. The *producer thread*, PROD, owns the circular buffer. It can write into the buffer at any time without experiencing any blocking and thus is a non-blocking writer of the buffer although this writer is completely different from the writer in the NBW mechanism. In addition to the circular buffer, there are two counters, the *update counter* (UC) and the *acknowledgment counter* (AC), also called the *ack-counter*.

PROD is a non-blocking writer for the update counter. The value of the update counter is interpreted as a pointer to the next slot in the circular buffer to be used for accommodating a new data item.

The *consumer thread*, CONS, is a non-blocking writer for the ack-counter. The value of the ack-counter is interpreted as a pointer to the next slot in the circular buffer to be used for reading a new data item. As will be clarified later, the two counters are used in ways to ensure that PROD and CONS always access different slots in the circular buffer. Therefore, PROD and CONS *never run into collisions in accessing the same buffer slot*.

It turns out that there are practically no significant types of collisions between PROD and CONS in accessing counters either. This is because as long as each counter is of a single-word integer type, a read or write operation of the counter is a trivially short atomic operation, especially in comparison to the time taken for accessing a buffer slot which may be large enough to accommodate a sizable message.

Therefore, the only possible collisions between PROD and CONS occur either when the buffer is empty while CONS tries to read a data item or when the buffer is full while PROD tries to deposit a new data item. These collisions are not due to the nature of interaction mechanisms. They are rather due to our goal of supporting *event message communications*. The NBB mechanism was devised along with the adoption of the software design style of making PROD to exit from NBB when the buffer is already full and then come back later to make a retry. The application designer should ensure that CONS invokes the message-read operation at a frequency which is, on average, higher or equal to that at which PROD invokes the message-insert operation. The purpose of having at least a certain minimal number of message-slots in an NBB is to allow PROD to generate temporary bursts of messages without experiencing any blocking.

The actions taken by PROD to judge whether the buffer saturation remains or not involves one non-blocking read of the ack-counter. Similarly, CONS checks if the buffer remains empty or not and it does so in a manner not disturbing PROD in any way. This means that the NBB mechanism provides a blocking-free event-message buffer between PROD and CONS with practically negligible collision overhead incurred.

Figure 3 depicts the details of the operation algorithms associated with the NBB mechanism. PROD

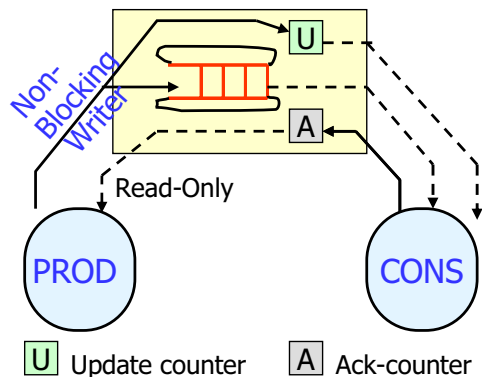


Figure 2. A high-level view of the non-blocking buffer scheme

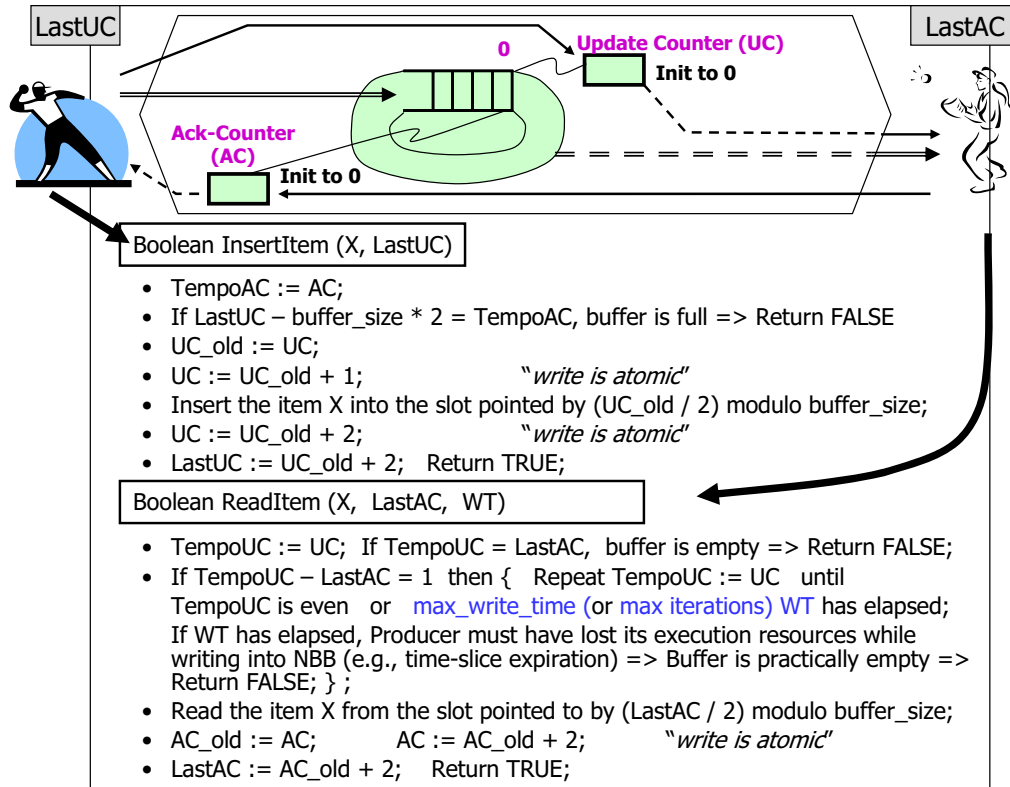


Figure 3. The Non-Blocking Buffer (NBB)

calls operation `InsertItem` to deposit item X into the circular buffer. Between accessing the NBB, PROD maintains a record of the latest value of the update counter which it has produced. By comparing this value, `LastUC`, against the current value of the ack-counter, PROD can tell if the circular buffer is full or not. If the buffer is full, then PROC exits from the operation `InsertItem` and tries the operation later again as the application designer specified.

The designer of a reliable real-time computing application should normally analyze the possible duration of the period of buffer saturation as well as the frequency of saturation occurrences [Kop97].

The storing of a counter value needed in several steps during operation `InsertItem` must be an atomic operation as in the case of the NBW mechanism. As mentioned before, as long as the counter is of a single-word integer type, then the atomicity requirement is met without the aid of any special OS mechanism. Thus the operation `InsertItem` is trivially easy to implement.

Since a circular buffer is used and the relationships between each counter and the corresponding buffer slot are unambiguous, there is no need to have an explicit operation for removing any data item from the buffer. Once the content of a buffer slot is read by CONS, the buffer slot is treated as an empty one until PROD writes a new data item into it.

CONS calls operation `ReadItem` to read item X from a slot in the circular buffer. Between accessing the NBB, CONS maintains a record of the latest value of the ack-counter which it has produced. By comparing this value, `LastAC`, against the current value of the update counter, CONS can tell if the circular buffer is empty or not. If the buffer is empty, then CONS exits from the operation `ReadItem` and try the operation later again as the application designer specified. The designer of a reliable real-time computing application should normally analyze the possible duration of the waiting period of CONS in the face of an empty buffer as well as the frequency of such waiting occurrences.

The first step in operation `ReadItem` is to check the update counter. The following cases may be encountered.

(Case 1) The update counter points to the same buffer slot pointed by LastAC:

In this case the buffer is empty and there is no sign of PROD being in the middle of writing into the buffer. Therefore, CONS exits from the operation ReadItem and tries the operation later again as the application designer specified.

(Case 2) The value of the update counter is an odd number which is only one larger than LastAC:

In this case PROD is in the middle of writing into the buffer slot next to the slot pointed by LastAC. Normally the operation by PROD of inserting a data item into a buffer slot is a quick operation and thus should not exceed the *maximum-write-time* WT, which is a reasonably small number representing the maximum execution time for InsertItem while the buffer is not full. Thus it is worth for CONS to continuously check the update counter until either the update counter becomes an even number, in which case CONS can proceed to access the slot just filled by PROD and read the data item, or WT elapses.

The situation where WT has elapsed while CONS is still checking the update counter arises only when PROD lost the necessary execution resources to another thread before PROD completes the InsertItem operation. In this case, it is not effective for CONS to continuously check the update counter. It is better for CONS to exit from the operation ReadItem and try the operation later again. A frequently occurring situation where PROD loses its execution resources is the expiration of the time-slice given to PROD.

(Case 3) The value of the update counter is larger than LastAC by two or more:

In this case, CONS can just proceed to access the slot pointed by LastAC and read the data item there, regardless of whether PROD is also inside the NBB or not.

Again, implementing operation ReadItem is very easy. Therefore, implementing the NBB mechanism on typical commercial computing platforms is as easy as implementing the NBW mechanism in spite of the greater application scope of the NBB.

In short, the NBB mechanism has the following properties.

Assumption 1: Writing a new value into and reading from each of the two single-word integer counters in the NBB mechanism defined in Figure 2 are atomic operations.

Property 1: If Assumption 1 holds true, the buffer slot into which the producer thread is inserting a data item is always different from the buffer slot from which the consumer thread is reading / copying a data item at the same time.

Property 2: If Assumption 1 holds true, the producer thread never experiences any blocking or looping in accessing the counters and buffer slots due to the actions of the consumer thread.

Property 3: If Assumption 1 holds true, the producer thread detects the full-buffer condition without experiencing any blocking and exits from the NBB mechanism upon such detection.

Property 4: If Assumption 1 holds true and there is at least one fresh data item in the circular buffer, the consumer thread never experiences any blocking or looping in accessing the counters and a buffer slot due to the actions of the producer thread.

Property 5: If Assumption 1 holds true and the time between InsertItem operations is significantly longer than the duration of an InsertItem or ReadItem operation, the maximum execution time of a ReadItem operation by the consumer thread is bounded by maximum-write-time MT plus the time for reading the update counter once and comparing two integers twice.

Proof: Since an InsertItem operation involves an insertion of data item into a buffer slot, a read of each of the two counters, two updates of one counter, and a testing of the buffer saturation condition, MT cannot be smaller than the time for reading a data item from a buffer slot and making two updates of a counter.

2.3. Two-way interactions between two threads via NBBs

If the relationship between two threads is symmetric, i.e., each thread functions as a producer as well as a consumer, two NBBs can be used between the two threads, each providing a one-way event-message

transfer path. Figure 4 illustrates such two-way data paths between processes.

2.4. The case of multiple producers serving a single consumer

The most straightforward approach to using the NBB mechanism to handle this case is to connect multiple producers and the single consumer via NBBs as depicted in Figure 5. Each producer owns a separate NBB connected to the consumer. The consumer reads NBBs in a round-robin fashion. Thus no new blocking possibilities are introduced.

Let us assume that we can use some other mechanism to make multiple producers to take InsertItem actions in such a coordinated manner that only one producer will take an InsertItem action at any given time. Of course, such coordination may accompany occasional blocking or busy-wait looping of some producers. Such coordination can be implemented by using known interaction mechanisms (e.g., shared lock, semaphore, or other mutual exclusion mechanisms) in a number of different ways. Then a version of an NBB that can be shared by multiple producers can be obtained through a very simple revision of the operations depicted in Figure 2. Operation ReadItem needs no change.

Operation InsertItem can be changed into the version in Figure 6. Note that producer threads do not maintain LastUC's, the records of the latest values of the update counter which they have produced. The reasons are as follows. First, it is not possible to let multiple producers maintain LastUC's cooperatively without introducing high overhead. Secondly, the purpose of using LastUC in Figure 3 was to enable the producer to determine without accessing the ack-counter AC, which is owned by the consumer, whether the buffer is full or not. Therefore, the only additional work that the producer in Figure 6 must do before determining whether the buffer is full or not is to access AC. The additional work does not represent a sizable portion of the typical execution time of InsertItem.

2.5. The case of a single producer serving multiple consumers

In this case, use of multiple NBBs is inevitable. First, let us consider the case where each message from the producer thread needs to be multicast to multiple consumer threads. If we use just one NBB to connect the producer to multiple consumers, then there is the problem of maintaining the ack-counter. Even though one can devise a revised complicated procedure for maintaining the ack-counter, such an approach is not as cost-effective as the approach of providing one dedicated NBB between the producer and each consumer.

Next, let us consider the case where the producer needs to send a different message to each different consumer. If we use just one NBB to connect the producer to multiple consumers, then there is the problem of facilitating a non-linear access to buffer slots by the consumers. The resulting complexity does not seem worth it. Again, the approach of providing one dedicated NBB between the producer and each consumer is judged to be more cost-effective.

3. Applications of the NBB mechanism to a middleware model supporting real-time distributed computing objects

3.1. Virtual machines and threads in the TMOSM model

The NBB mechanism is valuable wherever the blocking costs

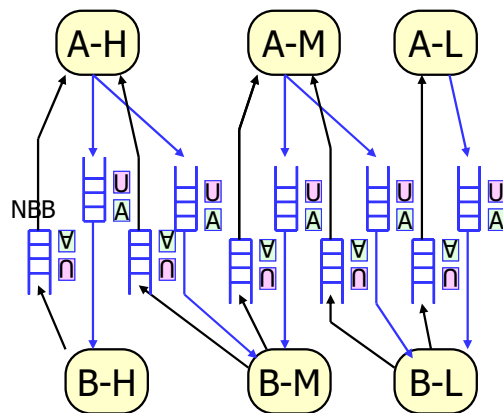


Figure 4. Symmetric connections via NBBs between threads

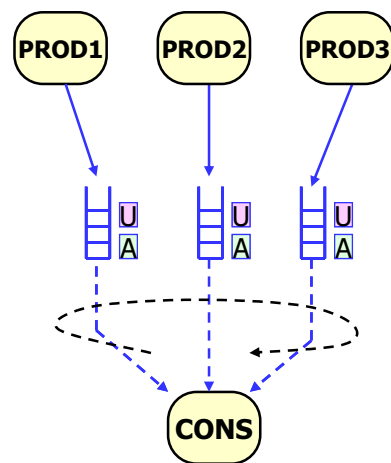


Figure 5. NBB connections to a consumer from multiple producers

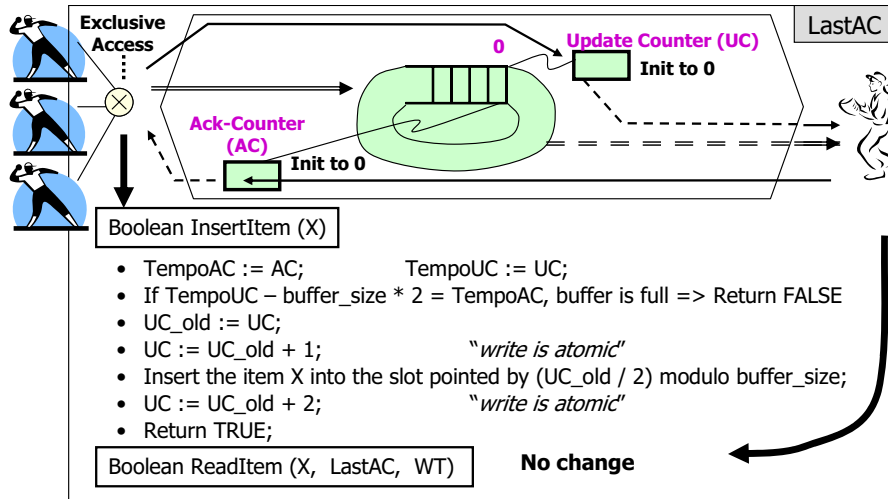


Figure 6. NBB accessed by multiple producers, one at a time

associated with earlier developed interaction mechanisms such as critical sections and semaphore to the applications, are very high. One important application area where we have performed some experiments and the NBB mechanism has turned out to be particularly valuable, is in the middleware supporting real-time distributed computing objects.

One such middleware architecture which we have formulated, experimented with, and discussed in literature in recent years is the TMOSM (TMO support middleware) architecture [Kim99, Kim00]. The real-time object model supported by TMOSM is a high-level real-time distributed computing object model called the Time-triggered Message-triggered Object (TMO) [Kim97, Kim00, Kim02a]. TMOSM has been found to be easily adaptable to most commercial hardware + kernel platforms, e.g., PCs or similar hardware with Windows XP, Windows CE, Linux, etc [Kim99, KimH02].

As depicted in Figure 7, within TMOSM, the innermost core is a super-micro thread called the WTST (*Watchdog Timer & Scheduler Thread*). It is a "super-thread" in that it runs at the highest possible priority level. It is also a "micro-thread" in that it manages the scheduling / activation of all other threads in TMOSM. Even those threads created by the node OS kernel before TMOSM starts are executed only if WTST allocates some time-slices to them. Therefore, WTST is in control of the processor and memory resources with the cooperation of the node OS kernel.

WTST leases processor and memory resources to three *virtual machines* (VMs) in a time-sliced and periodic manner. Each VM can be viewed conceptually as being periodically activated to run for a time-slice. Each VM is responsible for a major part of the functions of TMOSM. Each VM maintains a number of *application threads*. In fact, whenever WTST assigns a time-slice to a VM,

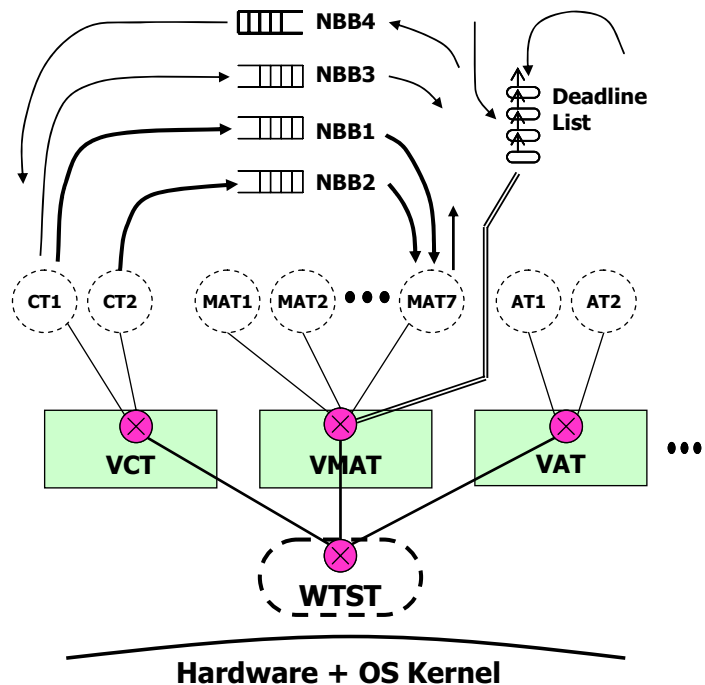


Figure 7. Virtual machines and threads in TMOSM

the VM in turn passes the time-slice onto one of the application threads belonging to itself. The component in each VM that handles this “time-slice relay” is the *application thread scheduler*. For example, VM-A has the application-thread-scheduler VM-A-Scheduler. The application thread scheduler is actually executed by WTST. To be more precise, at the beginning of each time-slice, a timer-interrupt results in WTST being awakened. WTST then determines which VM should get this new time-slice. If VM-A is chosen, WTST executes VM-A-Scheduler and as a result, an application thread belonging to VM-A is activated to run for a time-slice as WTST enters into the event-waiting mode.

The set of VMs is *fixed at the TMOSM start time*. One iteration of the execution of a specified set of VMs is called a TMOSM cycle. For example, one TMOSM cycle may be: VCT VMAT VAT VMAT. The following three VMs handle the core functions:

(1) VCT (*VM for Communication Threads*): The application threads maintained by this VM are those dedicated to handling the sending and receiving of *middleware messages*. Middleware messages are exchanged through the communication network among the middleware instantiations running on different nodes to support interaction among real-time distributed computing objects, i.e., TMOs. Therefore, these application threads are called *communication threads* and denoted as CTs in Figure 7. A communication thread also distributes middleware messages coming through the network to their destination threads, typically belonging to another VM discussed below.

(2) VMAT (*VM for Main Application Threads*): The application threads maintained by this VM are those dedicated to executing methods of TMOs with maximal exploitation of concurrency. Those application threads are called *main application threads* and denoted as MATs in Figure 7. Normally to each execution of a method of an application TMO is dedicated a main application thread. In principle, TMO method executions may proceed concurrently whenever there are no data conflicts among the method executions. In every one of our prototype implementations of TMOSM, the application thread scheduler in VMAT uses a kind of a deadline-driven policy for choosing a main application thread to receive the next time-slice [Kim02b].

(3) VAT (*VM for Auxiliary Threads*): This VM maintains a pool of threads which are called auxiliary threads and denoted as ATs in Figure 7. Some auxiliary threads are designed to be devoted to controlling certain peripherals under orders from TMO methods (executed by main application threads). Others wait for orders for executing certain application program-segments and such orders come from main application threads in execution of TMO methods. Use of this VAT has been motivated partly by the consideration that it should be easier to analyze the temporal predictability of the application computations handled by each VM, i.e., those handled by VMAT and those by VAT, than to analyze the temporal predictability of the application computations when there is no VAT and thus VMAT alone handles the combined set of application computations.

Also, WTST provides the services of checking for any deadline violations and if a violation is found, it provides an exception signal to the user.

We believe that structuring of VMs as periodic VMs is a fundamentally sound approach which leads to easier analysis of the worst-case time behavior of the middleware without incurring any significant performance drawback.

3.2. NBBs facilitating data exchanges between VMs

There are some data variables shared by multiple VMs and more than one of the VMs function as writers. In such cases, the NBB mechanism is not applicable. However, there are many more places in TMOSM where one VM merely supplies data to the other VM. Such a data passing needs to be facilitated by a shared data structure also.

If a well established shared data structure such as the monitor object [Bri73, Sil02] is used to facilitate such a data passing, then the overhead is much greater than in the case of using an NBB. In Figure 7, both CT1 and CT2 in VCT regularly supply data to MAT7 in VMAT. The data passing path from CT1 to MAT7 consists of an NBB, NBB1, and the path from CT2 to MAT7 consists of NBB2 in the figure.

Suppose that all NBBs are replaced by monitor objects, i.e., NBB1 by a monitor object MON1, NBB2

by MON2, etc. Then, between CT1 and MAT7, only one of them can enter MON1 at any given time. The other should be blocked at the entrance to the monitor object. The same situation exists between CT2 and MAT7. Consider the following scenario.

Scenario S1:

- (1) MAT7 is checking the variable inside MON1 while the variable contains no new data item and then the time-slice used by MAT7 (and VMAT) expires before MAT7 exits from MON1.
- (2) Later when VCT and CT1 get a time-slice, CT1 tries to access the monitor object in order to deposit a new data item there. CT1 will be blocked since MAT7 is already inside MON1 and has not released the monitor lock.
- (3) When VMAT and MAT7 get a time-slice again, MAT7 will complete the rest of the monitor procedure for checking the content of the variable and exit from MON1 thereby unlocking it.
- (4) Later when VCT and CT1 get a time-slice again, CT1 will receive a wakeup signal and enter MON1, deposit a data item there, and exit from MON1.
- (5) At the end of this time-slice, VMAT and MAT7 may get a time-slice again and then MAT7 may revisit MON1 and succeed in getting the new data item.

Therefore, CT1 wastes at least one full TMOSM cycle before it succeeds in depositing a new data item into MON1. Especially, CT1 wastes the remainder of the time-slice once it gets blocked in step (2). If CT1 is connected to another monitor object, say MON4, it could have wasted an additional substantial amount of time due to blocking at the entrance to MON4 before coming to step (2) above. If CT1 had to check MON3 to get a new data item before coming to step (2) above in order to prepare a data item to pass on to MAT7, CT1 could have wasted at least two full TMOSM cycles before coming to (2) in the same manner as MAT7 experienced in the above scenario. Also, if MAT7 in step (5) revisits MON1 after a considerable delay such that by that time CT1 has already revisited MON1 with the intention of depositing another data item but it lost its time-slice inside MON1, then MAT7 will have to be blocked at the entrance to MON1.

Obviously, CT2 and MAT7 can experience similar scenarios in accomplishing a data passing through MON2.

The above situation where the time-slice given to MAT7 expires while MAT7 is inside MON1 may be a low-frequency situation but it is not a negligible-frequency situation either. With NBBs used as in Figure 7, CT1 is highly likely to succeed in its first attempt to deposit a new data item into NBB1 since there is no blocking, even if MAT7 is inside NBB1 when its time-slice expires. Then the next time VMAT and MAT7 get a time-slice, MAT7 is highly likely to get the data item. Thus CT1 does not waste any part of its time-slice in a blocked mode.

There is a possible deviation from the above scenario with NBBs. CT1 may be inside NBB1 and its time-slice may expire before depositing a new data item into a buffer slot. Thus at this time both threads, CT1 and MAT7 are inside NBB1 and none of them are active. Then the next time VMAT and MAT7 get a time-slice, MAT7 will exit from NBB1. MAT7 may revisit NBB1 to get a data item before the expiration of its time-slice. MAT7 will find an empty NBB1 again in this case. Therefore, only after VCT and CT1 get another time-slice, a new data item will be deposited into a buffer slot in NBB1 and only after VMAT and MAT7 get another time-slice and revisit NBB1, MAT7 will get the data item. However, CT1 wasted no part of its time-slice. In contrast, CT1 in scenario S1 is blocked at the entrance of MON1 and sits idle for a full time-slice. Even if CT1 had to check NBB3 to get a new data item and prepare a data item to pass on to MAT7 before coming to NBB1 in order to deposit the data item, CT1 does not waste any time due to blocking. This is in sharp contrast to what can happen with the use of monitor objects as discussed above with scenario S1.

Therefore, in the case of implementing TMOSM-like middleware, use of the NBB mechanism is significantly advantageous over the use of blocking-prone interaction mechanisms such as the monitor object.

4. Conclusion

In this paper, the *non-blocking buffer* (NBB) mechanism was presented. It can be viewed as a major extension of the *non-blocking writer* (NBW) mechanism invented by Kopetz, which is useful in facilitating state message communication from a producer to a consumer thread in real-time applications. The NBB mechanism facilitates communication of event messages from a producer to a consumer without causing any party to experience blocking. Therefore, its application scope includes all conceivable producer-consumer situations. The NBB mechanism is not a replacement of but rather a great companion to the NBW mechanism since the latter facilitates the most efficient state message communication. It is also a companion to other established synchronization mechanisms since there are many situations where just efficient synchronizations of threads, not data communication, is required.

The application of NBBs in building middleware supporting real-time distributed computing objects was discussed to point out the compelling nature of the NBB mechanism in such situations. Although some experiments have been conducted with the NBB mechanism, much further experimental studies are needed to obtain better understanding of the potentials of the mechanism. Also, the impacts of using NBBs as interaction mechanisms on resource allocation techniques within operating systems and middleware are considered an important subject for future study.

Acknowledgment: The research work reported here was supported in part by the NSF under Grant Numbers 02-04050 (NGS) and 03-26606 (ITR) and under Cooperative Agreement ANI-0225642 to the University of California, San Diego for "The OptIPuter", and in part by SKKU. No part of this paper represents the views and opinions of any of the sponsors mentioned above.

References

- [And97] Anderson, J.H., Jain, R., and Ramamurthy, S., "Wait-free object sharing schemes for real-time uniprocessors and multiprocessors", *Proc. 18th IEEE Real-Time Systems Symp.*, 1997, pp.111-122.
- [Ber93] Bershad, B., "Practical considerations for non-blocking concurrent objects", *Proc. IEEE Int'l Conf. on Distributed Computing Systems*, 1993, pp. 264–273.
- [Bri73] Brinch Hansen, P., '*Operating System Principles*', Prentice-Hall, Englewood Cliffs, NJ, 1973.
- [Dij65] Disjkstra, E.W., "Communicating Sequential Processes", Tech. Rept. EWD-123, Tech. Univ. of Eindhoven, the Netherlands; published as a chapter in F. Genuys, ed., '*Programming Languages*', Academic Press, London, England, 1968, pp. 43-112.
- [Kim95] Kim, K.H.(Kane), Mori, K., and Nakanishi, H., "Realization of Autonomous Decentralized Computing with the RTO.k Object Structuring Scheme and the HU-DF Inter-Process-Group Communication Scheme", *Proc. ISADS '95 (IEEE Computer Society's 2nd Int'l Symp. on Autonomous Decentralized Systems)*, April 1995, Phoenix, AZ, pp.305-312.
- [Kim97] Kim, K.H., "Object Structures for Real-Time Systems and Simulators", *IEEE Computer*, August 1997, pp.62-70.
- [Kim99] Kim, K.H., Ishida, M., and Liu, J., "An Efficient Middleware Architecture Supporting Time-Triggered Message-Triggered Objects and an NT-based Implementation", *Proc. ISORC '99 (IEEE CS 2nd Int'l Symp. on Object-oriented Real-time distributed Computing)*, May 1999, pp.54-63.
- [Kim00] Kim, K.H., "APIs for Real-Time Distributed Object Programming", *IEEE Computer*, June 2000, pp.72-80.
- [Kim02a] Kim, K.H., "Commanding and Reactive Control of Peripherals in the TMO Programming Scheme", *Proc. ISORC '02 (5th IEEE CS Int'l Symp. on Object-Oriented Real-time Distributed Computing)*, Crystal City, VA, April 2002, pp.448-456.
- [Kim02b] Kim, K.H., and Liu, J.Q., "Going Beyond Deadline-Driven Low-level Scheduling in Distributed Real-Time Computing Systems", in B. Kleinjohann et al. eds., '*Design and Analysis of Distributed Embedded Systems*' (Proc. IFIP 17th World Computer Congress, TC10 Stream, Montreal, Aug 2002), Kluwer, pp.205-215.
- [KimH02] Kim, H.J, et al., "TMO-Linux: A linux-based Real-time Operating Systems Supporting

- Execution of TMOs", *Proc. 5th IEEE CS Int'l Symp. on Object-Oriented Real-time Distributed Computing (ISORC '02)*, Washington D.C., April 2002, pp. 288-294.
- [Kop89] Kopetz, H., and *et al*, "Distributed fault-tolerant real-time systems: the Mars approach", *IEEE Micro*, Vol.9, No.1, 1989, pp.25-40.
- [Kop93] Kopetz, H., and Reisinger, J., "NBW: A Non-Blocking Write Protocol for Task Communication in Real-Time Systems", *Proc. IEEE CS 1993 Real-Time Systems Symp.*, Dec. 1993, pp.131-137.
- [Kop97] Kopetz, H., '*Real-Time Systems: Design Principles for Distributed Embedded Applications*', Kluwer Academic Publishers, ISBN: 0-7923-9894-7, Boston, 1997.
- [Liu73] C.L. Liu and J.W. Layland. "Scheduling algorithms for multiprogramming in a hard real-time environment", *J. ACM*, 20(1):46-61, Jan. 1973, pp.46-61.
- [Moc99] Mock, M.; and Nett, E.; "Real-time communication in autonomous robot systems", *Proc. Fourth Int'l Symp. on Autonomous Decentralized Systems (ISADS)*, March 1999, pp.34 - 41.
- [Pra94] Prakash, S., Lee, Y.-H., and Johnson, T., "A non-blocking algorithm for shared queues using compare-and-swap", *IEEE Trans. on Computers*, Vol.43, No.5, 1994, pp.548-559.
- [Qaz93] Qazi, N.U., Woo, M., and Ghafoor, A., "A synchronization and communication model for distributed multimedia objects", *Proc. first ACM int'l conf. on Multimedia*, Anaheim, CA, Sept. 1993, pp. 147 - 155.
- [Raj91] Rajkumar, R., '*Synchronization in real-time systems - a priority inheritance approach*', *Kluwer Academic Publishers*, 1991.
- [Raj95] Rajkumar, R., Gagliardi, M., and Sha, L., "The real-time publisher/subscriber inter-process communication model for distributed real-time systems: design and implementation", *Proc. Real-Time Technology and Applications Symp.*, May 1995, pp.66 - 75.
- [Sil02] Silberschatz, A., Galvin, P.B., and Gagne, G., '*Operating System Concepts*', 6th ed., John Wiley & Sons, Inc., 2002.
- [Zha90] Zhao, W.; Stankovic, J.A.; and Ramamritham, K.; "A window protocol for transmission of time-constrained messages", *IEEE Trans. on Computers*, Vol.39 , No.9, Sept. 1990, pp.1186 - 1203.