Non-Blocking Buffer between Producer and Consumer Real-Time Threads

Kwang-Hae (Kane) Kim

UCI DREAM Lab
khkim@uci.edu,
http://dream.eng.uci.edu/

* The work presented here includes results of joint work with several collaborators, e.g., Juan Colmenares, Moon-Hae Kim, Kee-Wook Rim, etc.

UCI
DREAM Lab

Major Sources of the Materials Presented

1. Kim, K.H. (Kane), "Basic Program Structures for Avoiding Priority Inversions", *Proc. ISORC 2003 (IEEE CS 6th Int'l Symp. on Object-oriented Real-time distributed Computing)*, Hakodate, Japan, May 2003, pp.26-34.
2. Kim, K.H., "A Non-Blocking Buffer Mechanism for Real-Time Event Message Communication", *Real-Time Systems - The International Journal of Time-Critical Computing Systems*, Vol. 32, No. 3, March 2006, pp. 197 - 211.
3. Kim, K. H. (Kane), Colmenares, Juan A., and Rim, Kee-Wook, "Efficient Adaptations of the Non-Blocking Buffer for Event Message Communication", *Proc. ISORC2007 (10th IEEE CS Int' Symp. on Object & Component Oriented Real-Time Distributed Computing)*, Santorini, Greece, May 7-9, 2007.

* (3) is the most recently refined version.

2

UCI
DREAM Lab

Borrowed from a typical textbook on OS – E.G., Silvershatz & Galvin

Bounded-Buffer Problem

Shared Data

```
type item = ... ;
var buffer array [0..n-1] of item;
in, out:              0..n-1;
counter:              0..n;
in, out, counter    := 0;
```

Producer

```
Repeat
   ...
   produce an item in nextp
   ...
   while counter = n do no-op;
   buffer [in] := nextp;
   in := in + 1 mod n;
   counter := counter +1;
until false;
```

Consumer

```
Repeat
   while counter = 0 do no-op;
   nextc := buffer [out];
   out := out + 1 mod n;
   counter := counter – 1;
   ...
   consume the item in nextc
   ...
until false;
```

* **The statements:**

- ***counter := counter + 1;***
- ***counter := counter - 1;***

must be executed *atomically*.

UCI DREAM Lab

Borrowed from a typical textbook on OS – E.G., Silvershatz & Galvin

Bounded-Buffer Problem

Shared Data

```
type item = ...
var buffer = ...
   full, empty, mutex: semaphore;
   nextp, nextc: item;
   full :=0; empty := n; mutex :=1;
```

Producer

```
repeat
   ...
   produce an item in nextp
   ...
   wait (empty);
   wait (mutex);
   ...
   insert an item in nextc to buffer
   ...
   signal (mutex);
   signal (full);
until false;
```

Consumer

```
repeat
   wait (full)
   wait (mutex);
   ...
   remove an item from buffer to nextc
   ...
   signal (mutex);
   signal (empty);
   ...
   consume the item in nextc
   ...
until false;
```


UCI DREAM Lab

Consequences in RT Concurrent Programming Field

- Implementation of producer-consumer situations in the forms shown in OS textbooks has led to real-time (RT) concurrent programs in which
  - There exists strong timing dependency between producers and consumers

=>

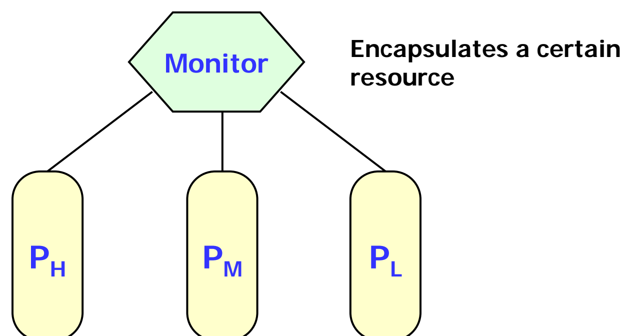
1. Low performance of RT concurrent programs
   - When timing constraints are in scales of milliseconds or sub-milliseconds, safe and efficient low-level message communication mechanisms are of critical importance

2. Timely service capabilities of RT concurrent programs could not be ensured via procedures of step-by-step divide-and-conquer style!

UCI DREAM Lab

Not for EECS223

Also, Some were Concerned with Priority Inversions Caused by Locking Mechanisms

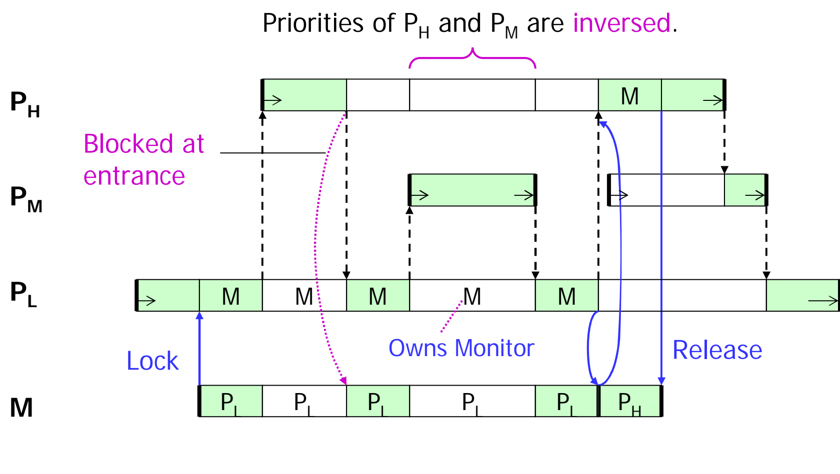


- P_H: High-priority process
- P_M: Medium-priority process
- P_L: Low-priority process

UCI DREAM Lab

Not for EECS223

Nature of the Priority Inversion



Processor switch

Begin & end of a Ready state

Process or Monitor in execution

Monitor locking & release

UCI DREAM Lab

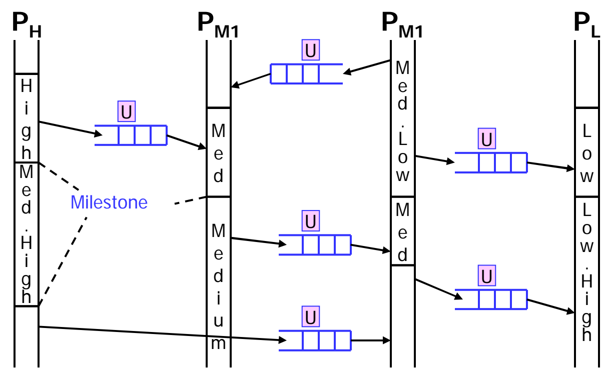
Not for EECS223

Efficient message communication among concurrent computing threads

without causing a thread to be blocked

(i.e., without involving locking)

is highly desirable in real-time computing systems



UCI DREAM Lab

The core idea behind Non-Blocking Buffer (NBB)

Shared Data

```
type item = ... ;
var buffer array [0..n-1] of item;
in, out:             0..n-1;
counter:             0..n;
in, out, counter   := 0;
```

Producer

```
Repeat
    ...
    produce an item in nextp
    ...
    while counter = n do no-op;
    buffer [in] := nextp;
    in := in + 1 mod n;
    counter := counter + 1;
until false;
```

Consumer

```
Repeat
    while counter = 0 do no-op;
    nextc := buffer [out];
    out := out + 1 mod n;
    counter := counter – 1;
    ...
    consume the item in nextc
    ...
until false;
```

1. **Instead of using Counter, let Producer read Out and compute the # of items in buffer by comparing In and Out.**

1. **Instead of using Counter, let Consumer read In and compute the # of items in buffer by comparing In and Out.**

UCI DREAM Lab

9

The core idea behind Non-Blocking Buffer (NBB)

Shared Data

```
type item = ... ;
var buffer array [0..n-1] of item;
in, out:             0..n-1;
counter:             0..n;
in, out, counter   := 0;
```

Producer

```
Repeat
    ...
    produce an item in nextp
    ...
    while counter = n do no-op;
    buffer [in] := nextp;
    in := in + 1 mod n;
    counter := counter + 1;
until false;
```

Consumer

```
Repeat
    while counter = 0 do no-op;
    nextc := buffer [out];
    out := out + 1 mod n;
    counter := counter – 1;
    ...
    consume the item in nextc
    ...
until false;
```


2. **In can be changed atomically without using any special mechanism if it is a single-word integer.** ⇐

```
in_tempo := in + 1 mod n;
in := in_tempo;
```

UCI DREAM Lab

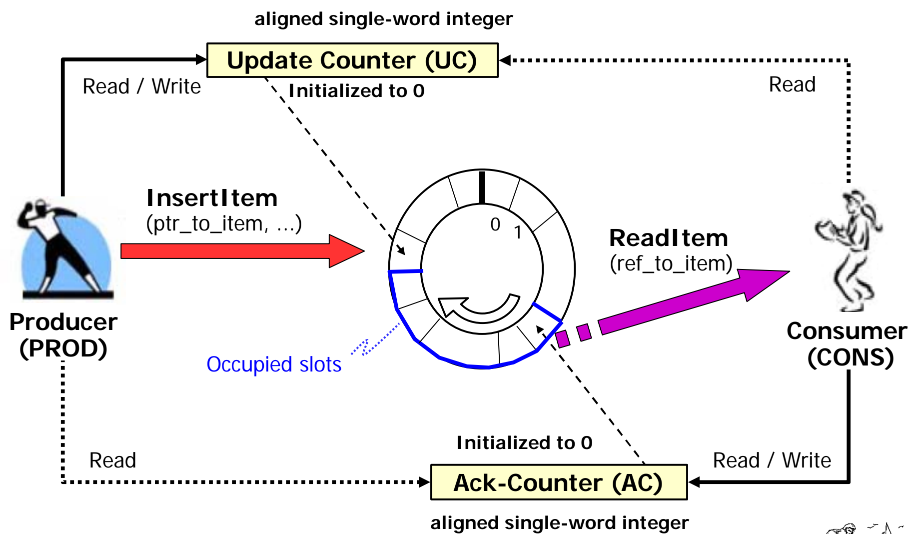
10

Non-Blocking Buffer (NBB)

- Circular FIFO queue
  - Facilitates communication of event messages from a single producer thread (PROD) to a single consumer thread (CONS) without causing any party to experience blocking
- Variation of the traditional circular buffer
- Highly practical and efficient scheme
- Provides two basic functions
  - InsertItem
    - which is called by PROD to deposit an item into the buffer
  - ReadItem
    - called by CONS to obtain an item from the buffer

UCI DREAM Lab

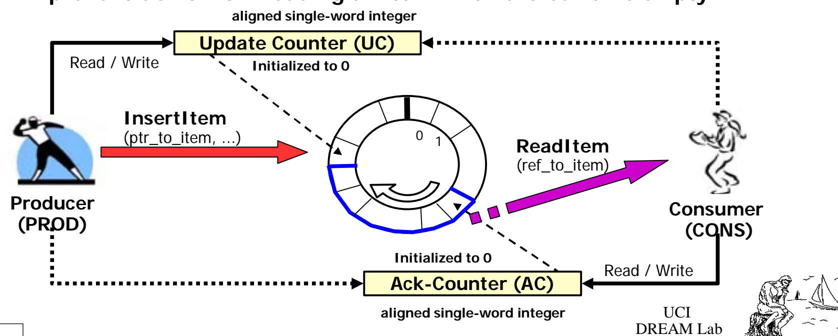
Non-Blocking Buffer (NBB) - Version 1



UCI DREAM Lab

NBB Includes 2 Counters:
Update Counter (UC) and Ack-Counter (AC)

- UC is increased whenever PROD inserts an item
- AC is increased whenever CONS reads an item
- They are used to
  - Ensure that PROD and CONS always access different slots of the circular buffer
  - Prevent PROD from inserting an item when the buffer is full, and prevent CONS from reading an item when the buffer is empty



UCI
DREAM Lab

UC and AC must be of an
Aligned Single-Word Integer Type

- Reading or writing the counters is a trivially short atomic operation
  - PROD is a non-blocking writer for UC
  - CONS is a non-blocking writer for AC
  - Both PROD and CONS are non-blocking readers for UC and AC
- PROD checks if the buffer is full or not without disturbing CONS
- CONS checks if the buffer is empty or not without disturbing PROD

NBB provides a blocking-free event-message buffer between PROD and CONS with practically negligible overhead for checking the status of the buffer

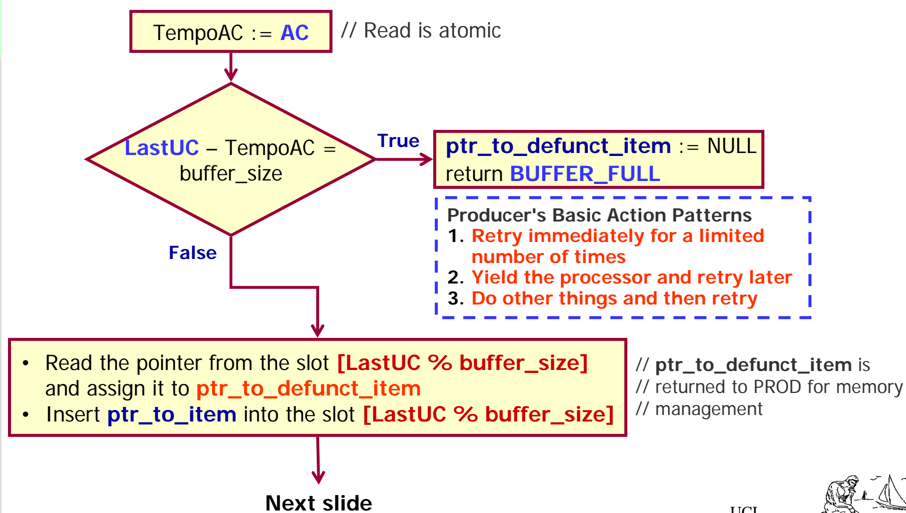
UCI
DREAM Lab

int InsertItem (ptr_to_item, ptr_to_defunct_item)

- Called by PROD to deposit a data item into the buffer
- Parameters
  - ptr_to_item [in]
    - Pointer to the item to be inserted
  - ptr_to_defunct_item [out]
    - Pointer to an item that has been read by CONS and is ready to be disposed of by PROD
      - If the returned pointer is NULL, the function did not find a defunct item in the slot where it inserted the new item
- Returns
  - OK and BUFFER_FULL

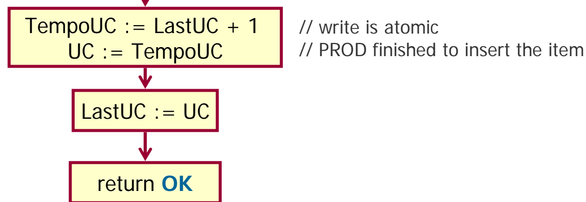
UCI DREAM Lab

int InsertItem (ptr_to_item, ptr_to_defunct_item)



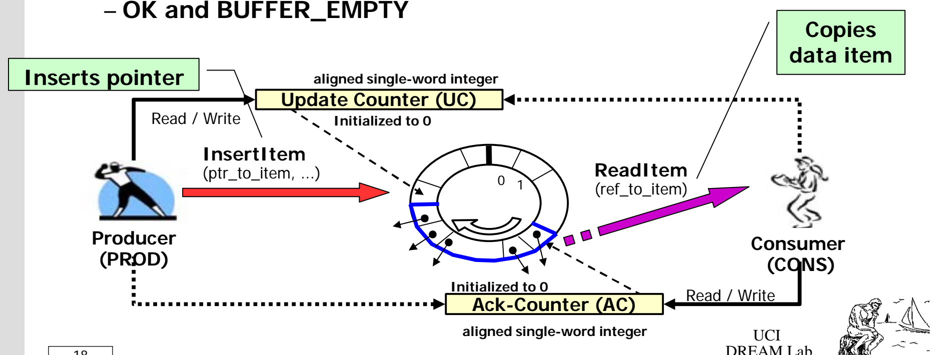
UCI DREAM Lab

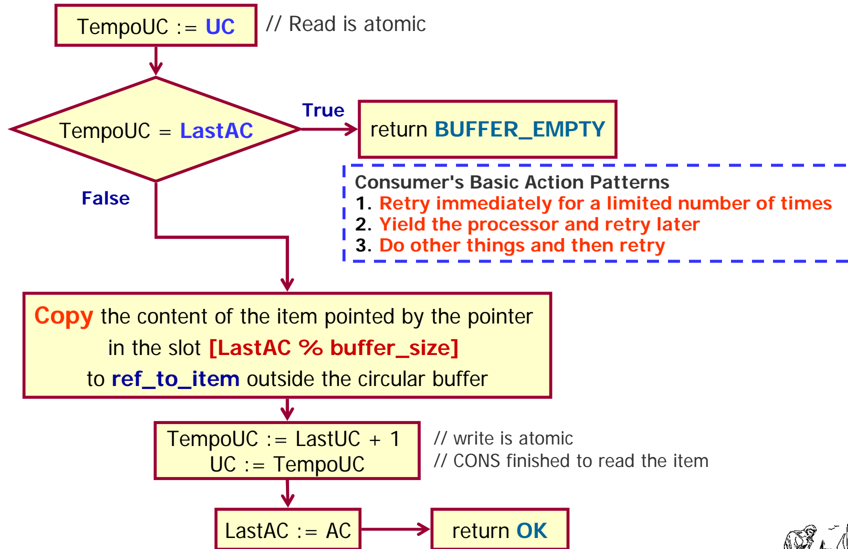
int InsertItem (ptr_to_item, ptr_to_defunct_item)

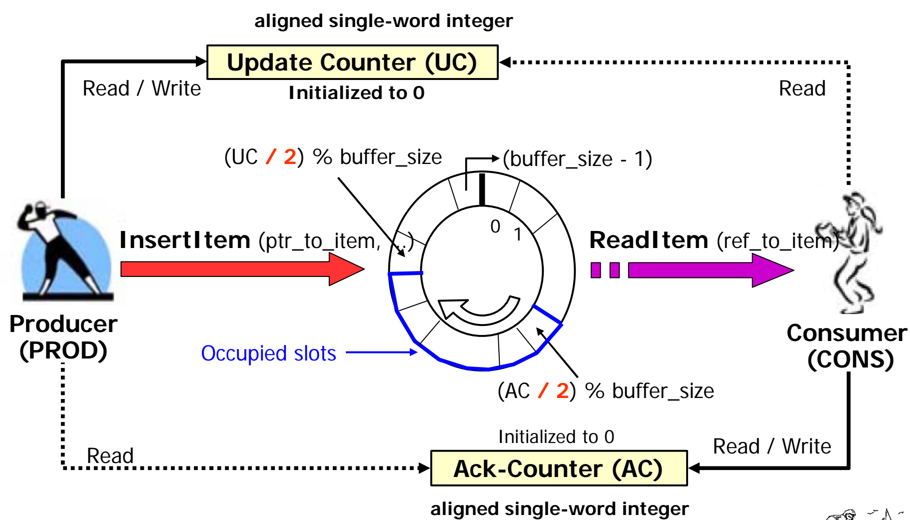


int ReadItem (ref_to_item)

- Called by CONS to obtain an item from the buffer
  - It copies the item pointed by the pointer stored in the NBB into the memory space indicated by CONS
- Parameters
  - ref_to_item [in/out]:
    - reference to the memory space in which the NBB will copy the item
- Return
  - OK and BUFFER_EMPTY




int ReadItem (ref_to_item)
TempoUC := UC
// Read is atomic
TempoUC = LastAC
True
return BUFFER_EMPTY
False
Consumer's Basic Action Patterns
1. Retry immediately for a limited number of times
2. Yield the processor and retry later
3. Do other things and then retry
Copy the content of the item pointed by the pointer
in the slot [LastAC % buffer_size]
to ref_to_item outside the circular buffer
TempoUC := LastUC + 1
UC := TempoUC
// write is atomic
// CONS finished to read the item
LastAC := AC
return OK
19
UCI
DREAM Lab


Non-Blocking Buffer (NBB) – Version 2
aligned single-word integer
Update Counter (UC)
Initialized to 0
Read / Write
Read
(UC / 2) % buffer_size
(buffer_size - 1)
0
1
InsertItem (ptr_to_item, ..)
ReadItem (ref_to_item)
Producer
(PROD)
Consumer
(CONS)
Occupied slots
(AC / 2) % buffer_size
Initialized to 0
Read
Read / Write
Ack-Counter (AC)
aligned single-word integer
20
UCI
DREAM Lab

NBB – Version 2
UC and AC are Increased by 2

- When PROD inserts an item
  - UC is increased by 1 just before PROD accesses the buffer
    - UC is odd → PROD is in the middle of inserting an item
  - UC is increased again immediately after PROD inserts the item
    - UC is even → PROD finished to insert the item
- The function ReadItem can tell CONS that the buffer is empty but PROD is in the middle of inserting an item
  - Depending on the characteristics of the real-time application, CONS may, for example:
    - Keep retrying to read an item for a limited amount of times, hoping that PROD will finish to insert the item by that time
    - Yield the processor to another thread and try again the next time it is scheduled

UCI DREAM Lab

NBB – Version 2
UC and AC are Increased by 2

- AC is increased in similar way as UC
- The function InsertItem can tell PROD that the buffer is full but CONS is about to retrieve an item
  - Since application designers should guarantee that buffer saturation occurs very rarely during normal operation, they may prefer to
    - Ignore that CONS is in activity and
    - Define the actions that PROD will take, considering only the fact that the buffer is full

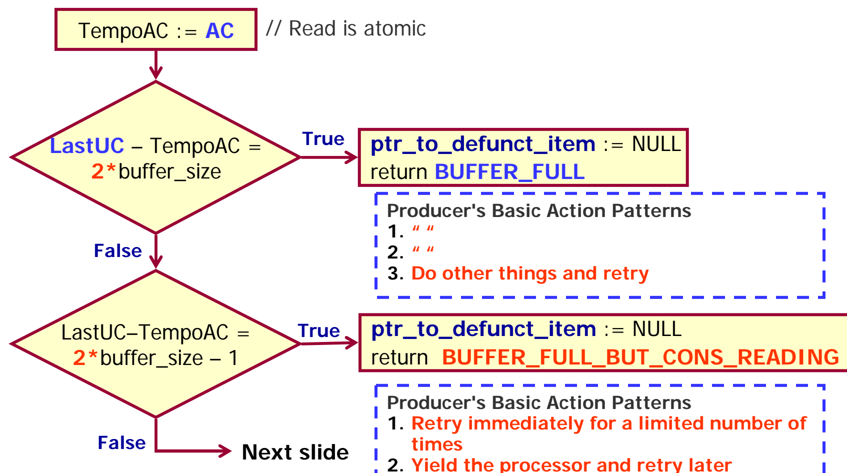
UCI DREAM Lab

int InsertItem (ptr_to_item, ptr_to_defunct_item)

- Called by PROD to deposit a data item into the buffer
- Parameters
  - ptr_to_item [in]
    - Pointer to the item to be inserted
  - ptr_to_defunct_item [out]
    - Pointer to an item that has been read by CONS and is ready to be disposed of by PROD
      - If the returned pointer is NULL, the function did not find a defunct item in the slot where it inserted the new item
- Returns
  - OK, BUFFER_FULL, and BUFFER_FULL_BUT_CONS_READING

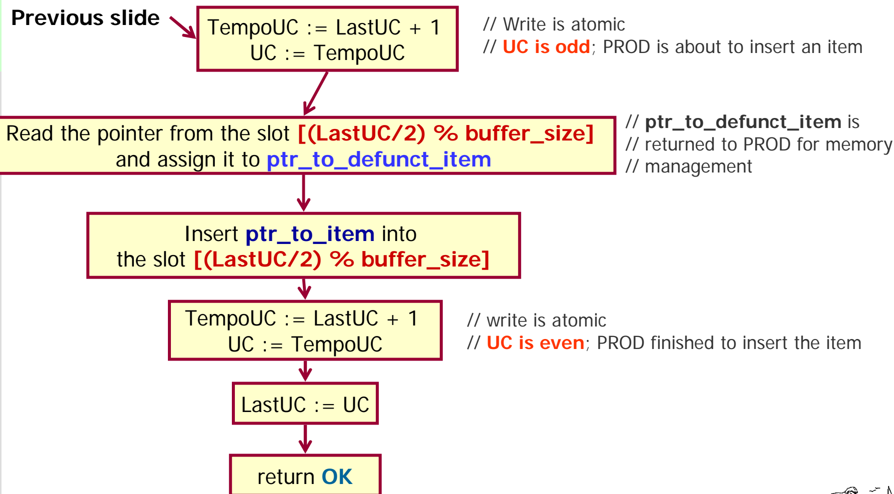
UCI
DREAM Lab

int InsertItem (ptr_to_item, ptr_to_defunct_item)



UCI
DREAM Lab

int InsertItem (ptr_to_item, ptr_to_defunct_item)



int ReadItem (ref_to_item)

- Called by CONS to obtain an item from the buffer
  - It copies the item pointed by the pointer stored in the NBB into the memory space indicated by CONS
- Parameters
  - ref_to_item [in/out]:
    - reference to the memory space in which the NBB will copy the item
- Return
  - OK, BUFFER_EMPTY, and BUFFER_EMPTY_BUT_PROD_INSERTING

int ReadItem (ref_to_item)

TempoUC := UC // Read is atomic

TempoUC = LastAC — True → return BUFFER_EMPTY

Producer's Basic Action Patterns
1. " "
2. " "
3. Do other things and retry

False ↓

TempoUC - LastAC = 1 — True → return BUFFER_EMPTY_BUT_PROD_INSERTING

Consumer's Basic Action Patterns
1. Retry immediately for a limited number of times
2. Yield the processor and retry later

False → Next slide

int ReadItem (ref_to_item)

Previous slide →

TempoAC := LastAC + 1
AC := TempoAC

// Write is atomic
// AC is odd; CONS is about to read an item

Copy the content of the item pointed by the pointer in the slot **[(LastAC/2) % buffer_size]** to **ref_to_item** outside the circular buffer

TempoUC := LastUC + 1
UC := TempoUC

// write is atomic
// AC is even; CONS finished to read the item

LastAC := AC

return OK

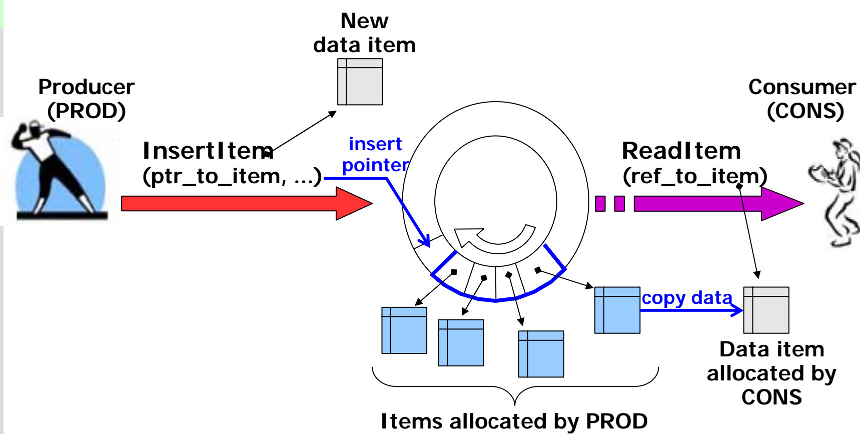
NBB – Version 2
Separate Heap Management

- Heap management shared by concurrent processes or threads cannot be freed of locking operations
- Enabling fully lock-free interaction between PROD and CONS requires that they manage their own heap
- This is why the NBB's functions InsertItem and ReadItem were defined such that
  - PROD inserts pointers to items into the buffer
  - CONS obtains copies of the items in its own memory

UCI DREAM Lab

Not for EECS223

PROD Inserts Pointers to Items and CONS Obtains Copies of the Items



UCI DREAM Lab

Not for EECS223

Defunct Items

Items already copied by CONS

- When PROD inserts a pointer to an item into the NBB, PROD just lends the NBB the pointer
- Once CONS copies the item pointed by that pointer, PROD must get the pointer back so that the memory location associated with the item can be recycled or reclaimed
- NBB must also have mechanisms that allow PROD to obtain the pointers to defunct items

UCI
DREAM Lab

Not for EECS223

Returning Pointers to Defunct Items

- The function InsertItem returns the pointer to the defunct item that was in the slot used to insert the pointer to the new item
  - A minor issue (hardly a problem):

    PROD receives pointers to defunct items only after inserting the first (buffer_size + 1) items
- NBB also includes the function getNextDefunctPointer

UCI
DREAM Lab

Not for EECS223

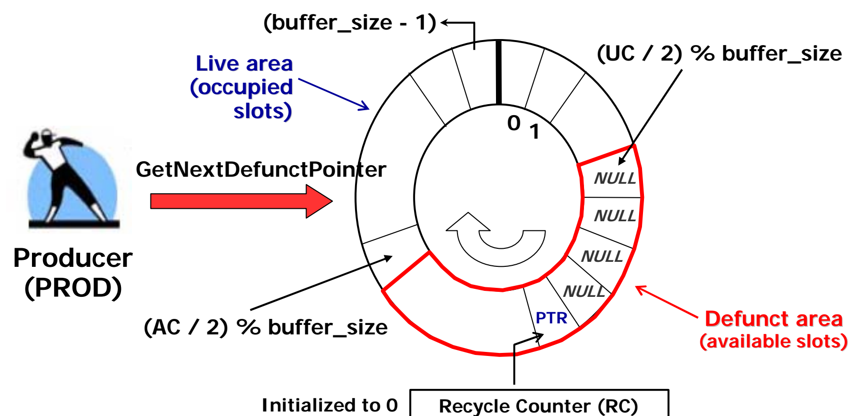
boolean getNextDefunctPointer (ptr_to_defunct_item)

- Called by PROD to obtain immediately a pointer to a defunct item, if such pointer exists in the buffer
- Parameter
  - ptr_to_defunct_item [out]:
    - Pointer to an item that was already copied by CONS and, therefore, is ready to be disposed of by PROD. If the returned pointer is NULL, no defunct item was found
- Returns
  - TRUE if a defunct item was found; FALSE otherwise.

UCI DREAM Lab

Not for EECS223

boolean getNextDefunctPointer (ptr_to_defunct_item)



It always **checks**, and **enforces** if necessary, that **RC points to a valid buffer slot in the defunct area**

UCI DREAM Lab

Two Implementations of NBB
Asymmetric NBB and Symmetric NBB

- **Asymmetric NBB** (the focus of previous slides)
  - PROD inserts pointers to the items, and CONS retrieves copies of the items
  - TMO application programmers are encouraged to use an Asymmetric NBB for non-blocking message communication between
    - A TMO method and a thread running outside VMAT (Virtual machine for Main Application Threads)

      The thread outside VMAT can be:
      - An OS native thread
      - A thread running in VAT (Virtual machine for Auxiliary Threads)

    * Recall that each virtual machine in TMOSM manages its own heap

UCI DREAM Lab

Two Implementations of NBB
Asymmetric NBB and Symmetric NBB

- We can also have a **Symmetric NBB**
  - Here PROD is a TMO method that inserts a copy of an item into the NBB, and CONS is a TMO method that obtains a copy of an item from the NBB

    => Each item is copied twice, when it is inserted and when it is read
    - However, the items are typically pointers
  - TMO application programmers are encouraged to use a Symmetric pointer-based NBB for non-blocking message communication between TMO-methods (e.g., SpMs and SvMs) of a TMO
    - Because both SpMs and SvMs run on the same virtual machine (VMAT, Virtual machine for Main Application Threads) and use the same heap,

      there is no advantage for using an Asymmetric NBB over a symmetric pointer-based NBB.

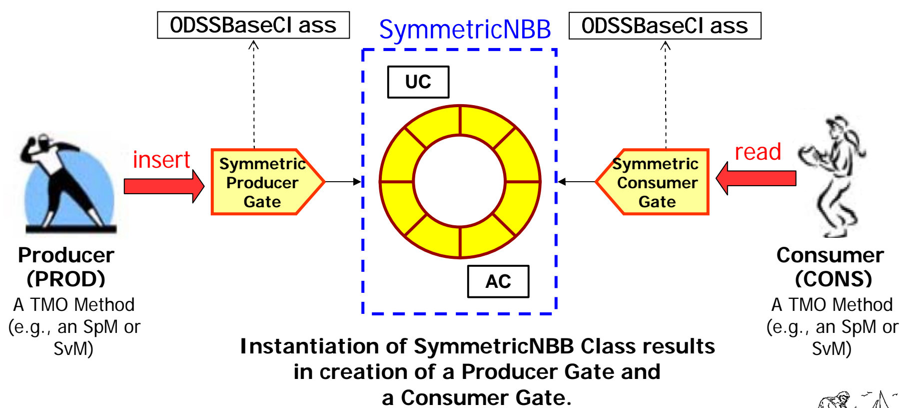
UCI DREAM Lab

Use of Symmetric NBBs between TMO-methods

UCI DREAM Lab

An Approach to Implementation of the Symmetric NBB

- Producer and consumer methods do not access the buffer directly
  - They use the Gates to insert items into and read items from the NBB

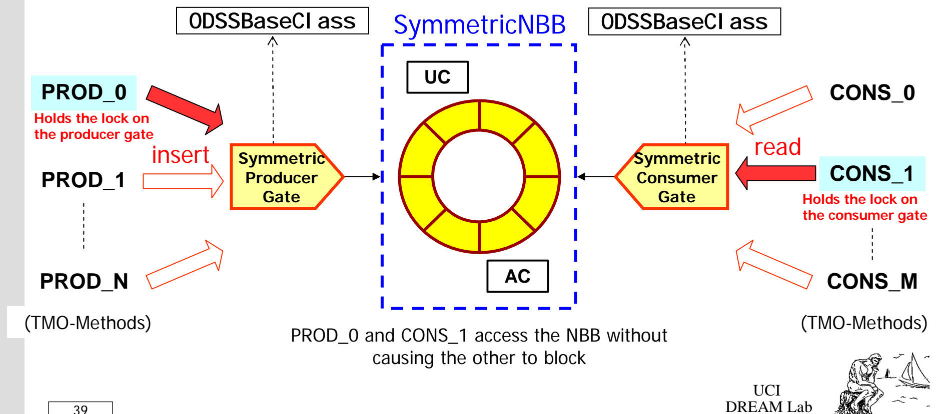


Instantiation of SymmetricNBB Class results in creation of a Producer Gate and a Consumer Gate.

UCI DREAM Lab

Producer & Consumer Gates are ODSSs

- Multiples producers & consumers can use the Symmetric NBB safely
  - A blocking situation among producers only & among consumers only, which is resolved by the ODSS-locks needed for TMO-method executions
  - A producer never causes a consumer to block, and vice versa



PROD_0 and CONS_1 access the NBB without causing the other to block

UCI DREAM Lab

39

An Approach to Implementation of the Symmetric NBB

- The Symmetric NBB consists of 3 classes
  - The SymmetricNBB class
    - A template class that encapsulates the buffer and the counters
  - The SymmetricProducerGate class
    - Access point to the NBB object for a TMO-method(s) that play the producer role
    - It is an ODSS class
  - The SymmetricConsumerGate class
    - Access point to the NBB object for a TMO method(s) that play the consumer role
    - It is an ODSS class
- The producer and consumer gates must be registered to TMOSM with the TMO-methods that intend to use them

UCI DREAM Lab

40

SymmetricNBB<T> Class

- Each item is copied twice, when it is inserted and when it is read
- The type of the items is defined by the template parameter T
  - T can be a primitive type or a type with a default constructor and assignment operator
  - T can also be a pointer type
    - This is particularly useful when copying the items is an expensive operation

UCI DREAM Lab

Constructor and Destructor of the SymmetricNBB<T> Class

- `SymmetricNBB(int size) throw (std::invalid_argument)`
  - **Constructor** that creates a `SymmetricNBB` object with the specified capacity (i.e., number of buffer slots).
  - The parameter `size` indicates the number of buffer-slots.
  - if `size` <= 0 , the exception `std::invalid_argument` is thrown
- `~SymmetricNBB()`
  - The Destructor

```
SymmetricNBB<Message> theNBB(100);
```

Results in creation of

theNBB.producergate and

theNBB.consumergate ,

both of which are ODSSs.

UCI DREAM Lab

Public Attributes of the SymmetricNBB<T> Class

- SymmetricProducerGate<T> producerGate
  - This gate object is used by the TMO-method(s), which play the producer role, to insert items into the NBB
- SymmetricConsumerGate<T> consumerGate
  - This gate object is used by the TMO-method(s), which play the consumer role, to read items from the NBB.

UCI
DREAM Lab

43

SymmetricProducerGate<T> Class

- **Represents the access point to a SymmetricNBB object for TMO-methods that play the producer role**
- **Its constructor is private**
  - **An instance of this class can only be obtained from a SymmetricNBB object**
- **Derived from the ODSSBaseClass**

ODSSBaseClass< SymmetricProducerGate<T> >

SymmetricProducerGate<T>

UCI
DREAM Lab

44

Public Member Functions of the SymmetricProducerGate<T> Class

- `int InsertItem(T item)`
  - Inserts an item into the NBB
  - Parameters

    `item`: the item to be inserted
    - The type of the item (i.e., the template parameter `T`) can be a pointer type, a primitive type, or a user-defined type that implements the default constructor and the assignment operator
  - Return values
    - `NBB_OK`: if the item was inserted successfully
    - `NBB_FULL`: if the NBB is full
    - `NBB_FULL_BUT_CONSUMER_READING`: if the NBB is full but the consumer TMO method is in the middle of reading an item

UCI
DREAM Lab

SymmetricConsumerGate<T> Class

- **Represents the access point to a `SymmetricNBB` object for TMO methods that play the consumer role**
- **Its constructor is private**
  - **An instance of this class can only be obtained from a `SymmetricNBB` object**
- **Derived from the `ODSSBaseClass`**

`ODSSBaseClass< SymmetricConsumerGate<T> >`

`SymmetricConsumerGate<T>`

UCI
DREAM Lab

Public Member Functions of the SymmetricConsumerGate<T> Class

- `int ReadItem(T& rItem)`
  - Reads an item from the NBB and copies its content into an item passed as a reference by the caller TMO consumer method
  - Parameters

    `rItem`: reference to the item that will receive the copy of the content of the item in the NBB that is being read
    - Note that this method modifies the content of the object referenced by this parameter
    - The type of the item (i.e., the template parameter T) can be a pointer type, a primitive type, or a user-defined type that implements the default constructor and the assignment operator
  - Return values
    - `NBB_OK`: if the pointer was read successfully
    - `NBB_EMPTY`: if the NBB is empty
    - `NBB_EMPTY_BUT_PRODUCER_INSERTING`: if the NBB is empty but the producer TMO method is in the middle of inserting an item

UCI DREAM Lab

Example of Use of the SymmetricNBB<T> Class

```
// Message that is communicated between
// producer TMO methods and consumer TMO methods
struct Message {
  // Content of the message
  long long b[1024 * 4];

  Message() {}; // Default constructor

  // Assignment operator
  Message& operator = (const Message& msg) {
    if (this == &msg) // Is the same object?
      return *this;
    int n = sizeof(b) / sizeof(b[0]);
    // In serious programs use memcpy (…) instead
    for (int i = 0; i < n; i++) {
      this->b[i] = msg.b[i];
    }
    return *this;
  }
};
```

File: common.h

UCI DREAM Lab

Example of Use of the SymmetricNBB<T> Class

File: MyTMO2.h

```
. . .
#include "TMOSL.h"
#include "common.h"
using namespace TMO;

// TMO class for testing the SymmetricNBB
class MyTMO2 : public CTMOBase {

public:
    // Maximum number of retries on an NBB operation
    static const int MAX_NUM_RETRIES = 3;

    MyTMO2(int nbbSize);

private:
    void prodSpM1(); // Producer SpM
    void consSpM1(); // Consumer SpM
    SymmetricNBB<Message> theNBB;
    static long long value;
};
```

The NBB automatically includes the **producerGate** and **consumerGate**, which are ODDSs

UCI
DREAM Lab

File: MyTMO2.cpp

```
. . .
void MyTMO2::prodSpM1() {
    Message i;
    i.b[0] = value++;
    int result = theNBB.producerGate.InsertItem(i);
    if (result == NBB_FULL) {
        theNBB.producerGate.ReleaseODSS();
        . . .
    }
    else if (result == NBB_FULL_BUT_CONSUMER_READING) {
        int retries = 0;
        while(retries++ < MAX_NUM_RETRIES) {
            result = theNBB.producerGate.InsertItem(i);
            if (result == NBB_OK) {
                theNBB.producerGate.ReleaseODSS();
                . . .
                return;
            }
        }
        if (retries >= MAX_NUM_RETRIES) {
            TMOSLprintf(_T("PROD says: Retrying was USELESS. \n"));
        }
    }
    else { // result == NBB_OK
        ---  Do application job ---
        theNBB.producerGate.ReleaseODSS(); // Let others use gate
        TMOSLprintf(_T("PROD inserted %d\n"), i.b[0]);
    }
}
```

File: MyTMO2.cpp

```
. . .
void MyTMO2::consSpM1() {
    Message j;
    int result = theNBB.consumerGate.ReadItem(j);
    if (result == NBB_EMPTY) {
        theNBB.consumerGate.ReleaseODSS();
        . . .
    }
    else if (result == NBB_EMPTY_BUT_PRODUCER_INSERTING) {
        int retries = 0;
        while(retries++ < MAX_NUM_RETRIES) {
            result = theNBB.consumerGate.ReadItem(j);
            if (result == NBB_OK) {
                theNBB.consumerGate.ReleaseODSS();
                . . .
                return;
            }
        }
        if (retries >= MAX_NUM_RETRIES) {
            TMOSLprintf(_T("PROD says: Retrying was USELESS. \n"));
        }
    }
    else { // result == NBB_OK
        --- Do application job ---
         theNBB.consumerGate.ReleaseODSS(); // Let others use gate
        TMOSLprintf(_T("CONS read  %d\n"), j.b[0]);
    }
}
```

File: MyTMO2.cpp

```
. . .
MyTMO2::MyTMO2(int nbbSize) : theNBB(nbbSize) {
  // Register producer SpM
  AAC aacOfProdSpM1(. . .);
  SpM_RegistParam paramOfProdSpM1;
  paramOfProdSpM1.build_regist_info_AAC(aacOfProdSpM1);
  paramOfProdSpM1.build_regist_info_ODSS(theNBB.producerGate.GetId(),
                                         RW);
  RegisterSpM((PFSpMBody) &MyTMO2::prodSpM1, &paramOfProdSpM1);

  // Register consumer SpM
  AAC aacOfConsSpM1(. . .);
  SpM_RegistParam paramOfConsSpM1;
  paramOfConsSpM1.build_regist_info_AAC(aacOfConsSpM1);
  paramOfConsSpM1.build_regist_info_ODSS(theNBB.consumerGate.GetId(),
                                         RW);
  RegisterSpM((PFSpMBody) &MyTMO2::consSpM1, &paramOfConsSpM1);

  // Registration parameters of this "MyTMO" instance
  TMO_RegistParam tmoParam;
  _tcscpy_s(tmoParam.global_name,_T("MY_TMO2"));
  tmoParam.start_time = tm4_DCS_age(2*SECOND);;

  // Register this "MyFirstTMO" instance to TMOSM
  RegisterTMO(&tmoParam);
}
```

Example of Use of the SymmetricNBB<T> Class

File: main.cpp

```
#include "TMOSL.h"
#include "MyTMO2.h"

using namespace TMO;

int _tmain(int argc, _TCHAR* argv[]) {
    StartTMOengine ();
    MyTMO2 myTMO2(100);
    MainThrSleep ();
    return 0;
}
```


UCI
DREAM Lab

Related Work

- There are a number of non-blocking comm mechanisms
  - Most of them are for state message communication

    E.G., Non-Blocking Writer (NBW) Protocol by Kopetz & Reisinger (1993)
- Non-Blocking Buffer (NBB) uses a circular buffer
  - Unlike other non-blocking event-message comm mechanisms proposed in literature that use a linked list
- NBB's implementation does not require use of complicated atomic instructions other than simple integer write and integer read operations

UCI
DREAM Lab

Related Work

Comparison between NBB and the Single-reader/Single-writer Version of a Patented Lock-free FIFO Structure

- Recently we learned that an engineer in IBM, India had proposed an approach which is nearly identical in concept but different in practical formulation from NBB
- This approach was patented before the proposal of the NBB
  - Reference: Pradeep Varma, Two lock-free, constant-space, multiple-(impure)-reader, single-writer structures, October 2001, Patent No. US 6304924 B1
  - The patent describes two lock-free, circular FIFO queues that enable highly-concurrent and efficient broadcast communication between a single producer and multiple consumers (1P/*C-broadcast)
  - Simplified versions of these two FIFO queues that only support single-producer/single-consumer (1P/1C) communication are also described in the patent
- NBB is very similar to the 1P/1C version of the lock-free, full-space-utilizing, circular FIFO queue (Scheme 1) patented by Varma
  - Main differences summarized in subsequent slides

Not for EECS223

Related Work

Comparison between NBB and the Single-reader/Single-writer Version of a Patented Lock-free FIFO Structure

- However, NBB differs from this simplified version of the Varma's lock-free, full-space-utilizing, circular FIFO queue (Scheme 1) in several aspects
  - A key difference (?) between them is that NBB does not include a variable dir that indicates the direction of chasing between the producer and the consumer on the buffer array, whereas the Varma's FIFO queue does.
    - Varma's FIFO structure needs to use the variable dir because the counters used to indicate the current positions of PROD and CONS can only take values in [0 , (buffer_size – 1)]
  - Variable dir is a read/write variable shared by PROD and CONS
    - However, this is safe because CONS never surpasses PROD and thus PROD and CONS never modify variable dir concurrently.
  - Instead, in NBB
    - The counters can go beyond the buffer size
    - There is no single read/write variable shared by PROD and CONS

Not for EECS223

Related Work

Comparison between NBB and the Single-reader/Single-writer Version of a Patented Lock-free FIFO Structure

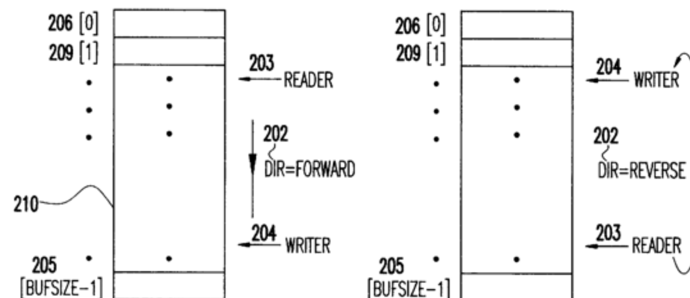
- **Similarities**
  - They both are highly-concurrent and efficient FIFO, constant-space, circular queues that rely on neither synchronization constructs (e.g., locks and monitors) nor machine-specific, special atomic instructions (e.g., `Exchange&Add`, `Test&Set`, and `Compare&Swap`).
  - Instead, they only rely on atomicity of reads and writes of aligned single-word integer values shared by PROD and CONS for bookkeeping purposes.
  - Also, they both make full use of the memory space available in their internal buffer array.

UCI
DREAM Lab

Not for EECS223

Related Work

Comparison between NBB and the Single-reader/Single-writer Version of a Patented Lock-free FIFO Structure



Direction of pointer chase in the circular buffer used in Varma's invention

Note that NBB does not use the DIR variable

UCI
DREAM Lab

Not for EECS223

Related Work

Comparison between NBB and the Single-reader/Single-writer Version of a Patented Lock-free FIFO Structure

- **Key difference between NBB and Varma's lock-free FIFO structure:**

NBB keeps counts of the number of items inserted into and retrieved from the buffer in the update counter (UC) and the ack-counter (AC), respectively.

- **Both counters are of a simple integer type with numerical limits much larger than twice the number of items that can be stored in the buffer, and are allowed to wrap freely.**
- **From |UC-AC|/2 we can always determine the number of items in the buffer, and thereby if the buffer is full or empty.**

UCI DREAM Lab

Not for EECS223

Related Work

Comparison between NBB and the Single-reader/Single-writer Version of a Patented Lock-free FIFO Structure

In contrast, Varma's FIFO structure includes three simple integer variables and atomic operations are performed on them.

- **The first 2 variables are the producer's pointer and consumer's pointer**
  - **During operation the consumer's pointer chases but does not surpass the producer's. The pointers wrap back to the buffer slot zero after reaching the slot (buffer_size – 1).**
- **The 3rd variable is called dir and takes two values:**
  - **FORWARD, if the producer's pointer is ahead of the consumer's pointer**
  - **REVERSE, if the producer's pointer is behind of the consumer's pointer**
  - **The variable dir is initialized to FORWARD**
  - **Every time the producer wraps back to the buffer slot zero dir is set to REVERSE**
  - **Every time the consumer wraps back to the buffer slot zero dir is set to FORWARD**
- **Each time the producer (or the consumer) tries to access the FIFO structure, it reads variable dir and uses it in checking whether the buffer is full (empty) or not.**

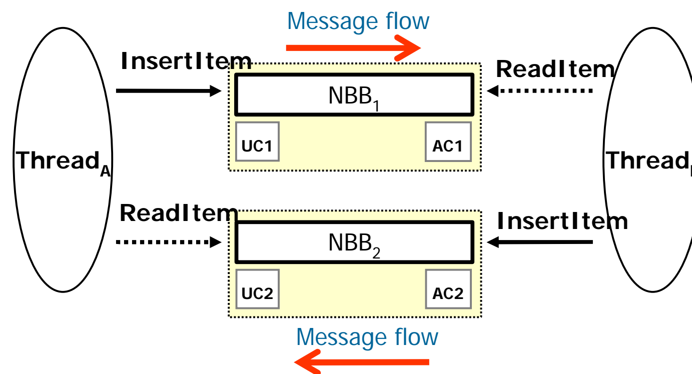
UCI DREAM Lab

NBB-based Approaches for Different Producer-Consumer Scenarios

- Two-way Interaction via NBBs Between Two Threads
- Multiple-Producers/One-Consumer (*P/1C)
- One-Producer/Multiple-Consumers (1P/*C)
  - Unicast Communication (1P/*C-Unicast)
  - Broadcast Communication (1P/*C-Broadcast)
  - Multicast Communication (1P/*C-Multicast)
  - Anycast Communication (1P/*C-Anycast)
- Multiple-Producers/Multiple-Consumers (*P/*C)
  - Unicast Communication (*P/*C-Unicast)
  - Broadcast Communication (*P/*C-Broadcast)
  - Multicast Communication (*P/*C-Multicast)
  - Anycast Communication (*P/*C-Anycast)

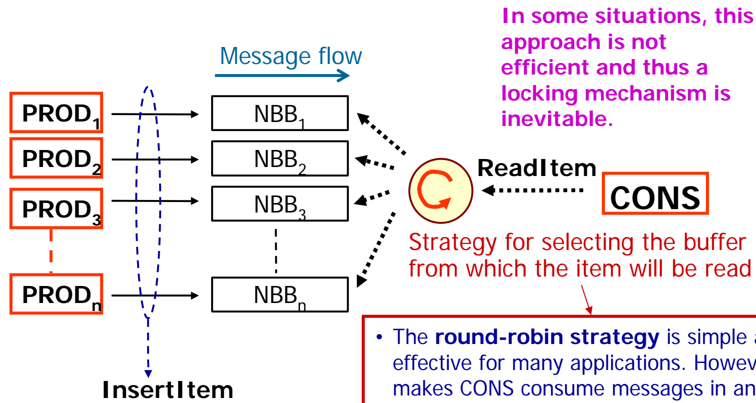
UCI DREAM Lab

Two-way Interaction via NBBs Between Two Threads



UCI DREAM Lab

*P/1C Scenario



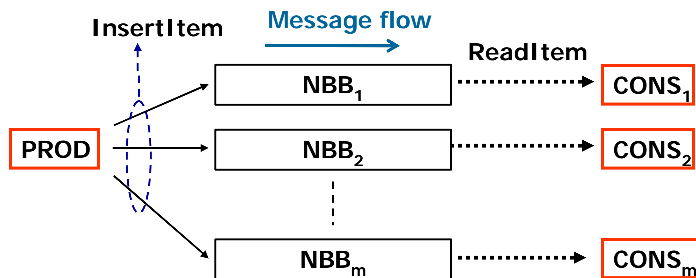
In some situations, this approach is not efficient and thus a locking mechanism is inevitable.

Strategy for selecting the buffer from which the item will be read

- The **round-robin strategy** is simple and effective for many applications. However, it makes CONS consume messages in an order different from that in which they were inserted.
- A more complex strategy needs to be employed if CONS must pick the messages in the order of their ages.

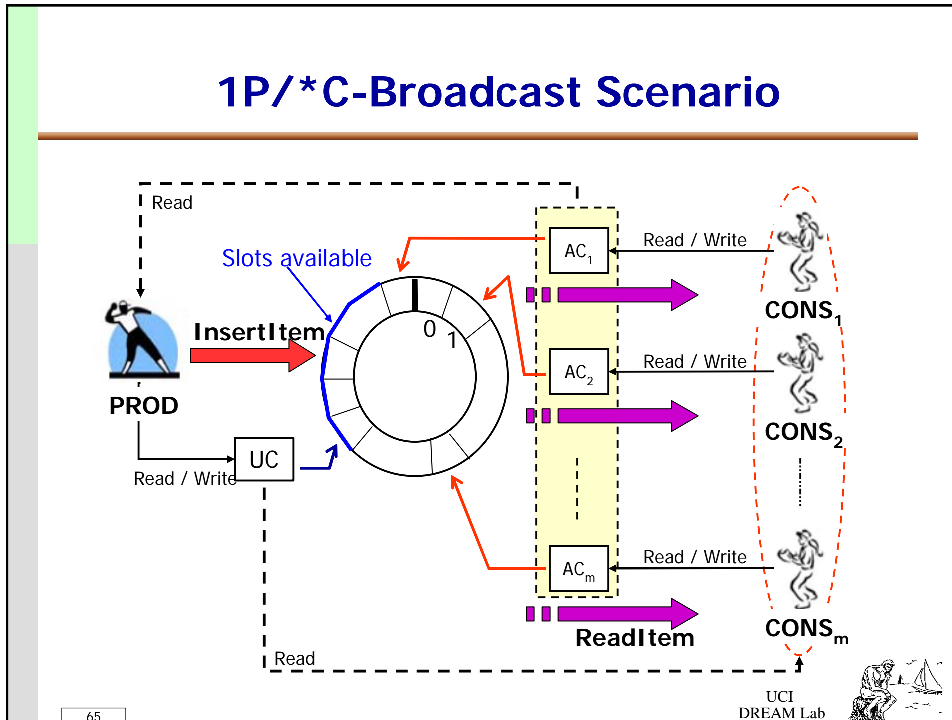
UCI DREAM Lab

1P/*C-Unicast and 1P/*C-Multicast Scenarios



UCI DREAM Lab

1P/*C-Broadcast Scenario

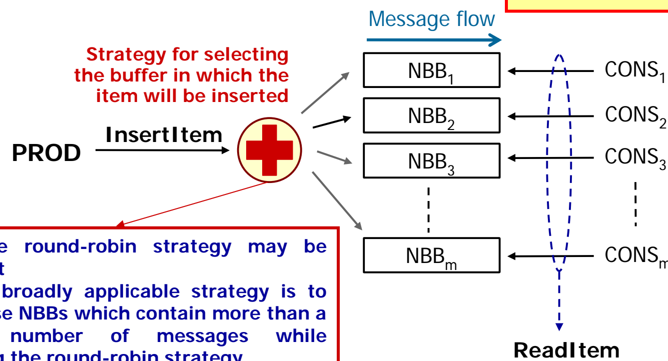


Not for EECS223

1P/*C-Anycast Scenario

- Any one of the consumers can read the next message **exclusively**
- Once a consumer starts reading a message, the message is not available for the rest of consumers

The term **anycast** refers to a different concept than that used in network communication



- A simple round-robin strategy may be sufficient
- A more broadly applicable strategy is to skip those NBBs which contain more than a certain number of messages while following the round-robin strategy

UCI
DREAM Lab

Not for EECS223

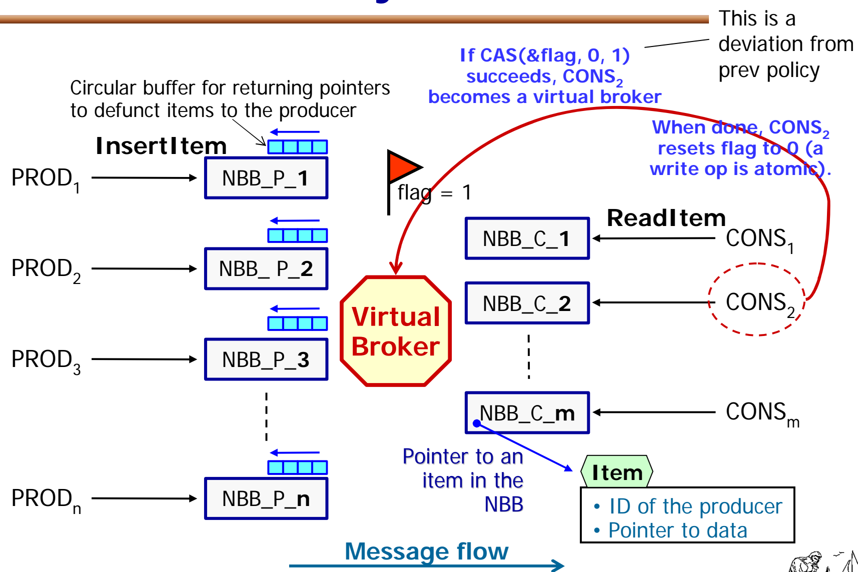
*P/*C Scenarios

- *P/*C-Unicast and *P/*C-Multicast Scenarios
  - We replicate the solution for the 1P/*C-Unicast scenario for each producer
- *P/*C-Broadcast Scenarios
  - We replicate the solution for the solution for the 1P/*C-broadcast for each producer

UCI
DREAM Lab

Not for EECS223

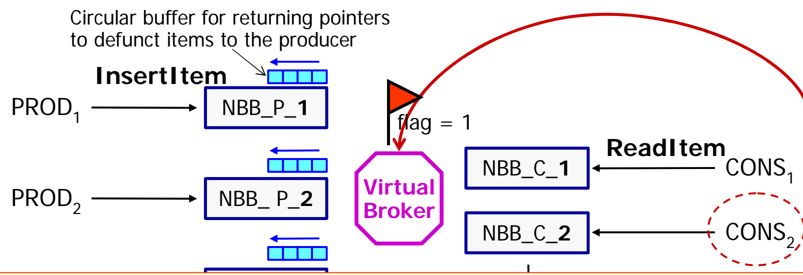
*P/*C-Anycast Scenario



UCI
DREAM Lab

Not for EECS223

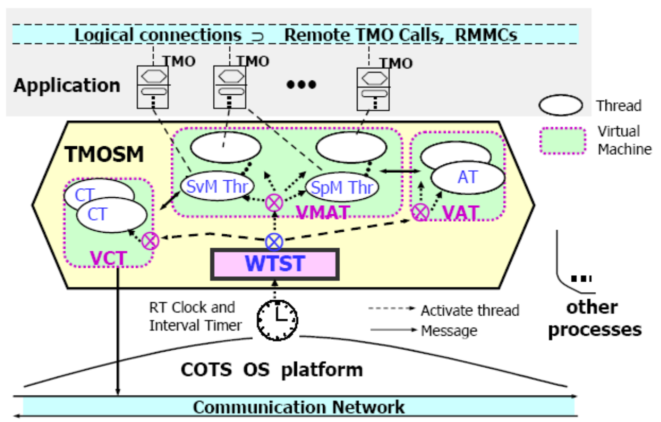
*P/*C-Anycast Scenario



- The variable `flag` is of an aligned single-word integer type.
  - If `flag = 1`, a consumer is currently acting as the broker.
  - If `flag = 0`, no consumer is acting as the broker.
  - The variable `flag` is initialized to zero.
- A consumer becomes the broker only if `CAS(&flag,0,1)` succeeds.
- Once the consumer finishes with the broker's tasks, it simply makes `flag = 0` again.

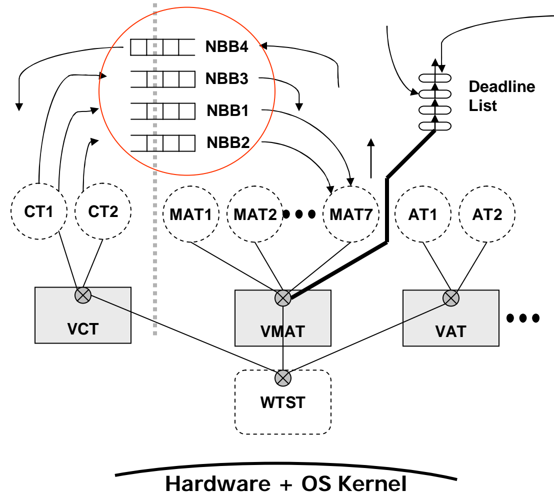
Applications of NBBs to a Middleware Model Supporting RT Distributed Computing Objects

- Time-triggered Message-triggered Object (TMO) Support Middleware



WTST: Watch-dog Timer and Scheduler Thread
VMAT: VM for Main Application Threads
VCT: VM for Communication Threads
VAT: VM for Auxiliary Threads

NBBs inside TMOSM



UCI
DREAM Lab

Conclusion

- NBB enables efficient lock-free event-message communication between a single producer and a single consumer
- NBB has turned out to be particularly valuable in linking real-time threads
  - When all locking message communication mechanisms inside TMOSM were replaced by NBBs, the improvement in the overall performance and the ease of determining execution time bounds was quite significant
- The potential benefit of the NBB mechanism in real-time concurrent programming seems large.

UCI
DREAM Lab