EEL-4736/5737
Principles of Computer System Design

Lecture Slides 9

Textbook Chapter 5

Virtual Links

Introduction

- Communication abstraction within a computer
- Bounded buffer, several threads may send/receive messages
  - Key challenge is to coordinate updates

API

buffer <- ALLOCATE_BOUNDED_BUFFER (*n*)

DEALLOCATE_BOUNDED_BUFFER(*buffer*)

SEND (*buffer*, *message*)

If there is room in buffer, insert message; if not, stop calling thread until there is room

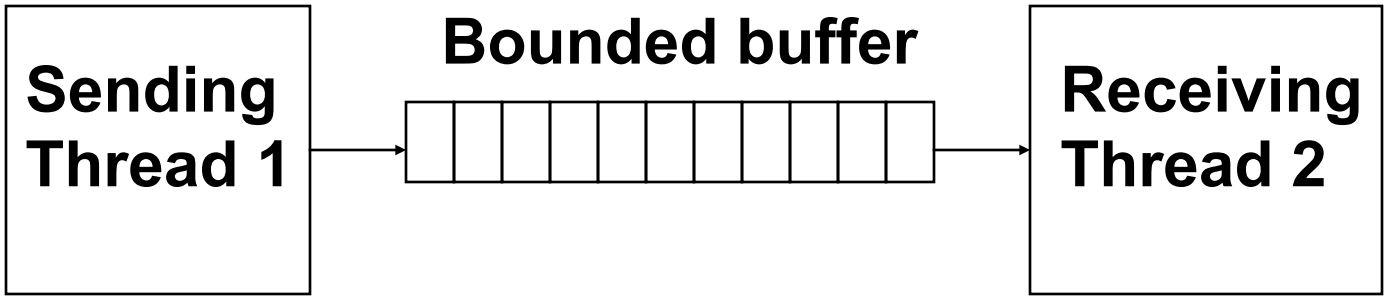
message <- RECEIVE (*buffer*)

If there is a message in buffer, return to calling thread.

If there is no message, stop the calling thread until a message is available

Sequence coordination

- Scenario – assume each thread runs in its own dedicated processor



- Sequence coordination: constraint among threads on precedence of events

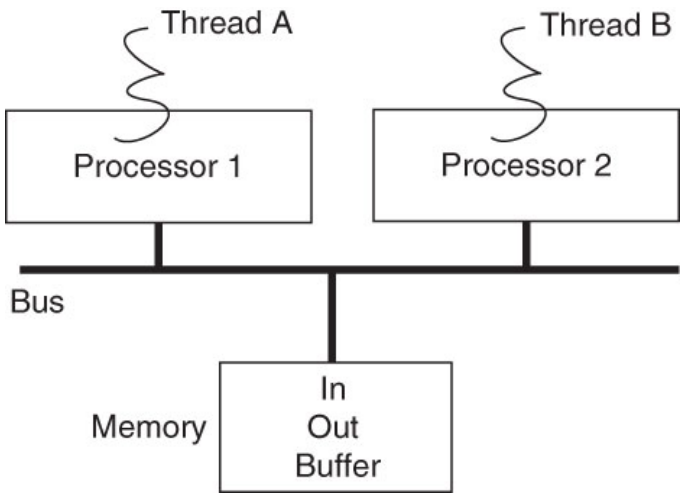
An implementation

1 **shared structure** *buffer*

2 *message* **instance** *message*[N]

3 **integer** *in* **initially** 0

4 **integer** *out* **initially** 0



Buffer needs to be shared by both threads

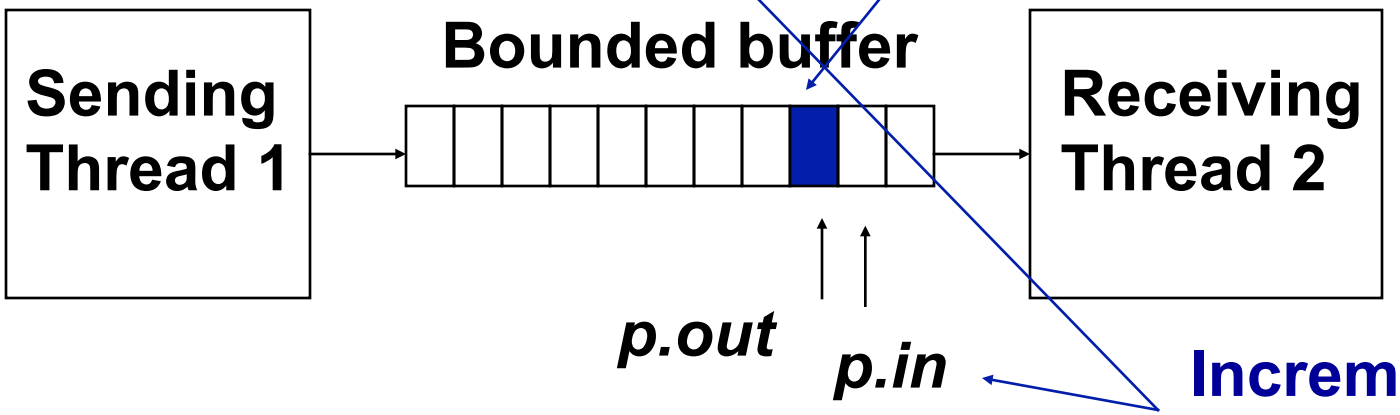
Sender - produces

```
5 procedure SEND (buffer reference p, message
    instance msg)
6 while p.in – p.out = N
        do nothing
7 p.message[p.in modulo N] <- msg
8 p.in <= p.in + 1
```

Buffer full; keep testing (spin loop)

why?

Insert message



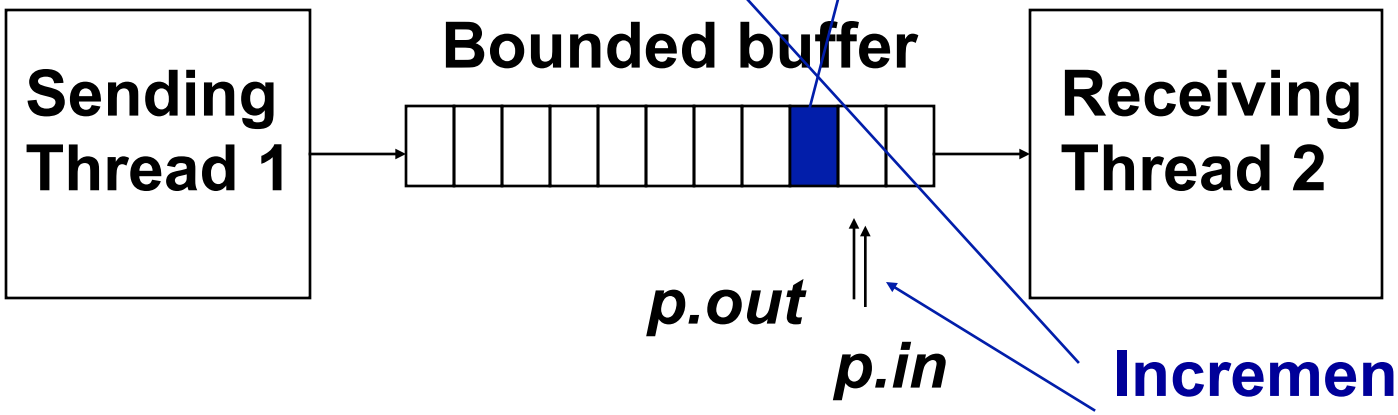
Increment for each New message

Receiver - consumer

```
9  procedure RECEIVE (buffer reference p)
10     while p.in = p.out
11           do nothing
12     msg <- p.message[p.out modulo N]
13     p.out <= p.out + 1
```

No message; keep testing (spin)

Retrieve message



Increment for each Message consumed

Assumptions for correctness

1. There is only one sending and one receiving thread

   p.in and p.out can only be written by one thread (sender and receiver, respectively)

   Buffer message[] only written by sender

- Multiple threads would require a different implementation to coordinate writes
- One-writer principle
  - *If each variable has only one writer, coordination becomes easier*
    - E.g. if buffer message[] was a linked list, pointer updates would require coordination
  - Also improves modularity – one flow of information

Assumptions for correctness

2. Spin loops on both sender and receiver
   – Assume each has dedicated processor
     • Because it blocks the thread
3. *in* and *out* – integers from very large address space so they do not overflow
   – p.in – p.out would become negative
4. Shared memory needs to provide read/write coherence for *in* and *out*
   – Need the most recent value stored by the other thread

Assumptions for correctness

5. Before-or-after atomicity for *in* and *out*
   - If p.in requires multiple writes (e.g. 128-bit variable requires 4 32-bit stores), must ensure receiver does not read a wrong value “in the middle” of a write
6. Result of executing a statement is visible to other threads *in program order*
   - What if update to p.in in sender is seen before message is placed on buffer?
     - Out-of-order processors, optimizing compilers

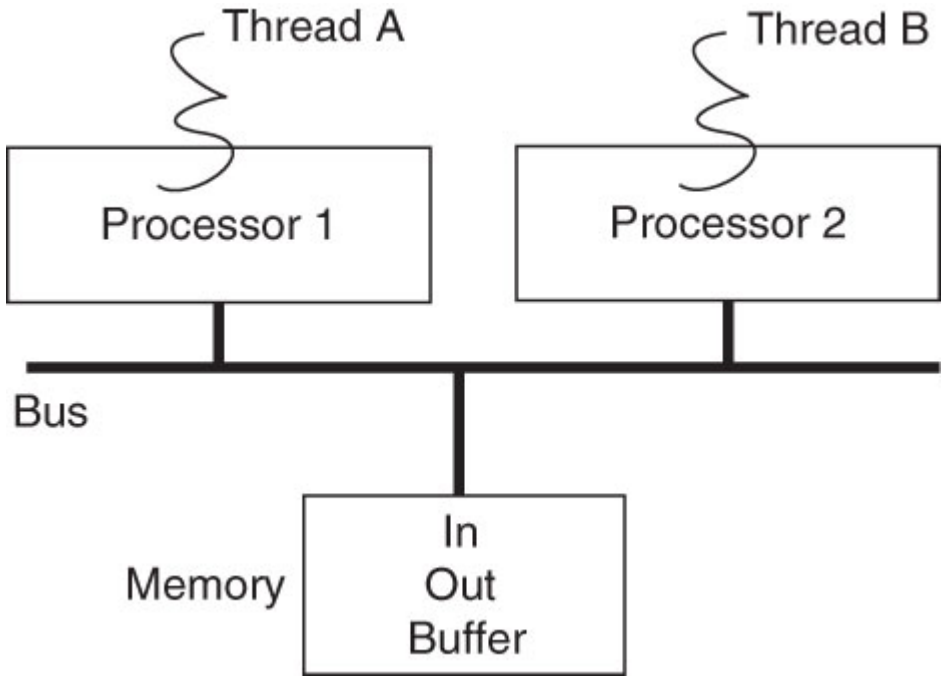
Roadmap

- We will assume 3 (no overflow), 4 (read/write coherence) and 6 (in-order results) hold
  - In practice, can choose very large integers, hardware systems provide read/write coherence and compilers/processors can be told to enforce in-order
- How can we deal with assumptions 1 (one-writer), 5 (before/after atomicity for multi-step memory operations), and 2 (more threads than processors)?
  - Not dealing with these would be too restrictive

Race conditions

- What if assumption #1 does not hold?
  - Allow multiple senders and receivers
- Assume two senders (A and B)
  - Each running in its own processor
  - Assume N=20, all entries are empty
- Asynchronous interpreters
  - While order of execution within threads A and B is sequential, the overlapping of their instructions cannot be determined

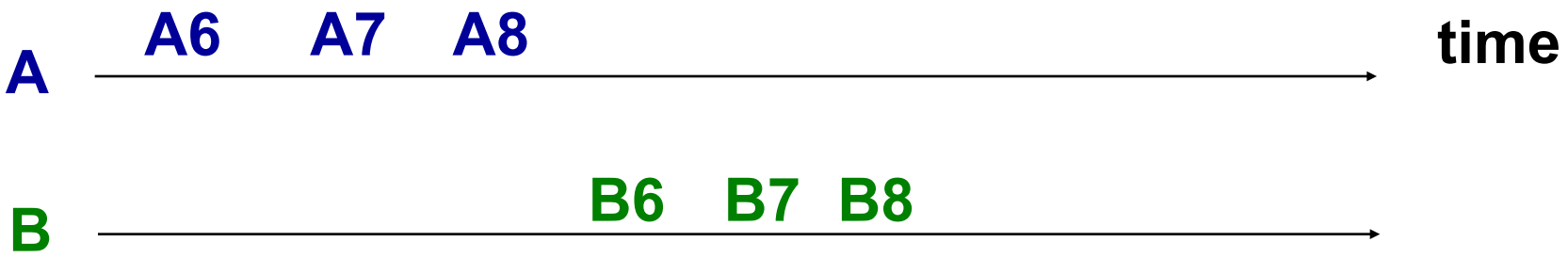
Model



A, B – asynchronous interpreters
Assume no memory caching, no reordering

Example – correct behavior

```
5 procedure SEND (buffer reference p, message
    instance msg)
6 while p.in – p.out = N
        do nothing
7 p.message[p.in modulo N] <- msg
8 p.in <= p.in + 1
```



Example – incorrect behavior

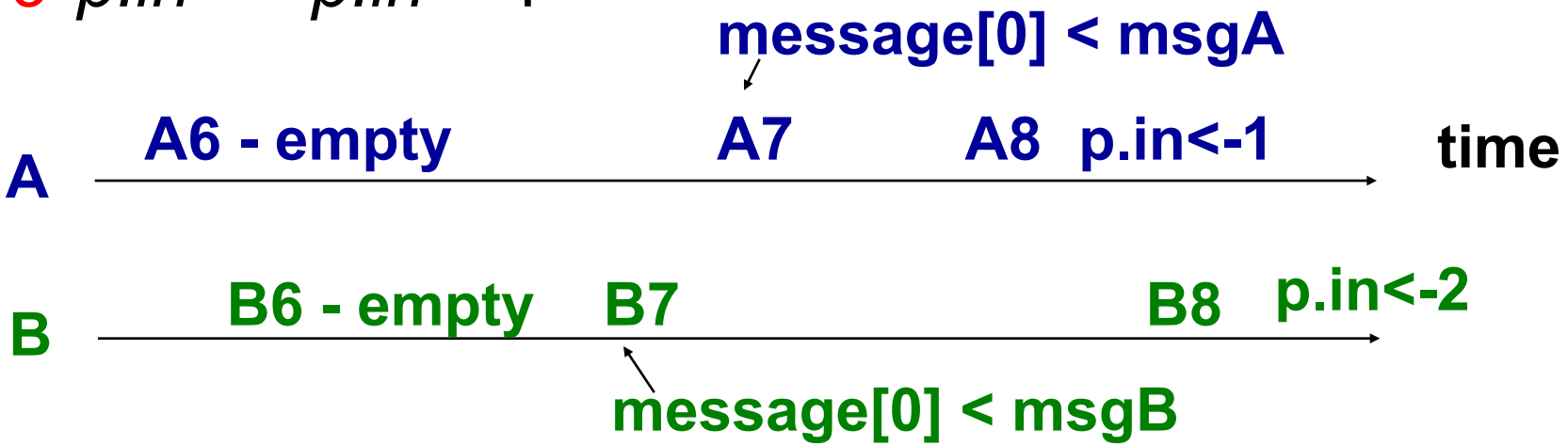
5 **procedure** SEND (*buffer* **reference** *p*, *message* **instance** *msg*)

6 **while** *p.in* – *p.out* = N

do nothing

7 *p.message*[*p.in* **modulo** N] <- *msg*

8 *p.in* <= *p.in* + 1



Race condition

- Outcome depends on the timing of threads
  - Whether error happens or not cannot be controlled
  - Error case can happen infrequently
    - Errors can be masked for a long time
  - Difficult to reproduce deterministically
  - Debugging – major challenge
- A major pitfall in developing concurrent programs

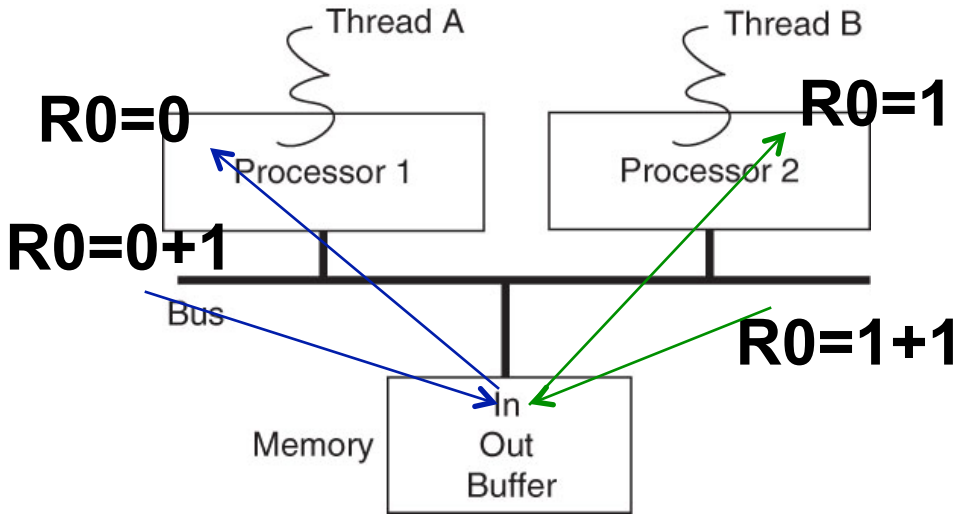
Unfolding into machine code

8 in <- in + 1

1 LOAD in, R0

2 ADD R0, 1

3 STORE R0, in



A **1 – read 0 2 3 in <- 1** **time**

B **1 – read 1 2 3 in <- 2**

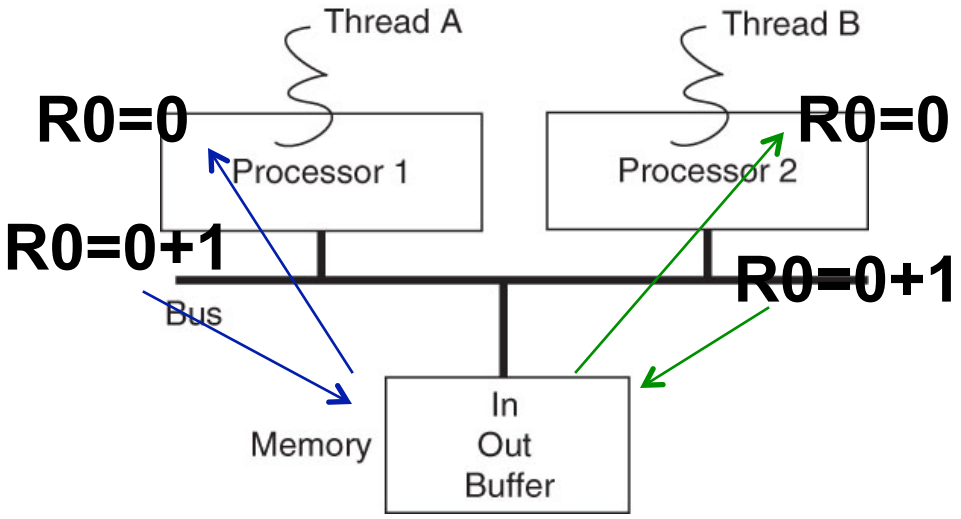
Unfolding into machine code

8 in <- in + 1

1 LOAD in, R0

2 ADD R0, 1

3 STORE R0, in



A **1 – read 0** **2** **3 in <- 1** **time**

B **1 – read 0** **2** **3** **in <- 1**

Multiple senders cause this race condition

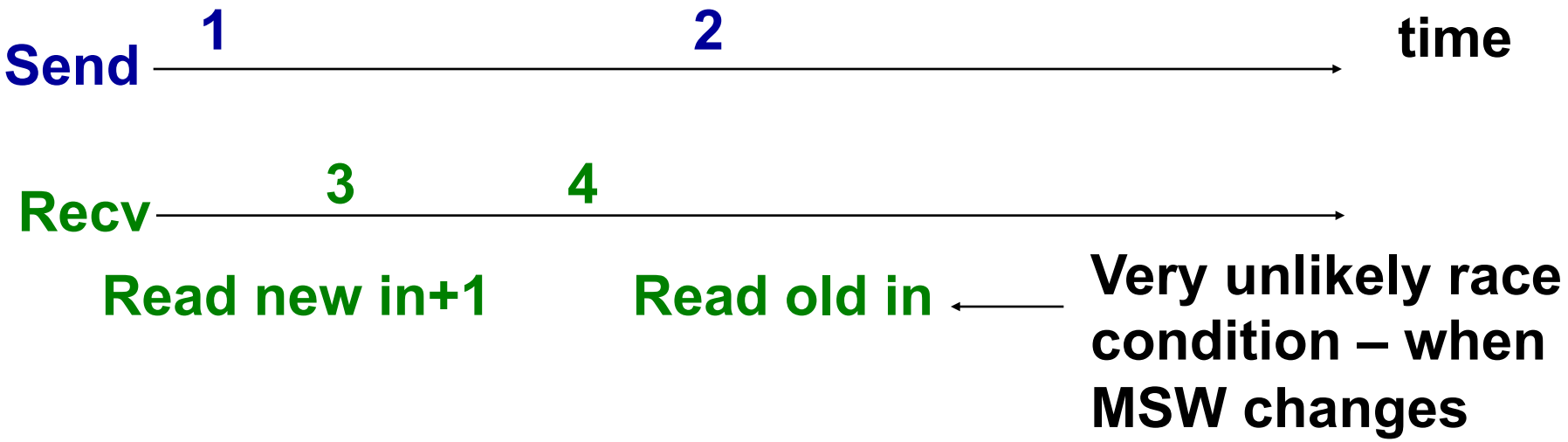
Single sender with no atomicity

1 STORE R0, in+1 (least-significant word)

2 STORE R1, in (most significant word)

3 LOAD in+1, R0

4 LOAD in, R1



Locks – before/after actions

- A “lock” is a widely-used mechanism to coordinate sequence among threads
  - Lock: a shared variable (typically in memory) used explicitly for sequence coordination of access to other variables
- Primitives:
  - ACQUIRE (lockname)
    - “Grab” lock and proceed when lock is available
    - Block if unavailable
  - RELEASE (lockname)
    - Make lock available

Implementation with locks

1 **shared structure** *buffer*

2 *message* **instance** *message*[N]

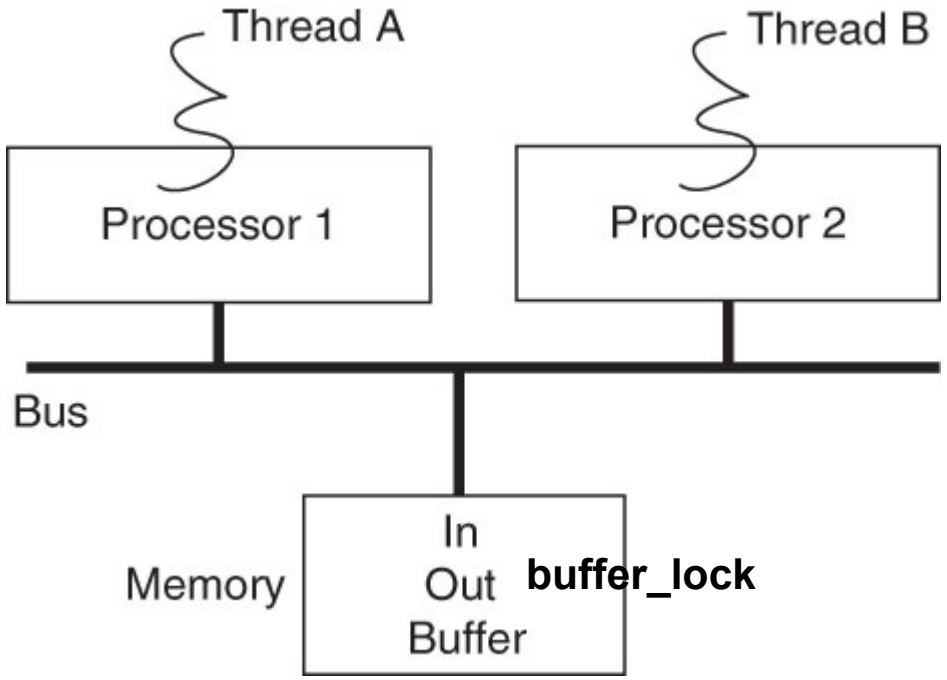
3 **integer** *in* **initially** 0

4 **integer** *out* **initially** 0

5 *lock* **instance** *buffer_lock* **initially** UNLOCKED

Single lock variable for the shared buffer

Model



buffer_lock – a variable also in memory

Sender with locks

```
6  procedure SEND (buffer reference p,
     message instance msg)
7  ACQUIRE (p.buffer_lock)
8  while p.in – p.out = N do
9        RELEASE (p.buffer_lock)
10       ACQUIRE (p.buffer_lock)
11 p.message[p.in modulo N] <- msg
12 p.in <= p.in + 1
13 RELEASE (p.buffer_lock)
```

Assume buffer not full

6 **procedure** SEND (*buffer* **reference** *p*, *message* **instance** *msg*)

7 ACQUIRE (*p.buffer_lock*)

11 *p.message*[*p.in* **modulo** N] <- *msg*

12 *p.in* <= *p.in* + 1

13 RELEASE (*p.buffer_lock*)

Must acquire lock before updating message and p.in

Dealing with full buffer

8 **while** *p.in* – *p.out* = N **do**

9 RELEASE (*p.buffer_lock*)

10 ACQUIRE (*p.buffer_lock*)

Need to release lock to allow other thread to make progress (RECEIVE)

Need to acquire again before testing for buffer full condition

Lock operations

- Operations ‘sandwiched’ between acquire and release are in effect executed as an indivisible action
  - Multiple operations; multi-step writes
- If properly programmed, enforces the single-writer assumption
  - There could be many writers, but only one holding the lock at any given time
- Note: locks do not physically prevent writes
  - As in procedure calls, it’s a convention - programming errors can bypass the lock’s intent

Using locks

- Where to insert locks?
  - Not always simple to determine
  - Improper placement may still allow race conditions

Example

6 **procedure** SEND (*buffer* **reference** *p*, *message* **instance** *msg*)

7 **while** *p.in* – *p.out* = N **do nothing**

8 ACQUIRE (*p.buffer_lock*)

9 *p.message*[*p.in* **modulo** N] <- *msg*

10 *p.in* <= *p.in* + 1

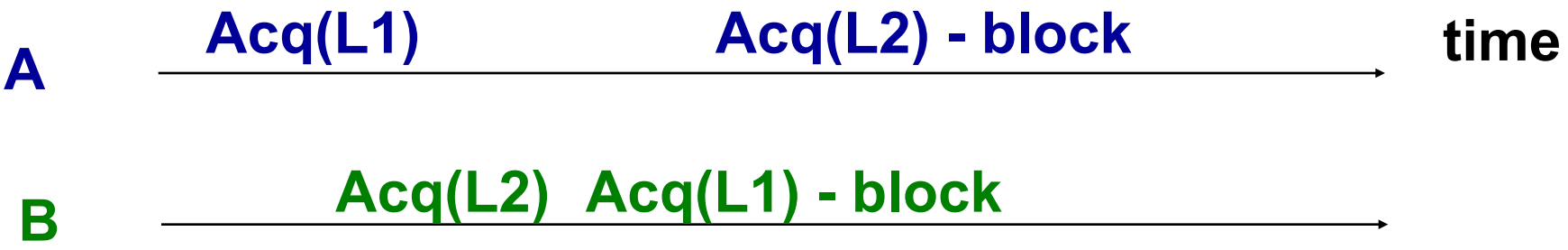
11 RELEASE (*p.buffer_lock*)

Would race conditions still happen?

Deadlocks

- Locks can introduce their own problems
  - Deadlock – e.g. thread A waits for thread B, while thread B waits for thread A
- Example:

Thread A	Thread B
ACQUIRE (L1)	ACQUIRE (L2)
ACQUIRE (L2)	ACQUIRE (L1)



Implementing lock

```
1 structure lock
2     integer state

4 procedure FAULTY_ACQUIRE (lock reference L)
5     while L.state = LOCKED do nothing
6     L.state <- LOCKED

8 procedure RELEASE (lock reference L)
9     L.state <- UNLOCKED
```

What can go wrong?

Implementing Locks

- Correct implementation requires “single-acquire”
  - Several threads may try to acquire the same lock concurrently, only a single thread proceeds
  - ACQUIRE itself needs to be a before-or-after action
  - Making multi-step operations on shared variables before-or-after, need to make an operation on single lock before-or-after

Implementing locks

- Help from a special instruction
  - Read-and-set memory (RSM)

```
procedure RSM (reference mem)
    do atomic
        r <- mem
        mem <- LOCKED
    return r
```

Implementing lock with RSM

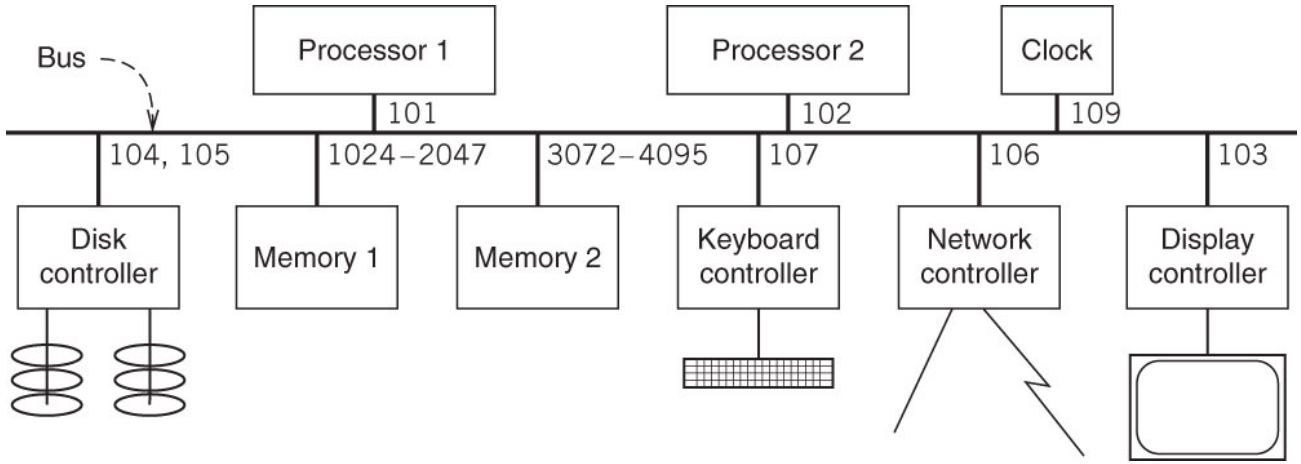
```
1 structure lock
2    integer state

4 procedure ACQUIRE (lock reference L)
5    R1 <- RSM (L.state)  // read and set lock L
6    while R1 = LOCKED do    // if it was locked
7        R1 <- RSM (L.state)      // keep trying

8 procedure RELEASE (lock reference L)
9    L.state <- UNLOCKED
```

Architecture implications

- Multiple devices might be independently trying to reference *mem*
- Processors cache memory including *mem*
- High-performance processors may reorder memory requests
- Synchronization primitives available in instruction set of a microprocessor, in addition to loads/stores
  - Locks with regular loads/stores possible (see sidebar 5.2) but inefficient



E.g. x86 LOCK

- Causes the processor's LOCK# signal to be asserted during execution of the accompanying instruction (turns the instruction into an atomic instruction). In a multiprocessor environment, the LOCK# signal insures that the processor has exclusive use of any shared memory while the signal is asserted.
- The LOCK prefix is typically used with the BTS instruction to perform a read-modify-write operation on a memory location in shared memory environment.
  - www.intel.com/assets/pdf/manual/253666.pdf

E.g. x86 LOCK

- Beginning with the P6 family processors, when the LOCK prefix is prefixed to an instruction and the memory area being accessed is cached internally in the processor, the LOCK# signal is generally not asserted. Instead, only the processor's cache is locked. Here, the processor's cache coherency mechanism insures that the operation is carried out atomically with regards to memory.

Reading

- Sections 5.3-5.5