Chapter 5 – Asynchronous Concurrent Execution

Outline
5.1 Introduction
5.2 Mutual Exclusion
5.2.1 Java Multithreading Case Study
5.2.2 Critical Sections
5.2.3 Mutual Exclusion Primitives
5.3 Implementing Mutual Exclusion Primitives
5.4 Software Solutions to the Mutual Exclusion Problem
5.4.1 Dekker’s Algorithm
5.4.2 Peterson’s Algorithm
5.4.3 N-Thread Mutual Exclusion: Lamport’s Bakery Algorithm
5.5 Hardware Solutions to the Mutual Exclusion Problem
5.5.1 Disabling Interrupts
5.5.2 Test-and-Set Instruction
5.5.3 Swap Instruction

Outline (continued)
5.6 Semaphores
5.6.1 Mutual Exclusion with Semaphores
5.6.2 Thread Synchronization with Semaphores
5.6.3 Counting Semaphores
5.6.4 Implementing Semaphores
Objectives

• After reading this chapter, you should understand:
  – the challenges of synchronizing concurrent processes and threads.
  – critical sections and the need for mutual exclusion.
  – how to implement mutual exclusion primitives in software.
  – hardware mutual exclusion primitives.
  – semaphore usage and implementation.

5.1 Introduction

• Concurrent execution
  – More than one thread exists in system at once
  – Can execute independently or in cooperation
  – Asynchronous execution
    • Threads generally independent
    • Must occasionally communicate or synchronize
    • Complex and difficult to manage such interactions

Interleaved Execution

• Machine language for \( x = x + 1 \)
  
  \[
  A = X \\
  A = A + 1 \\
  X = A
  \]

  • \( X \) should be 2 after both processes execute
  • Due to parallel processing or context switching, above instructions can be interleaved producing inconsistent result
Interleaved Execution

\[
\begin{align*}
\text{p1} : & \quad A = X \quad (A = 0) \\
\text{p1} : & \quad A = A + 1 \quad (A = 1) \\
\text{p2} : & \quad A = X \quad (A = 0) \\
\text{p2} : & \quad A = A + 1 \quad (A = 1) \\
\text{p1} : & \quad X = A \quad (X = 1) \\
\text{p2} : & \quad X = A \quad (x = 1)
\end{align*}
\]

- The value of \( X \) may be either 1 or 2, where the correct result should be 2 depending on how they are scheduled.

Critical Section

- Problems?
  - Must prevent the interleaving or concurrent execution of the three machine instructions

- Critical Section (CS)
  - Any segment of code involved in reading and writing a shared data area
  - \( n \) processes all competing to use some shared data
  - Each process has a code segment, called critical section, in which the shared data is accessed.
  - Problem: ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

5.2 Mutual Exclusion

- Problem of two threads accessing data simultaneously
  - Data can be put in inconsistent state
    - Context switch can occur at anytime, such as before a thread finishes modifying value
  - Such data must be accessed in mutually exclusive way
    - Only one thread allowed access at one time
    - Others must wait until resource is unlocked
    - Serialized access
    - Must be managed such that wait time not unreasonable
5.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

- Producer/Consumer relationship
  - One thread creates data to store in shared object
  - Second thread reads data from that object
  - Large potential for data corruption if unsynchronized

Producer/Comsumer Bounded Buffer Problem

- Producer
  - Generates data elements and adds them to a shared buffer
- Consumer
  - Removes data elements from the shared buffer concurrently

```
Producer
while (1) {
  while (((in + 1) % BUFFER_SIZE) == out)    // Full
    buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
}

Consumer
while (1) {
  while (in == out)    // Empty
    nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE
}
```
Bounded Buffer Problem
(BUFFER_SIZE items)

Producer
while (1) {
while (counter == BUFFER_SIZE) // Full
buffer[n] = nextProduced;
in = (in + 1) % BUFFER_SIZE;
}
Consumer
while (1) {
while (counter == 0) // Empty
nextConsumed = buffer[out];
out = (out + 1) % BUFFER_SIZE
counter--;
}

• Shared variables:
  - buffer[]
  - in, counter, out

Bounded Buffer
- Machine language for count ++
  \[ A = \text{counter}, \quad A = A + 1, \quad \text{counter} = A \]
- Machine language for count --
  \[ A = \text{counter}, \quad A = A - 1, \quad \text{counter} = A \]

• The value of count may be either 4 or 6, where the correct result should be 5 depending on how they are scheduled.

context switching
5.2.1 Java Multithreading Case Study, Part II:
A Producer/Consumer Relationship in Java

Figure 5.1 Buffer interface used in producer/consumer examples.

```java
1 // Fig. 5.1: Buffer.java
2 // Buffer interface specifies methods to access buffer data.
3 4 public interface Buffer {
5   6     public void put(int value); // place value into buffer
7   8     public int get(); // return value from buffer
9 }
```

5.2.1 Java Multithreading Case Study, Part II:
A Producer/Consumer Relationship in Java

Figure 5.2 Producer class represents the producer thread in a producer/consumer relationship. (1 of 3)

```java
1 // Fig. 5.2: Producer.java
2 // Producer’s run method controls a producer thread that
3 // stores values from 1 to 4 in Buffer sharedLocation.
4 5 public class Producer extends Thread {
6   7     private Buffer sharedLocation; // reference to shared object
8     9     // Producer constructor
10    11     public Producer (Buffer shared)
12     13     { super("Producer" ); // create thread named "Producer"
14     15     // initialize sharedLocation
16     17     } // end Producer constructor
18
19     // Producer method: run stores values from
20    21     // 1 to 4 in Buffer sharedLocation
22     23     public void run() {
24         25         for (int count = 1; count <= 4; count++)
26             27             try
28                 29                 Thread.sleep (int (Math.random() * 1000));
30             31         // waitLocation.wait (count); // write to the buffer
32             33         } // end try
34     35     } // end run
36
37     // catch InterruptedException exception
38     39     catch (InterruptedException exception)
40                 41         exception.printStackTrace();
42     43     } // end catch
44     45     } // end for
46
```

5.2.1 Java Multithreading Case Study, Part II:
A Producer/Consumer Relationship in Java

Figure 5.2 Producer class represents the producer thread in a producer/consumer relationship. (2 of 3)
5.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

5.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

### Figure 5.2
Producer class represents the producer thread in a producer/consumer relationship. (3 of 3)

```java
36 System.err.println( getNum() + " done producing." );
37 "informing: " + getName() + ": " );
38 }
39 } // end method run
40 } // end class Producer
```

### Figure 5.3
Consumer class represents the consumer thread in a producer/consumer relationship. (1 of 3)

```java
// Fig. 5.3: Consumer.java

1 public class Consumer extends Thread {
2 // Consumer's run method controls a thread that looks for four
3 // times and reads a value from sharedLocation each time.
4 // Consumer constructor
5 public Consumer( Buffer shared ) {
6 private Buffer sharedLocation; // reference to shared object
7 // Consumer constructor
8 public Consumer( Buffer shared ) {
9 super( "Consumer" ); // create thread named "Consumer"
10 sharedLocation = shared; // initialize sharedLocation
11 }
12 // Consumer thread
13 }
14 } // end class Consumer
```

### Figure 5.3
Consumer class represents the consumer thread in a producer/consumer relationship. (2 of 3)

```java
// read sharedLocation's value four times and sum the values
// alternate between sleeping and getting Buffer value
// for ( int count = 1 ) ; count <= 2 ; ++count )
19 try {
20 Thread.sleep( ( int ) ( Math.random() * 800 ) );
21 sum += sharedLocation.get();
22 }
23 } // end Consumer thread
```

### Figure 5.3
Consumer class represents the consumer thread in a producer/consumer relationship. (3 of 3)

```java
// read sharedLocation's value four times and sum the values
13 // alternate between sleeping and getting Buffer value
14 // for ( int count = 1 ) ; count <= 4 ; ++count )
15 try {
16 int sum = 0;
17 // alternate between sleeping and getting Buffer value
18 for ( int count = 1 ) ; count <= 4 ; ++count )
19 try {
20 Thread.sleep( ( int ) ( Math.random() * 800 ) );
21 sum += sharedLocation.get();
22 }
23 } // end Consumer thread
```
5.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

Figure 5.3 Consumer class represents the consumer thread in a producer/consumer relationship. (3 of 3)

```java
31 // if sleeping thread interrupted, print stack trace
32 catch (InterruptedException exception) {
33     exception.printStackTrace();
34 }
35 // end for
36 System.err.println( getname() + " read values totaling: "+sum); // end method run
37 }
38 // end class Consumer
```

5.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

Figure 5.4 UnsynchronizedBuffer class maintains the shared integer that is accessed by a producer thread and a consumer thread via methods set and get. (1 of 2)

```java
1 // Fig. 5.4: UnsynchronizedBuffer.java
2 // UnsynchronizedBuffer represents a single shared integer.
3 public class UnsynchronizedBuffer implements Buffer {
4     private int buffer = -1; // shared by Producer and Consumer
5     // place value into buffer
6     public void set(int value) {
7         System.err.println( Thread.currentThread().getName() + " write " + value);
8     }
9     // end method set
10     public int get() {
11         System.err.println( Thread.currentThread().getName() + " read " + buffer);
12         return buffer;
13     }
14     // end method get
15 }
16 // end class UnsynchronizedBuffer
```

5.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

Figure 5.4 UnsynchronizedBuffer class maintains the shared integer that is accessed by a producer thread and a consumer thread via methods set and get. (2 of 2)
5.2.1 Java Multithreading Case Study, Part II:
A Producer/Consumer Relationship in Java

Figure 5.5 SharedBuffer class enables threads to modify a shared object without synchronization. (1 of 4)

```java
public class SharedBufferTest {
    public static void main(String[] args) {
        Buffer sharedLocation = new JavaBuffer();
        // create producer and consumer objects
        Producer producer = new Producer(sharedLocation);
        Consumer consumer = new Consumer(sharedLocation);
        producer.start(); // start producer thread
        consumer.start(); // start consumer thread
    }
}
```

5.2.1 Java Multithreading Case Study, Part II:
A Producer/Consumer Relationship in Java

Figure 5.5 SharedBuffer class enables threads to modify a shared object without synchronization. (2 of 4)

Sample Output 1:
- Consumer reads 1
- Producer writes 1
- Consumer reads 1
- Consumer reads 1
- Producer reads 1
- Value 1 is lost
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Value 4 is used twice
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
- Producer reads 1
5.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

Figure 5.5 SharedBuffer class enables threads to modify a shared object without synchronization. (4 of 4)

Sample Output:

```
Producer writes 1
Producer writes 2
Producer writes 3
Producer writes 4
Producer done producing.
Terminating Producer.
Consumer reads 1
Consumer reads 2
Consumer reads 3
Consumer reads 4
Consumer reads values totaling: 10.
Terminating Consumer.
```

5.2.2 Critical Sections

- Most code is safe to run concurrently
- Sections where shared data is modified must be protected
  - Known as critical sections
  - Only one thread can be in its critical section at once
    - Must be careful to avoid infinite loops and blocking inside a critical section
  - (Common misunderstanding) Note that critical section doesn’t mean the section of the shared data but the section of the code that access the shared data

5.2.3 Mutual Exclusion Primitives

- Indicate when critical data is about to be accessed
  - Mechanisms are normally provided by programming language or libraries
  - Delimit beginning and end of critical section
    - `enterMutualExclusion`
    - `exitMutualExclusion`
5.3 Implementing Mutual Exclusion Primitives

- Common properties of mutual exclusion primitives
  - Each mutual exclusion machine language instruction is executed indivisibly
  - Cannot make assumptions about relative speed of thread execution
  - Thread not in its critical section cannot block other threads from entering their critical sections
  - Thread may not be indefinitely postponed from entering its critical section