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
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
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
// Fig. 5.1: Buffer.java
// Buffer interface specifies methods to access buffer data.

public interface Buffer {
    public void set(int value); // place value into Buffer
    public int get(); // return value from Buffer
}
```

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
// Fig. 5.2: Producer.java
// Producer's run method controls a producer thread that
// stores values from 1 to 4 in Buffer sharedLocation.

public class Producer extends Thread {
    private Buffer sharedLocation; // reference to shared object

    // Producer constructor
    public Producer( Buffer shared ) {
        super( "Producer" ); // create thread named "Producer"
        sharedLocation = shared; // initialize sharedLocation
    }
}
```
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)

```java
public void run()
{
  for (int count = 1; count <= 4; count++)
  {
    // sleep 0 to 3 seconds, then place value in Buffer
    try
    {
      Thread.sleep( (int)(Math.random() * 3001) );
      sharedLocation.set(count); // write to the buffer
    } // end try
    // if sleeping thread interrupted, print stack trace
    catch (InterruptedException exception)
    {
      exception.printStackTrace();
    } // end catch
  } // end for
}
```

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

```java
System.err.println( getName() + " done producing."
  + "\nTerminating " + getName() + "." );

} // end method run

} // end class Producer
```
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. (1 of 3)

```java
1 // Fig. 5.3: Consumer.java
2 // Consumer's run method controls a thread that loops four
3 // times and reads a value from sharedLocation each time.
4 public class Consumer extends Thread
5 {
6    private Buffer sharedLocation; // reference to shared object
7    // Consumer constructor
8    public Consumer(Buffer shared)
9    {
10       super("Consumer"); // create thread named "Consumer"
11       sharedLocation = shared; // initialize sharedLocation
12    } // end Consumer constructor
13 }
```

---

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. (2 of 3)

```java
15 // read sharedLocation's value four times and sum the values
16 public void run()
17 {
18    int sum = 0;
19    // alternate between sleeping and getting Buffer value
20    for (int count = 1; count <= 4; ++count)
21    {
22       // sleep 0-3 seconds, read Buffer value and add to sum
23       try
24       {
25          Thread.sleep((int)(Math.random() * 3001));
26          sum += sharedLocation.get();
27       }
28    }
29 }
```
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 {
34 exception.printStackTrace();
35 }
36 } // end for
37 System.err.println( getName() + " read values totaling: "
38 + sum + ".\nTerminating " + getName() + ";" );
39 } // end method run
40 } // 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 {
5     private int buffer = -1; // shared by Producer and Consumer
6     // place value into buffer
7     public void set( int value )
8     {
9         System.err.println( Thread.currentThread().getName() +
10             " writes " + value );
11         buffer = value;
12     } // end method set
13 }
```
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)

```java
17 // return value from buffer
18 public int get()
19 {
20 System.err.println( Thread.currentThread().getName() + " reads " + buffer);
21
22 return buffer;
23 } // end method get
24 // end class UnsynchronizedBuffer

```

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

```java
1 // Fig. 5.5: SharedBufferTest.java
2 // SharedBufferTest creates producer and consumer threads.
3 public class SharedBufferTest
4 {
5     public static void main( String[] args )
6     {
7         // create shared object used by threads
8         Buffer sharedLocation = new UnsynchronizedBuffer();
9
10         // create producer and consumer objects
11         Producer producer = new Producer( sharedLocation );
12         Consumer consumer = new Consumer( sharedLocation );
13
14         producer.start(); // start producer thread
15         consumer.start(); // start consumer thread
16
17     } // end main
18 } // end class SharedCell
```

© 2004 Deitel & Associates, Inc. All rights reserved.
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
Consumer reads 1
Consumer read values totaling: 2.
Terminating Consumer.
Producer writes 2
Producer writes 3
Producer writes 4
Producer done producing.
Terminating Producer.

Sample Output 2:
Producer writes 1
Producer writes 2
Consumer reads 2
Producer writes 3
Consumer reads 3
Producer writes 4
Producer done producing.
Terminating Producer.
Consumer reads 4
Consumer reads 4
Consumer read values totaling: 13.
Terminating Consumer.
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  
Consumer reads 1  
Producer writes 2  
Consumer reads 2  
Producer writes 3  
Consumer reads 3  
Producer writes 4  
Producer done producing.  
Terminating Producer.  
Consumer reads 4  
Consumer read 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
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
5.4.1 Dekker’s Algorithm

- First version of Dekker’s algorithm
  - Succeeds in enforcing mutual exclusion
  - Uses variable to control which thread can execute
  - Constantly tests whether critical section is available
    - Busy waiting
    - Wastes significant processor time
  - Problem known as lockstep synchronization
    - Each thread can execute only in strict alternation

Figure 5.6 Mutual exclusion implementation – version 1 (1 of 2).
5.4.1 Dekker’s Algorithm

- Second version
  - Removes lockstep synchronization
  - Violates mutual exclusion
    - Thread could be preempted while updating flag variable
  - Not an appropriate solution
5.4.1 Dekker’s Algorithm

Figure 5.7 Mutual exclusion implementation – version 2 (1 of 3).

```java
4 System:
5 boolean t1Inside = false;
6 boolean t2Inside = false;
7 startThreads(); // initialize and launch both threads
8 Thread T1:
9 void main() {
10   while (!done)
11     { // inside critical section
12       while (!t2Inside); // enter MutualExclusion
13       t1Inside = true; // enter MutualExclusion
14     // critical section code
15     t1Inside = false; // exit MutualExclusion
16   }
17 }
```

5.4.1 Dekker’s Algorithm

Figure 5.7 Mutual exclusion implementation – version 2 (2 of 3).

```java
22 // code outside critical section
23 } // end outer while
24 } // end Thread T1
25
26 Thread T2:
27
28 void main() {
29   while (!done)
30     { // inside critical section
31       while (!t1Inside); // enter MutualExclusion
32       t2Inside = true; // enter MutualExclusion
33     // critical section code
34   }
```

© 2004 Deitel & Associates, Inc. All rights reserved.
5.4.1 Dekker’s Algorithm

**Figure 5.7** Mutual exclusion implementation – version 2 (3 of 3).

```c
39 t2Inside = false; // exitMutualExclusion
40 // code outside critical section
41 42 } // end outer while
43 44 } // end Thread T2
```

5.4.1 Dekker’s Algorithm

- **Third version**
  - Set critical section flag before entering critical section test
    - Once again guarantees mutual exclusion
  - Introduces possibility of deadlock
    - Both threads could set flag simultaneously
    - Neither would ever be able to break out of loop
  - Not a solution to the mutual exclusion problem
5.4.1 Dekker’s Algorithm

Figure 5.8 Mutual exclusion implementation – version 3 (1 of 2).

```java
System:
3    boolean t1WantsToEnter = false;
4    boolean t2WantsToEnter = false;
5    startThreads(); // initialize and launch both threads

Thread T1:
9    void main()
11    {while (!done)
12        {t1WantsToEnter = true; // enterMutualExclusion
13        while (t2WantsToEnter); // enterMutualExclusion
14        // critical section code
16        t1WantsToEnter = false; // exitMutualExclusion
18        // code outside critical section
```

© 2004 Deitel & Associates, Inc. All rights reserved.

5.4.1 Dekker’s Algorithm

Figure 5.8 Mutual exclusion implementation – version 3 (2 of 2).

```java
Thread T2:
28    void main()
30    {while (!done)
32        {t2WantsToEnter = true; // enterMutualExclusion
34        while (t1WantsToEnter); // enterMutualExclusion
37        // critical section code
39        t2WantsToEnter = false; // exitMutualExclusion
42        // code outside critical section
44        } // end outer while
46    } // end Thread T2
```

© 2004 Deitel & Associates, Inc. All rights reserved.
5.4.1 Dekker’s Algorithm

- Fourth version
  - Sets flag to false for small periods of time to yield control
  - Solves previous problems, introduces indefinite postponement
    - Both threads could set flags to same values at same time
    - Would require both threads to execute in tandem (unlikely but possible)
  - Unacceptable in mission- or business-critical systems

---

**Figure 5.9** Mutual exclusion implementation – version 4 (1 of 4).

```java
System:
3    boolean t1WantsToEnter = false;
4    boolean t2WantsToEnter = false;
5
6    startThreads(); // initialize and launch both threads

Thread T1:
10   void main()
11   {
12       while ( !done )
13           {
14           t1WantsToEnter = true;  // enterMutualExclusion
```

© 2004 Deitel & Associates, Inc. All rights reserved.
5.4.1 Dekker’s Algorithm

Figure 5.9 Mutual exclusion implementation – version 4 (2 of 4).

```c
while ( t2WantsToEnter ) // enterMutualExclusion
{
    t1WantsToEnter = false; // enterMutualExclusion
    // wait for small, random amount of time
    t1WantsToEnter = true;
} // end while
// critical section code
// exitMutualExclusion
// code outside critical section
} // end outer while
```

5.4.1 Dekker’s Algorithm

Figure 5.9 Mutual exclusion implementation – version 4 (3 of 4).

```c
} // end Thread T1
Thread T2:
void main()
{
    while ( !done )
    {
        t2WantsToEnter = true; // enterMutualExclusion
        while ( t1WantsToEnter ) // enterMutualExclusion
        {
            t2WantsToEnter = false; // exitMutualExclusion
            // wait for small, random amount of time
            t2WantsToEnter = true;
        } // end while
```
5.4.1 Dekker’s Algorithm

Figure 5.9 Mutual exclusion implementation – version 4 (4 of 4).

```c
// critical section code
52
53
tzWantsToEnter = false; // exitMutualExclusion
54
55
// code outside critical section
56
57
} // end outer while
58
59
} // end Thread T2
```

• Dekker’s Algorithm
  – Proper solution
  – Uses notion of favored threads to determine entry into critical sections
    • Resolves conflict over which thread should execute first
    • Each thread temporarily unsets critical section request flag
    • Favored status alternates between threads
  – Guarantees mutual exclusion
  – Avoids previous problems of deadlock, indefinite postponement
5.4.1 Dekker’s Algorithm

Figure 5.10 Dekker’s Algorithm for mutual exclusion. (1 of 4)

System:

1. int favoredThread = 1;
2. boolean t1WantsToEnter = false;
3. boolean t2WantsToEnter = false;

4. startThreads(); // initialize and launch both threads

Thread T1:

5. void main()
6. {
7.     while (!done)
8.     {
9.         t1WantsToEnter = true;

5.4.1 Dekker’s Algorithm

Figure 5.10 Dekker’s Algorithm for mutual exclusion. (2 of 4)

17.     while (t2WantsToEnter)
18.     {
19.         if (favoredThread == 2)
20.         {
21.             t1WantsToEnter = false;
22.             while (favoredThread == 2); // busy wait
23.             t1WantsToEnter = true;
24.         } // end if
25.     } // end while
26.     // critical section code
27.     favoredThread = 2;
28.     t1WantsToEnter = false;
29.     // code outside critical section
30.     } // end outer while
31.     } // end Thread T1
5.4.1 Dekker’s Algorithm

Figure 5.10 Dekker’s Algorithm for mutual exclusion. (3 of 4)

```java
Thread T2:
void main()
{
  while (!done)
  {
    t2WantsToEnter = true;
    while (t1WantsToEnter)
    {
      if (favoredThread == 1)
      {
        t2WantsToEnter = false;
        while (favoredThread == 1); // busy wait
        t2WantsToEnter = true;
      } // end if
    } // end while
  }
}
```

5.4.1 Dekker’s Algorithm

Figure 5.10 Dekker’s Algorithm for mutual exclusion. (4 of 4)

```java
// critical section code
FavoredThread = 1;
t2WantsToEnter = false;
// code outside critical section
} // end outer while
} // end Thread T2
```
5.4.2 Peterson’s Algorithm

- Less complicated than Dekker’s Algorithm
  - Still uses busy waiting, favored threads
  - Requires fewer steps to perform mutual exclusion primitives
  - Easier to demonstrate its correctness
  - Does not exhibit indefinite postponement or deadlock
5.4.2 Peterson’s Algorithm

Figure 5.11 Peterson’s Algorithm for mutual exclusion. (2 of 3)

```
void main()
{
    while (!done)
    {
        t1WantsToEnter = true;
        favoredThread = 2;
        while (t2WantsToEnter && favoredThread == 2);
        // critical section code
        t1WantsToEnter = false;
        // code outside critical section
    } // end while
} // end Thread T1
```

---

5.4.2 Peterson’s Algorithm

Figure 5.11 Peterson’s Algorithm for mutual exclusion. (3 of 3)

```
void main()
{
    while (!done)
    {
        t2WantsToEnter = true;
        favoredThread = 1;
        while (t1WantsToEnter && favoredThread == 1);
        // critical section code
        t2WantsToEnter = false;
        // code outside critical section
    } // end while
} // end Thread T2
```
5.4.3 N-Thread Mutual Exclusion:
Lamport’s Bakery Algorithm

- Applicable to any number of threads
  - Creates a queue of waiting threads by distributing numbered “tickets”
  - Each thread executes when its ticket’s number is the lowest of all threads
  - Unlike Dekker’s and Peterson’s Algorithms, the Bakery Algorithm works in multiprocessor systems and for \( n \) threads
  - Relatively simple to understand due to its real-world analog

---

**Figure 5.12** Lamport’s Bakery Algorithm. (1 of 3)

```plaintext
System:
1 // array that records which threads are taking a ticket
2 boolean choosing[n];
3 // value of the ticket for each thread initialized to 0
4 int ticket[n];
5 startThreads(); // initialize and launch all threads
```
5.4.3 N-Thread Mutual Exclusion:
Lamport’s Bakery Algorithm

Figure 5.12 Lamport’s Bakery Algorithm. (2 of 3)

```c
11  // Thread T;
12  void main()
13  {
14      x = threadNumber(); // store current thread number
15  while (!done)
16  {
17      // take a ticket
18      choosing[x] = true; // begin ticket selection process
19      ticket[x] = maxValue(tick) + 1;
20      choosing[x] = false; // end ticket selection process
21  // wait for number to be called by comparing current
22  // ticket value to other thread’s ticket value
23  for (int i = 0; i < n; i++)
24  {
25      if (i == x)
26      {
27          continue; // no need to check own ticket
28      } // end if
29  }
30  // busy wait while thread[i] is choosing
31  while (choosing[i] != false);
32  // busy wait until current ticket value is lowest
33  while (ticket[i] != 0 && ticket[i] < ticket[x]);
34  // tie-breaker code favors smaller thread number
35  if (ticket[i] == ticket[x] && i < x)
36  // loop until thread[i] leaves its critical section
37  while (ticket[i] != 0); // busy wait
38  // end for
39  // critical section code
40  ticket[x] = 0; // exitMutualExclusion
41  // code outside critical section
42  } // end while
43  // end Thread T
```

5.4.3 N-Thread Mutual Exclusion:
Lamport’s Bakery Algorithm

Figure 5.12 Lamport’s Bakery Algorithm. (3 of 3)
5.5 Hardware Solutions to the Mutual Exclusion Problem

- Implementing mutual exclusion in hardware
  - Can improve performance
  - Can decreased development time
    - No need to implement complex software mutual exclusion solutions like Lamport’s Algorithm

5.5.1 Disabling Interrupts

- Disabling interrupts
  - Works only on uniprocessor systems
  - Prevents the currently executing thread from being preempted
  - Could result in deadlock
    - For example, thread waiting for I/O event in critical section
  - Technique is used rarely
5.5.2 Test-and-Set Instruction

• Use a machine-language instruction to ensure that mutual exclusion primitives are performed indivisibly
  – Such instructions are called atomic
  – Machine-language instructions do not ensure mutual exclusion alone—the software must properly use them
    • For example, programmers must incorporate favored threads to avoid indefinite postponement
  – Used to simplify software algorithms rather than replace them

• Test-and-set instruction
  – testAndSet(a, b) copies the value of b to a, then sets b to true
  – Example of an atomic read-modify-write (RMW) cycle

```java
System:
boolean occupied = false;
startThreads(); // initialize and launch both threads

Thread 1:
void main()
{
  boolean p1MustWait = true;
  while (!done )
  {
    while ( p1MustWait )
    {
      testAndSet( p1MustWait, occupied );
    }
  }
```

Figure 5.13 testAndSet instruction for mutual exclusion. (1 of 3)
5.5.2 Test-and-Set Instruction

Figure 5.13 testAndSet instruction for mutual exclusion. (2 of 3)

```java
20     // critical section code
21
22     p1MustWait = true;
23     occupied = false;
24
25     // code outside critical section
26
27     } // end while
28
29     } // end Thread T1
30
31     Thread T2:
32
33     void main()
34     {
35         boolean p2MustWait = true;
36
37         while ( !done )
38         {
39             while ( p2MustWait )
40             {
41                 testAndSet( p2MustWait, occupied );
42             }
43
44             // critical section code
45             p2MustWait = true;
46             occupied = false;
47
48             // code outside critical section
49
50         } // end while
51
52     } // end Thread T2
```

5.5.2 Test-and-Set Instruction

Figure 5.13 testAndSet instruction for mutual exclusion. (3 of 3)
5.5.3 Swap Instruction

- \texttt{swap(a, b)} exchanges the values of \texttt{a} and \texttt{b} atomically
- Similar in functionality to test-and-set
  - \texttt{swap} is more commonly implemented on multiple architectures
Figure 5.14 swap instruction for mutual exclusion. (2 of 3)

```java
while (!done) {
    do {
        swap(p1MustWait, occupied);
    } while (p1MustWait);

    // critical section code
    p1MustWait = true;
    occupied = false;
    // code outside critical section
}
// end while

} // end Thread T1
```

Figure 5.14 swap instruction for mutual exclusion. (3 of 3)

```java
Thread T2:
void main()
{
    boolean p2MustWait = true;
    while (!done) {
        do {
            swap(p2MustWait, occupied);
        } while (p2MustWait);

        // critical section code
        p2MustWait = true;
        occupied = false;
        // code outside critical section
    }
// end while
}
// end Thread T2
```
5.6 Semaphores

- Semaphores
  - Software construct that can be used to enforce mutual exclusion
  - Contains a protected variable
    - Can be accessed only via wait and signal commands
    - Also called $P$ and $V$ operations, respectively

5.6.1 Mutual Exclusion with Semaphores

- Binary semaphore: allow only one thread in its critical section at once
  - Wait operation
    - If no threads are waiting, allow thread into its critical section
    - Decrement protected variable (to 0 in this case)
    - Otherwise place in waiting queue
  - Signal operation
    - Indicate that thread is outside its critical section
    - Increment protected variable (from 0 to 1)
    - A waiting thread (if there is one) may now enter
5.6.1 Mutual Exclusion with Semaphores

Figure 5.15 Mutual exclusion with semaphores.

```c
1 System;
2 // create semaphore and initialize value to 1
3 Semaphore occupied = new Semaphore(1);
4 startThreads(); // initialize and launch both threads
5
6 Thread Tc
7
8 void main()
9 {
10    while(!done)
11    {
12        P(occupied); // wait
13        // critical section code
14        V(occupied); // signal
15        // code outside critical section
16    } // end while
17    } // Thread Tc
18}
```

5.6.2 Thread Synchronization with Semaphores

- Semaphores can be used to notify other threads that events have occurred
  - Producer-consumer relationship
    - Producer enters its critical section to produce value
    - Consumer is blocked until producer finishes
    - Consumer enters its critical section to read value
    - Producer cannot update value until it is consumed
  - Semaphores offer a clear, easy-to-implement solution to this problem
5.6.2 Thread Synchronization with Semaphores

Figure 5.16 Producer/consumer relationship implemented with semaphores. (1 of 2)

```cpp
1 System:
2 // semaphores that synchronize access to shared Value
3 Semaphore valueProduced = new Semaphore(0);
4 Semaphore valueConsumed = new Semaphore(1);
5 int sharedValue; // variable shared by producer and consumer
6 startThreads(); // initialize and launch both threads
7 Producer Thread:
8 void main()
9 {
10   int newValueProduced; // variable to store value produced
11   while (!done)
12     { newValueProduced = generateTheValue(); // produce value
13       P(valueConsumed); // wait until value is consumed
14       sharedValue = newValueProduced; // critical section
15       V(valueProduced); // signal that value has been produced
16     } // end while
17 } // end producer thread
```

Figure 5.16 Producer/consumer relationship implemented with semaphores. (2 of 2)

```cpp
Consumer Thread:

25 void main()
26 {
27   int newValue; // variable to store value consumed
28   while (!done)
29     { P(valueProduced); // wait until value is produced
30       newValueConsumed = sharedValue; // critical section
31       V(newValueConsumed); // signal that value has been consumed
32       processTheValue(newValueConsumed); // process the value
33     } // end while
34 } // end consumer thread
```
5.6.3 Counting Semaphores

- **Counting semaphores**
  - Initialized with values greater than one
  - Can be used to control access to a pool of identical resources
    - Decrement the semaphore’s counter when taking resource from pool
    - Increment the semaphore’s counter when returning it to pool
    - If no resources are available, thread is blocked until a resource becomes available

5.6.4 Implementing Semaphores

- **Semaphores can be implemented at application or kernel level**
  - Application level: typically implemented by busy waiting
    - Inefficient
  - Kernel implementations can avoid busy waiting
    - Block waiting threads until they are ready
  - Kernel implementations can disable interrupts
    - Guarantee exclusive semaphore access
    - Must be careful to avoid poor performance and deadlock
    - Implementations for multiprocessor systems must use a more sophisticated approach