Chapter 5 – Asynchronous Concurrent Execution

Chapter 5 – Asynchronous Concurrent Execution

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

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
   } // end Producer constructor

```

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)

```
   // Producer method run stores values from
   // 1 to 4 in Buffer sharedLocation
   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
```


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)

```

      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
// Fig. 5.3: Consumer.java
// Consumer's run method controls a thread that loops four
// times and reads a value from sharedLocation each time.

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

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


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

   // read sharedLocation's value four times and sum the values
   public void run()
   {
      int sum = 0;

      // alternate between sleeping and getting Buffer value
      for ( int count = 1; count <= 4; ++count )
      {
         // sleep 0-3 seconds, read Buffer value and add to sum
         try
         {
            Thread.sleep( ( int ) ( Math.random() * 3001 ) );
            sum += sharedLocation.get();
         }

```

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)

```
         // if sleeping thread interrupted, print stack trace
         catch ( InterruptedException exception )
         {
            exception.printStackTrace();
         }
      } // end for

      System.err.println( getName() + " read values totaling: "
         + sum + ".\nTerminating " + getName() + "." );

   } // end method run

} // 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)

```
// Fig. 5.4: UnsynchronizedBuffer.java
// UnsynchronizedBuffer represents a single shared integer.

public class UnsynchronizedBuffer implements Buffer
{
   private int buffer = -1; // shared by Producer and Consumer

   // place value into buffer
   public void set( int value )
   {
      System.err.println( Thread.currentThread().getName() +
         " writes " + value );

      buffer = value;
   } // end method set

```

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)

```
    // return value from buffer
    public int get()
    {
       System.err.println( Thread.currentThread().getName() +
          " reads " + buffer );

       return buffer;
    } // end method get

 } // end class UnsynchronizedBuffer
```


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)

```
// Fig. 5.5: SharedBufferTest.java
// SharedBufferTest creates producer and consumer threads.

public class SharedBufferTest
{
   public static void main( String [] args )
   {
      // create shared object used by threads
      Buffer sharedLocation = new UnsynchronizedBuffer();

      // 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

   } // end main

} // end class SharedCell
```

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.
```


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. (3 of 4)

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 3:

```
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

5.4.1 Dekker's Algorithm

Figure 5.6 Mutual exclusion implementation – version 1 (1 of 2).

```
System:

int threadNumber = 1;

startThreads(); // initialize and launch both threads

Thread T1:

void main() {

   while ( !done )
   {
      while ( threadNumber == 2 ); // enterMutualExclusion

      // critical section code

      threadNumber = 2; // exitMutualExclusion

      // code outside critical section

   } // end outer while
```

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

5.4.1 Dekker's Algorithm

Figure 5.6 Mutual exclusion implementation – version 1 (2 of 2).

```

} // end Thread T1

Thread T2:

void main() {

   while ( !done )
    {
      while ( threadNumber == 1 );  // enterMutualExclusion

      // critical section code

      threadNumber = 1; // exitMutualExclusion

      // code outside critical section

   } // end outer while

} // end Thread T2
```


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).

```
System:

boolean t1Inside = false;
boolean t2Inside = false;

startThreads(); // initialize and launch both threads

Thread T1:

void main() {

   while ( !done )
   {
      while ( t2Inside ); // enterMutualExclusion

      t1Inside = true; // enterMutualExclusion

      // critical section code

      t1Inside = false; // exitMutualExclusion

```


5.4.1 Dekker's Algorithm

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

```
      // code outside critical section

   } // end outer while

} // end Thread T1

Thread T2:

void main() {

   while ( !done )
   {
      while ( t1Inside ); // enterMutualExclusion

      t2Inside = true; // enterMutualExclusion

      // critical section code
```

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

5.4.1 Dekker’s Algorithm

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

```

         t2Inside = false; // exitMutualExclusion

         // code outside critical section

      } // end outer while

   } // 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).

```
System:

boolean t1WantsToEnter = false;
boolean t2WantsToEnter = false;

startThreads(); // initialize and launch both threads

Thread T1:

void main()
{
   while ( !done )
   {
      t1WantsToEnter = true; // enterMutualExclusion

      while ( t2WantsToEnter ); // enterMutualExclusion

      // critical section code

      t1WantsToEnter = false; // exitMutualExclusion

      // code outside critical section

```


5.4.1 Dekker's Algorithm

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

```
   } // end outer while

} // end Thread T1

Thread T2:

void main()
{
   while ( !done )
   {
      t2WantsToEnter = true; // enterMutualExclusion

      while ( t1WantsToEnter ); // enterMutualExclusion

      // critical section code

      t2WantsToEnter = false; // exitMutualExclusion

      // code outside critical section

   } // end outer while

} // end Thread T2
```

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

5.4.1 Dekker's Algorithm

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

```
System:

boolean t1WantsToEnter = false;
boolean t2WantsToEnter = false;

startThreads(); // initialize and launch both threads

Thread T1:

void main()
{
   while ( !done )
   {
      t1WantsToEnter = true;  // enterMutualExclusion

```

5.4.1 Dekker’s Algorithm

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

```
      while ( t2WantsToEnter )  // enterMutualExclusion
      {
         t1WantsToEnter = false;  // enterMutualExclusion

         // wait for small, random amount of time

         t1WantsToEnter = true;
      } // end while

      // critical section code

      t1WantsToEnter = false; // 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).

```
} // end Thread T1
```

Thread T_2:

```
void main()
{
   while ( !done )
   {
      t2WantsToEnter = true;  // enterMutualExclusion

      while ( t1WantsToEnter )  // enterMutualExclusion
      {
         t2WantsToEnter = false;  // enterMutualExclusion

         // 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).

```
         // critical section code

         t2WantsToEnter = false; // exitMutualExclusion

         // code outside critical section

      } // end outer while

   } // end Thread T2
```


5.4.1 Dekker's Algorithm

- 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:

int favoredThread = 1;
boolean t1WantsToEnter = false;
boolean t2WantsToEnter = false;

startThreads(); // initialize and launch both threads

Thread T1:

void main()
{
   while ( !done )
   {
      t1WantsToEnter = true;

```


5.4.1 Dekker's Algorithm

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

```
      while ( t2WantsToEnter )
      {
         if ( favoredThread == 2 )
         {
            t1WantsToEnter = false;
            while ( favoredThread == 2 ); // busy wait
            t1WantsToEnter = true;
         } // end if

      } // end while

      // critical section code

      favoredThread = 2;
      t1WantsToEnter = false;

      // code outside critical section

   } // end outer while

} // end Thread T1
```

5.4.1 Dekker's Algorithm

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

```
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)

```

      // 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. (1 of 3)

```
System:

int favoredThread = 1;
boolean t1WantsToEnter = false;
boolean t2WantsToEnter = false;

startThreads(); // initialize and launch both threads
```

5.4.2 Peterson's Algorithm

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

```
Thread T1:

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)

```
Thread T2:

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

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

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

```
System:

// array that records which threads are taking a ticket
boolean choosing[n];

// value of the ticket for each thread initialized to 0
int ticket[n];

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)

```
Thread Tx:

void main()
{
   x = threadNumber(); // store current thread number

   while ( !done )
   {
      // take a ticket
      choosing[x] = true; // begin ticket selection process
      ticket[x] = maxValue( ticket ) + 1;
      choosing[x] = false; // end ticket selection process

      // wait for number to be called by comparing current
      // ticket  value to other thread's ticket value
      for ( int i = 0; i < n; i++)
      {
         if ( i == x )
         {
            continue; // no need to check own ticket
         } // end if

```


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

Figure 5.12 Lamport's Bakery Algorithm. (3 of 3)

```
         // busy wait while thread[i] is choosing
         while ( choosing[i] != false );

         // busy wait until current ticket value is lowest
         while ( ticket[i] != 0 && ticket[i] < ticket[x] );

         // tie-breaker code favors smaller thread number
         if ( ticket[i] == ticket[x] && i < x )

            // loop until thread[i] leaves its critical section
            while ( ticket[i] != 0 ); // busy wait
      } // end for

      // critical section code

      ticket[x] = 0; // exitMutualExclusion

      // code outside critical section

   } // end while

} // end Thread TX
```

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

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

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

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

5.5.2 Test-and-Set Instruction

Figure 5.13 `testAndSet` instruction for mutual exclusion. (1 of 3)

```
System:

boolean occupied = false;

startThreads(); // initialize and launch both threads

Thread T1:

void main()
{
   boolean p1MustWait = true;

   while ( !done )
   {
      while ( p1MustWait )
      {
         testAndSet( p1MustWait, occupied );
      }

```

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

5.5.2 Test-and-Set Instruction

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

```
         // critical section code

         p1MustWait = true;
         occupied = false;

         // code outside critical section

      } // end while

   } // end Thread T1

   Thread T2:

```


5.5.2 Test-and-Set Instruction

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

```
void main()
{
   boolean p2MustWait = true;

   while ( !done )
   {
      while ( p2MustWait )
      {
         testAndSet( p2MustWait, occupied );
      }

      // critical section code

      p2MustWait = true;
      occupied = false;

      // code outside critical section

   } // end while

} // end Thread T2
```

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

5.5.3 Swap Instruction

- `swap(a, b)` exchanges the values of `a` and `b` atomically
- Similar in functionality to test-and-set
  - `swap` is more commonly implemented on multiple architectures

5.5.3 Swap Instruction

Figure 5.14 swap instruction for mutual exclusion. (1 of 3)

```
System:

boolean occupied = false;

startThreads(); // initialize and launch both threads

Thread T1:

void main()
{
   boolean p1MustWait = true;

```

5.5.3 Swap Instruction

Figure 5.14 swap instruction for mutual exclusion. (2 of 3)

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

      // critical section code

      p1MustWait = true;
      occupied = false;

      // code outside critical section

   } // end while

} // end Thread T1

```


5.5.3 Swap Instruction

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

```
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

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

5.6.1 Mutual Exclusion with Semaphores

Figure 5.15 Mutual exclusion with semaphores.

```
System:

// create semaphore and initialize value to 1
Semaphore occupied = new Semaphore(1);

startThreads(); // initialize and launch both threads

Thread TX:

void main()
{
   while ( !done )
   {
      P( occupied );  // wait

      // critical section code

      V( occupied ); // signal

      // code outside critical section
   } // end while
} // Thread TX
```


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)

```
System:
// semaphores that synchronize access to sharedValue
Semaphore valueProduced = new Semaphore(0);
Semaphore valueConsumed = new Semaphore(1);
int sharedValue; // variable shared by producer and consumer

startThreads(); // initialize and launch both threads

Producer Thread:

void main()
{
   int nextValueProduced; // variable to store value produced

   while ( !done )
   {
      nextValueProduced = generateTheValue(); // produce value
      P( valueConsumed ); // wait until value is consumed
      sharedValue = nextValueProduced; // critical section
     V( valueProduced );  // signal that value has been produced

   } // end while

} // end producer thread
```


5.6.2 Thread Synchronization with Semaphores

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

```

Consumer Thread:

void main()
{
   int nextValue; // variable to store value consumed

   while ( !done )
   {
      P( valueProduced ); // wait until value is produced
      nextValueConsumed = sharedValue; // critical section
     V( valueConsumed ); // signal that value has been consumed
     processTheValue( nextValueConsumed ); // process the value

   } // end while

} // end consumer thread
```

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

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