Chapter 6 – Concurrent Programming

Outline
6.1 Introduction
6.2 Monitors
   6.2.1 Condition Variables
   6.2.2 Simple Resource Allocation with Monitors
   6.2.3 Monitor Example: Circular Buffer
   6.2.4 Monitor Example: Readers and Writers
6.3 Java Monitors
6.4 Java Multithreading Case Study, Part III:
   Producer/Consumer Relationship in Java
6.5 Java Multithreading Case Study, Part IV:
   Circular Buffer in Java

Objectives

• After reading this chapter, you should understand:
  – how monitors synchronize access to data.
  – how condition variables are used with monitors.
  – solutions for classic problems in concurrent programming such as readers and writers and circular buffer.
  – Java monitors.
  – remote procedure calls.
6.2.1 Introduction

- Recent interest in concurrent programming languages
  - Naturally express solutions to inherently parallel problems
  - Due to proliferation of multiprocessing systems, distributed systems and massively parallel architectures
  - More complex than standard programs
    - More time required to write, test and debug

Semaphore

- Is Semaphore better than previous solutions like Peterson’s?
- Does Semaphore provide an appropriate solution to the critical section problem?
- What would be the drawbacks, if any, of Semaphore?
- Is there other better way than Semaphore?

Producer Process

\[
\text{do (}
\text{produce an item in } \text{nextp} \\
\text{wait(\text{empty});} \\
\text{wait(\text{mutex});} \\
\text{add nextp to buffer} \\
\text{signal(\text{mutex});} \\
\text{signal(\text{full});} \\
\text{) while (1);}
\]

Consumer Process

\[
\text{do (}
\text{wait(\text{full});} \\
\text{remove an item from buffer to } \text{nextc} \\
\text{signal(\text{mutex});} \\
\text{signal(\text{empty})} \\
\text{consume the item in } \text{nextc} \\
\text{) while (1);}
\]

Recall producer/consumer problem
Use three semaphores
semaphore full = 0;
empty = n;
mutex = 1;
Producer Process

```
do {
    produce an item in nextp
    wait(mutex);
    wait(empty);
    add nextp to buffer
    signal(mutex);
    signal(full);
} while (1);
```

Consumer Process

```
do {
    remove an item from buffer to nextc
    signal(mutex);
    signal(empty)
    consume the item in nextc
    signal(mutex);
    signal(full);
} while (1);
```

Semaphore, semaphore, semaphore, ...

- With semaphore, IPC and synchronization looks easy
- How careful we must be when using semaphore !!
- Tanenbaum said "It is like programming in assembly language, only worse, because the errors are race conditions, deadlocks, and other forms of unpredictable and irreproducible behavior"

More problems with Semaphores ??

- Use of semaphores to avoid confusion in dealing with process synchronization
- We still have timing errors and possible starvation

```
<table>
<thead>
<tr>
<th>signal (mutex)</th>
<th>wait (mutex)</th>
<th>wait (mutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>critical_section</td>
<td>critical_section</td>
<td>critical_section</td>
</tr>
<tr>
<td>wait (mutex)</td>
<td>wait (mutex)</td>
<td>signal (mutex)</td>
</tr>
</tbody>
</table>
```

Mutual exclusion | Deadlock | Both
Motivation

- Semaphores and Events are:
  - Powerful but low-level abstractions
    - Programming with them is highly error prone
    - Such programs are difficult to design, debug, and maintain
  - Not usable in distributed memory systems
- Need higher-level primitives
  - Based on semaphores or messages

Monitors

- To make it easier to write correct concurrent programs, Hoare (1974) and Brinch Hansen (1976) proposed a higher level synchronization primitive called **monitor**
- Monitor
  - A collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package
  - Follow principles of abstract data type (object-oriented)

6.2 Monitors

- Monitor
  - Contains data and procedures needed to allocate shared resources
    - Accessible only within the monitor
    - No way for threads outside monitor to access monitor data
- Resource allocation using monitors
  - Thread must call monitor entry routine
  - Mutual exclusion is rigidly enforced at monitor boundary
  - A thread that tries to enter monitor when it is in use must wait
- Threads return resources through monitors as well
  - Monitor entry routine calls **signal**
    - Alerts one waiting thread to acquire resource and enter monitor
    - Higher priority given to waiting threads than ones newly arrived
    - Avoids indefinite postponement
Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```plaintext
monitor monitor-name
{
  shared variable declarations
  procedure body P1 (...) {
    ...
  }
  procedure body P2 (...) {
    ...
  }
  procedure body Pn (...) {
    ...
  }
  {
    initialization code
  }
}
```

- Monitor is a programming language construct
  - It’s up to the compiler to implement the mutual exclusion on monitor entries (using binary semaphores in general)
  - The compiler, not the programmer, arranges for the mutual exclusion → less error prone

- Implementation must guarantee:
  1. Resource accessible only by monitor procedures
  2. Monitor procedures are mutually exclusive
     - At any time, only one process may be executing a procedure within a given monitor
     - Only one process (thread) may be inside a monitor at a time

Condition Variables

- Is mutual exclusion enough for our requirements?
  - (ex) In producer-consumer problem, what if the buffer is full and the producer has to block?
  - (ex) What if the buffer is empty and the consumer has to block?

- It does not provide any means for processes to communicate or synchronize with one another
  - Inside the monitor, use a conditional variable

- Introducing condition variables
  - wait & signal operations on the conditional variables
  - A process inside the monitor can block on the condition variables and allow another process to enter the monitor
Condition Variables

- For coordination, monitors provide:
  - **c.wait**
    - Calling process is blocked and placed on waiting queue associated with condition variable c
  - **c.signal**
    - Calling process wakes up the first process on c queue
    - When no process is waiting, signal is lost

- "condition variable" c is not a conventional variable
  - c has no value
  - c is an arbitrary name chosen by programmer to designate an event, state, or condition
  - Each c has a waiting queue associated
  - A process may "block" itself on c -- it waits until another process issues a signal on c

Conditional Variables

- When signal is called
  - Two processes eligible to be inside the monitor
    1. The process calling the signal
    2. Another process that waits on the conditional variable
  - Only one can be executing inside the monitor

- What should happen after a SIGNAL operation?
  - Hoare’s approach (Signal-and-continue monitor)
  - Hansen’s approach (Signal-and-exit monitor)

6.2.1 Condition Variables

- Signal-and-exit monitor (by Hansen)
  - Before a thread can reenter the monitor, the thread calling signal must first exit monitor
  - Requires thread to exit the monitor immediately upon signaling

- Signal-and-continue monitor (by Hoare)
  - Allows thread inside monitor to signal that the monitor will soon become available
  - Still maintain lock on the monitor until thread exits monitor
  - Thread can exit monitor by waiting on a condition variable or by completing execution of code protected by monitor
Hoare’s Semantics

Let the newly awakened process run, suspending the process that called signal.

The process that called signal is given the highest priority and reenters the monitor as soon as the reactivated process leaves the monitor.
Hansen’s Semantics

- A process doing a SIGNAL must exit the monitor immediately
- SIGNAL can appear only as the final statement in a monitor procedure
- Conceptually simpler and easier to implement

Bounded buffer problem

```c
monitor BoundedBuffer {
    char buffer[n];
    int nextin=0, nextout=0, fullCount=0;
    condition notempty, notfull;

deposit(char data) {
    ...
}

remove(char data) {
    ...
}
```

```c
deposit(char data) {
    if (fullCount==n) notfull.wait;
    buffer[nextin] = data;
    nextin = (nextin+1) % n;
    fullCount = fullCount+1;
    notempty.signal;
}

remove(char data) {
    if (fullCount==0) notempty.wait;
    data = buffer[nextout];
    nextout = (nextout+1) % n;
    fullCount = fullCount - 1;
    notfull.signal;
}
Conditional Variables

- wait on condition variable vs. wait on semaphore
  - Condition variables do not accumulate signals for later use the way semaphores do
  - If a condition variable is signaled with no one waiting for it, the signal is lost

6.2.2 Simple Resource Allocation with Monitors

- Thread inside monitor may need to wait outside until another thread performs an action inside monitor
- Monitor associates separate condition variable with distinct situation that might cause thread to wait
  - Every condition variable has an associated queue

```pseudocode
// Figure 6.1: Simple resource allocation with a monitor in pseudocode.
// ... (Pseudocode code snippet)
```
6.2.3 Monitor Example: Circular Buffer

- Circular buffer implementation of the solution to producer/consumer problem
  - Producer deposits data in successive elements of array
  - Consumer removes the elements in the order in which they were deposited (FIFO)
  - Producer can be several items ahead of consumer
  - If the producer fills last element of array, it must “wrap around” and begin depositing data in the first element of array

- Due to the fixed size of a circular buffer
  - Producer will occasionally find all array elements full, in which case the producer must wait until consumer empties an array element
  - Consumer will occasionally find all array elements empty, in which case the consumer must wait until producer deposits data into an array element
6.2.3 Monitor Example: Circular Buffer

Figure 6.2 Monitor pseudocode implementation of a circular buffer. (Part 2 of 2.)

```java
26 // monitor entry called by consumer to read data
27 monitorEntry void getChar( outputParameter slotData )
28 {
29   // wait on condition variable hasData if the buffer is empty
30   if ( occupiedSlots == 0 )
31     { wait( hasData ); // wait until hasData is signaled
32     } // end if
33   // read character from buffer into output parameter slotData
34   slotData = circularBuffer[ readPosition ];
35   occupiedSlots--; // one fewer slots has data
36   readPosition = ( readPosition + 1 ) % BUFFER_SIZE;
37   signal( hasSpace ); // signal that character has been read
38   } // end getChar
```

6.2.4 Monitor Example: Readers and Writers

- **Readers**
  - Read data, but do not change contents of data
  - Multiple readers can access the shared data at once
  - When a new reader calls a monitor entry routine, it is allowed to proceed as long as no thread is writing and no writer thread is waiting to write
  - After the reader finishes reading, it calls a monitor entry routine to signal the next waiting reader to proceed, causing a "chain reaction"

- **Writers**
  - Can modify data
  - Must have exclusive access
6.2.4 Monitor Example: Readers and Writers

Figure 6.3 Monitor pseudocode for solving the readers and writers problem. (Part 2 of 3.)

```plaintext
23 // reader entry called after reading
24 monitorEntry(void rReadLock)
25 { // readers: there are fewest readers
26    if (readers == 0) {
27      signal(2); // allow writer to proceed
28    }
29    // and if
30    } // end cond
31 // monitor entry called before performing write
32 monitorEntry( void wWriteLock)
33 { // call when readers are reading or if a writer is writing
34    if (writers == 0) {
35      signal(1); // wWriteLock
36    }
37 // use commit(); // wait until writing is allowed
38    // and if
39 } // end cond
```

6.2.4 Monitor Example: Readers and Writers

Figure 6.3 Monitor pseudocode for solving the readers and writers problem. (Part 3 of 3.)

```plaintext
41    wWriteLock = false; // lock out all readers and writers
42 } // end beginwrite
43 monitorEntry( void endWriteLock)
44 { // wWriteLock = false; // release lock
45    // if a reader is waiting to write, signal reader
46    if (queue.next == 1) {
47      signal(2); // rReadLock
48    }
49    // and if
50    // wWriteLock = false; // signal writer if no readers are waiting
51    { // use commit(); // see waiting writer can proceed
52      signal(1); // comwrite
53    } // and else
54 } // end endWriteLock
```