4. Monitors

The monitor concept was developed by Tony Hoare and Per Brinch Hansen in the early ‘70’s. (Same time period in which the concept of information hiding [Parnas 1972] and the class construct [Dahl et al. 1970] originated.)

Monitors support data encapsulation and information hiding and are easily adapted to an object-oriented environment.

4.1 Definition of Monitors

A monitor encapsulates shared data, all the operations on the data, and any synchronization required for accessing the data.

Object-oriented definition: a monitor is a synchronization object that is an instance of a special *monitor* class.
- A monitor class defines private variables and public and private access methods.
- The variables of a monitor represent shared data.
- Threads communicate by calling monitor methods that access shared variables.
4.1.1 Mutual Exclusion

At most one thread is allowed to execute inside a monitor at any time.
- Mutual exclusion is automatically provided by the monitor’s implementation.
- If a thread calls a monitor method, but another thread is already executing inside the monitor, the calling thread must wait outside the monitor.
- A monitor has an entry queue to hold the calling threads that are waiting to enter the monitor.

4.1.2 Condition Variables and SC Signaling

Condition synchronization is achieved using condition variables and operations \texttt{wait()} and \texttt{signal()}.

A condition variable \texttt{cv} is declared as

```
conditionVariable cv;
```

- Operation \texttt{cv.wait()} is used to block a thread (analogous to a \texttt{P} operation).
- Operation \texttt{cv.signal()} unblocks a thread (analogous to a \texttt{V} operation).

A monitor has one entry queue plus one queue associated with each condition variable. For example, Listing 4.1 shows the structure of monitor class \texttt{boundedBuffer}. Class \texttt{boundedBuffer} inherits from class \texttt{monitor}. It has five data members, condition variables named \texttt{notFull} and \texttt{notEmpty}, and monitor methods \texttt{deposit()} and \texttt{withdraw()}.

Fig. 4.2 is a graphical view of class \texttt{boundedBuffer}, which shows its entry queue and the queues associated with condition variables \texttt{notFull} and \texttt{notEmpty}.
class boundedBuffer extends monitor {
    public void deposit(...) { ... }
    public int withdraw (...) { ... }
    public boundedBuffer() { ... }

    private int fullSlots = 0; // # of full slots in the buffer
    private int capacity = 0; // capacity of the buffer
    private int [] buffer = null; // circular buffer of ints
    // in is index for next deposit, out is index for next withdrawal
    private int in = 0, out = 0;
    // producer waits on notFull when the buffer is full
    private conditionVariable notFull;
    // consumer waits on notEmpty when the buffer is empty
    private conditionVariable notEmpty;
}

Listing 4.1 Monitor class boundedBuffer.

Figure 4.2 Graphical view of monitor class boundedBuffer.

A thread that is executing inside a monitor method blocks itself on condition variable cv by executing cv.wait():
- releases mutual exclusion (to allow another thread to enter the monitor)
- blocks the thread on the rear of the queue for cv.

A thread blocked on condition variable cv is awakened by cv.signal();
- If there are no threads blocked on cv, signal() has no effect; otherwise, signal() awakens the thread at the front of the queue for cv.
- For now, we will assume that the “signal-and-continue” (SC) discipline is used. After a thread executes an SC signal to awaken a waiting thread, the signaling thread continues executing in the monitor and the awakened thread is moved to the entry queue; the awakened thread does not reenter the monitor immediately.
A: denotes the set of threads that have been awakened by \textit{signal()} operations and are waiting to reenter the monitor,

S: denotes the set of signaling threads,

C: denotes the set of threads that have called a monitor method but have not yet entered the monitor. (The threads in sets A and C wait in the entry queue.)

\textbf{=>} The relative priority associated with these three sets of threads is \( S > C = A \).

cv.\textit{signalAll()} wakes up all the threads that are blocked on condition variable \( cv \).
cv.\textit{empty()} returns \textit{true} if the queue for \( cv \) is empty, and \textit{false} otherwise.
cv.\textit{length()} returns the current length of the queue for \( cv \).

Listing 4.3 shows a complete \textit{boundedBuffer} monitor.

```java
class boundedBuffer extends monitor {
   private int fullSlots = 0; // number of full slots in the buffer
   private int capacity = 0; // capacity of the buffer
   private int[] buffer = null; // circular buffer of ints
   private int in = 0, out = 0;
   private conditionVariable notFull = new conditionVariable();
   private conditionVariable notEmpty = new conditionVariable();
   public boundedBuffer(int bufferCapacity ) {
      capacity = bufferCapacity;buffer = new int[bufferCapacity];
   }
   public void deposit(int value) {
      while (fullSlots == capacity)
         notFull.wait();
      buffer[in] = value; in = (in + 1) % capacity; ++fullSlots;
      notEmpty.signal(); //alternatively:if (fullSlots == 1) notEmpty.signal();
   }
   public int withdraw() {
      int value;
      while (fullSlots == 0)
         notEmpty.wait();
      value = buffer[out]; out = (out + 1) % capacity; --fullSlots;
      notFull.signal(); //alternatively:if (fullSlots == capacity–1) notFull.signal();
      return value;
   }
}
```

Listing 4.3 Monitor class \textit{boundedBuffer}.
Assume that the buffer is empty and that the thread at the front of the entry queue is Consumer_1 (C_1). The queues for condition variables *notFull* and *notEmpty* are also assumed to be empty (Fig. 4.4a).

When Consumer_1 enters method `withdraw()`, it executes the statement

```java
while (fullSlots == 0)
    notEmpty.wait();
```

Since the buffer is empty, Consumer_1 blocks itself by executing a `wait()` operation on condition variable *notEmpty* (Fig. 4.4b).

Producer_1 (P_1) then enters the monitor. Since the buffer is not full, Producer_1 deposits an item and executes `notEmpty.signal()`.

This signal operation awakens Consumer_1 and moves Consumer_1 to the rear of the entry queue behind Consumer_2 (C_2) (Fig. 4.4c).

After its `signal()` operation, Producer_1 can continue executing in the monitor, but since there are no more statements to execute, Producer_1 exits the monitor.
Consumer₂ now barges ahead of Consumer₁ and consumes an item. Consumer₂ executes `notFull.signal()`, but there are no Producers waiting so the signal has no effect.

When Consumer₂ exits the monitor, Consumer₁ is allowed to reenter, but the loop condition (`fullSlots == 0`) is true again:

```java
while (fullSlots == 0)
    notEmpty.wait();
```

Thus, Consumer₁ is blocked once more on condition variable `notEmpty` (Fig4.4d). Even though Consumer₁ entered the monitor first, it is Consumer₂ that consumes the first item.

This example illustrates why the `wait()` operations in an SC monitor are usually found inside while-loops: A thread waiting on a condition variable cannot assume that the condition it is waiting for will be true when it reenters the monitor.
4.2 Monitor-Based Solutions to Concurrent Programming Problems

These solutions assume that condition variable queues are First-Come-First-Serve.

4.2.4 Readers and Writers

Listing 4.10 is an SC monitor implementation of strategy R>W.1, which allows
concurrent reading and gives readers a higher priority than writers (see Section 3.5.4.)
Reader and writer threads have the following form:

```c
r_gt_w.1 rw;
```

Reader Threads:
```
rw.startRead(); /* read shared data */
rw.endRead();
```

Writer Threads:
```
rw.startWrite(); /* write to shared data */
rw.endWrite();
```

Writers are forced to wait in method `startWrite()` if any writers are writing or any readers
are reading or waiting.

In method `endWrite()`, all the waiting readers are signaled since readers have priority.

However, one or more writers may enter method `startWrite()` before the signaled readers
reenter the monitor. Variable `signaledReaders` is used to prevent these barging writers
from writing when the signaled readers are waiting in the entry queue and no more
readers are waiting in `readerQ`.

Notice above that the shared data is read outside the monitor. This is necessary in order to
allow concurrent reading.
class r_gt_w_1 extends monitor {
    int readerCount = 0;   // number of active readers
    boolean writing = false; // true if a writer is writing
    conditionVariable readerQ = new conditionVariable();
    conditionVariable writerQ = new conditionVariable();
    int signaledReaders = 0; // number of readers signaled in endWrite
    public void startRead() {
        if (writing) {       // readers must wait if a writer is writing
            readerQ.wait();
            --signaledReaders; // another signaled reader has started reading
        }
        ++readerCount;
    }
    public void endRead() {
        --readerCount;
        if (readerCount == 0 && signaledReaders==0)
            // signal writer if no more readers are reading and the signaledReaders
            // have read
            writerQ.signal();
    }
    public void startWrite() {
        // the writer waits if another writer is writing, or a reader is reading or waiting,
        // or the writer is barging
        while (readerCount > 0 || writing || !readerQ.empty() || signaledReaders>0)
            writerQ.wait();
        writing = true;
    }
    public void endWrite() {
        writing = false;
        if (!readerQ.empty()) { // priority is given to waiting readers
            signaledReaders = readerQ.length();
            readerQ.signalAll();
        }
        else writerQ.signal();
    }
}

Listing 4.10 Class r_gt_w_1 allows concurrent reading and gives readers a higher priority than writers.
4.3 Monitors in Java

Java’s wait, notify, and notifyAll operations combined with synchronized methods and user-defined classes enables the construction of objects that have some of the characteristics of monitors.

Adding synchronized to the methods of a Java class automatically provides mutual exclusion for threads accessing the data members of an instance of this class.

However, if some or all of the methods are inadvertently not synchronized, a data race may result. This enables the very types of bugs that monitors were designed to eliminate!

There are no explicit condition variables in Java. When a thread executes a wait operation, it can be viewed as waiting on a single, implicit condition variable associated with the object.

Operations wait, notify, and notifyAll use SC signaling:

- A thread must hold an object’s lock before it can execute a wait, notify, or notifyAll operation. Thus, these operations must appear in a synchronized method or synchronized block (see below); otherwise, an IllegalMonitorStateException is thrown.

- Every Java object has a lock associated with it. Methods wait, notify, and notifyAll are inherited from class Object, the base class for all Java objects.

- When a thread executes wait, it releases the object’s lock and waits in the “wait set” that is associated with the object:
  - A notify operation awakens a single waiting thread in the wait set.
  - A notifyAll operation awakens all the waiting threads.
  - Operations notify and notifyAll are not guaranteed to wake up the thread that has been waiting the longest.
4.7 A Monitor Toolbox for Java

A monitor toolbox is a program unit that is used to simulate the monitor construct. The Java monitor toolboxes are class `monitorSC` for SC monitors and class `monitorSU` for SU monitors.

Classes `monitorSC` and `monitorSU` implement operations `enterMonitor` and `exitMonitor`, and contain a member class named `conditionVariable` that implements `waitC` and `signalC` operations on condition variables.

A regular Java class can be made into a monitor class by doing the following:
1. extend class `monitorSC` or `monitorSU`
2. use operations `enterMonitor()` and `exitMonitor()` at the start and end of each public method
3. declare as many `conditionVariables` as needed
4. use operations `waitC()`, `signalC()`, `signalCall()`, `length()`, and `empty()`, on the `conditionVariables`.

Listing 4.22 shows part of a Java `boundedBuffer` class that illustrates the use of class `monitorSC`.

Simulated monitors are not as easy to use or as efficient as real monitors, but they have some advantages:
- A monitor toolbox can be used to simulate monitors in languages that do not support monitors directly, e.g., C++/Win32/Pthreads.
- Different versions of the toolbox can be created for different types of signals, e.g., an SU toolbox can be used to allow SU signaling in Java.
- The toolbox can be extended to support testing and debugging.
final class boundedBuffer extends monitorSC {
...
    private conditionVariable notFull = new conditionVariable();
    private conditionVariable notEmpty = new conditionVariable();
    ...
    public void deposit(int value) {
        enterMonitor();
        while (fullSlots == capacity)
            notFull.waitC();
        buffer[in] = value;
        in = (in + 1) % capacity;
        ++fullSlots;
        notEmpty.signalC();
        exitMonitor();
    }
    ...
}

Listing 4.22 Using the Java monitor toolbox class monitorSC.