**EE 382C/361C: Multicore Computing** **Fall 2016**

# Lecture 1: September 22

*Lecturer: Vijay Garg* *Scribe: Chuiye Kong*

## 1.1 Introduction

In this lecture, we first go over the puzzle of merging algorithm, then we discuss Monitor. Outline is shown as follows:

1. Sequential and two parallel solutions for Merging algorithm (puzzle)
2. Synchronization
3. Monitor
4. Problems using monitor

## 1.2 Merging Puzzle

As shown in Figure 1.1, the problem is to merge two sorted array into one sorted array.

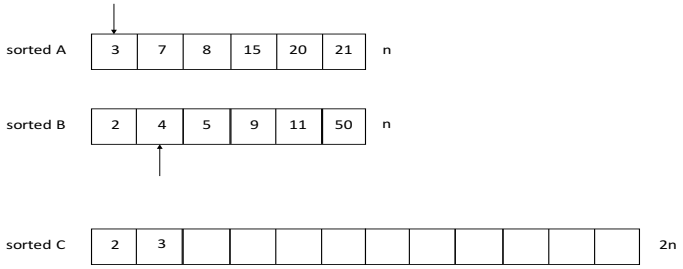


Figure 1.1: Merging Sorted Arrays

The sequential solution is using two pointers to point each array and compare the two pointed array items. The smaller item will be put into array C and its point will move to next element in the array. The timing complexity T(n) of this solution is O(n) and the work W(n) is O(n) as well.

### 1.2.1 Parallel Solution I

Select a key element in A (for example, we can choose 15), then array A will be split into left and right, which are annotated as A.left and A.right in Figure 1.2. Do the binary search for 15 (the key element we selected) in array B. B

will be divided up into the <15 part and >15 part, and allocated on the left side of the key element and on right side of it. The time complexity T(n) is O(log(n)log(n)) and work is O(nlog(n)log(n)).

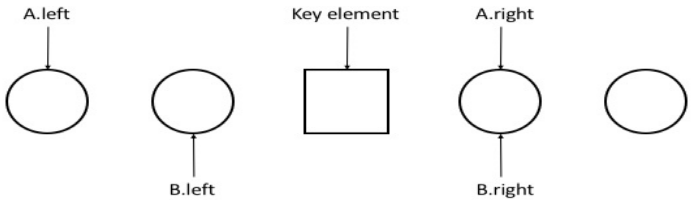


Figure 1.2: Binary Search for Key Element

### 1.2.2 Parallel Solution II

Now, we define rank(x), which is equal to the sum of the number of elements in A less than X and the number of elements in B less than X. For example, if we are looking for rank(A[j]), then the answer is j + binary search to find rank of A[j] in B, where j is the index in A as shown in Figure 1.3. The timing complexity of this algorithm is O(log) and the work is O(nlog(n)).

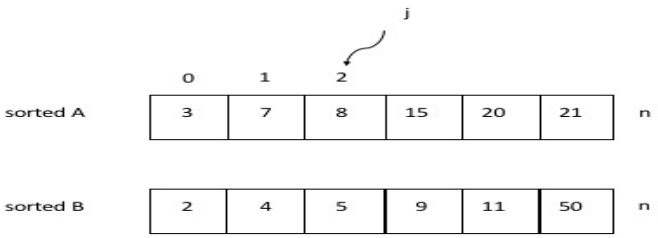


Figure 1.3: Rank example of A[j]

## 1.3 Synchronization

As far as we know, Synchronization can be used for mutual exclusion and conditional synchronization. To achieve mutual exclusion, waiting thread deploys two methods, which are busy-wait and sleep. Conditional synchronization is needed when a thread must wait for a certain condition to become true. Example is shown in Figure 1.4. We can see foo() has to do v() before T2 go to bar(). The drawbacks of using semaphore to achieve synchronization

can be summarized into two points. First, it is hard to tell the difference between mutual exclusion and conditional synchronization. Secondly, the order of acquire() matters, as it may cause deadlock (refer back to Semaphore lecture). It is not very reasonable to let programmers take care of this ordering problem. Therefore, we will introduce Monitor in the next section.

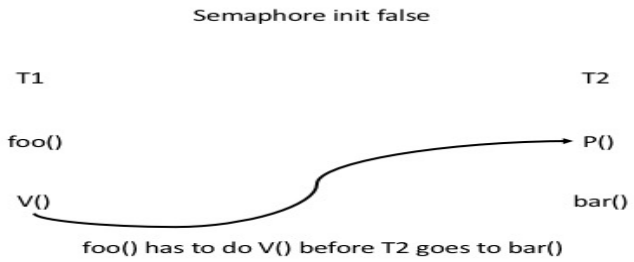


Figure 1.4: Conditional Synchronization Example

## 1.4 Monitor

We can use Monitor to replace Semaphore for both mutual exclusion and conditional synchronization purposes. There are two types of Monitors we can use, which are synchronized and reenetrant lock. A conditional variable has two operations: *wait* and *notify* (also called *signal*). In Java, the threads usually wait for the condition as shown in Algorithm 1.

**Algorithm 1** Conditional Wait.

```
1: while (!B) {
2:         wait();
3: }
```

*notify* is used to wake up one thread and *notifyAll* is used to wake up all threads in the waiting queue. *Wait* inserts the thread in the wait queue.
*Monitor Invariant* is defined as: at most one thread can be in the monitor at any given time. A example of monitor is shown in Figure 1.5.

Wait queue and condition queue may both have ready thread that can enter the monitor. The problem here is: which thread should continue after the notify operation. There are two possible answers:

1. One of threads that was waiting on the condition variable continues execution. Monitors that follow this rule are called *Hoare* monitors (or, *signal-and-wait monitors*).

2. The thread that made the notify call continues its execution. When this thread goes out of the monitor, then other threads can enter the monitor. This is the semantics followed in Java and is called *signal-and-continue*.

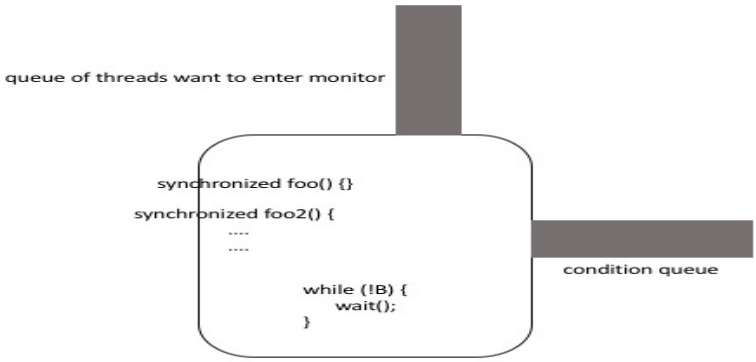


Figure 1.5: Monitor Example

## 1.5 Problems Using Monitor

### 1.5.1 Producer and Consumer

The buffer is shown in Figure 1.6. The BoundedBufferMonitor is used for one producer and one consumer, which has two entry methods: *deposit* and *fetch*. If a thread is executing the method *deposit* or *fetch*, then no other thread can execute *deposit* or *fetch*. The code of BoundedBufferMonitor is shown in [1]. This code works correctly only when there is exactly one producer and one consumer. The modified MBoundedBufferMonitor is designed to solve this issue and it can be found at [2].

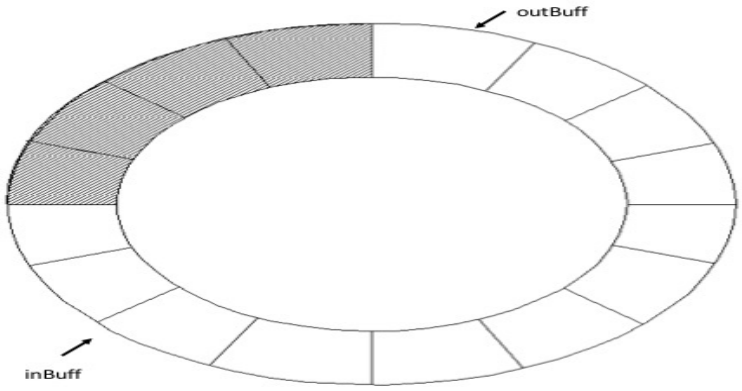


Figure 1.6: Producer and Consumer

### 1.5.2 Dining Philosophers

The code for using Monitor to solve Dining Philosophers is demoed in [3]. The code guarantees the mutual exclusion and deadlock-free, however, it does not guarantee freedom from starvation.

### 1.5.3 Nested Monitor

Nested monitor can cause deadlock problem when a thread waits for a condition inside a nested monitor. For example, a thread t1 is inside a monitor for an object *object1* and calls a synchronized method for another object *object2*. If the thread invokes a *wait()* inside it, then it will release the lock on *object2*. However, it will continue to hold the monitor lock of *object1*. If we have a thread t2 can notify thread t1, but thread t2 needs the lock of *object1*, a deadlock appears.

## References

[1] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore/blob/master/chapter3-synchronization_primitives/BoundedBufferMonitor.java

[2] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore/blob/master/chapter3-synchronization_primitives/MBoundedBufferMonitor.java

[3] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore/blob/master/chapter3-synchronization_primitives/DiningMonitor.java