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

Lecture 5: September 8

Lecturer: Vijay Garg *Scribe: Wenbo Xu*

5.1 Semaphore

5.1.1 Motivation

Recall from the Peterson's Algorithm:

```
while (wantCS[j] && (turn == i)){};
```

While threads are blocked to enter the critical section, they consistently check on the condition and running no-ops. This called busy wait and consume a lots of computing resources. With the help of OS, we can save the resource from busy wait.
The OS maintain two queues:

```
Running queue: contains all threads are runnable.
Blocked queue: contains all threads are blocked on other operations.
```

OS scheduler is responsible to put blocked thread from the running queue into blocked queue. And put unblocked queue from blocked queue to running queue. In this way, any threads are blocked would not doing busy waits and the computing resource can be used elsewhere.

5.1.2 Implementation

The concept of Semaphore is introduced by Dijkstra. It has two operations, and one variable

```
Value, indicate whether the CS can be entered
P(), acquire operation
V(), release operation
```

For P() operation, it first check Value to check if the lock is available. If yes, acquire the lock by setting it to false.

```
P(){
        while(Value){}; //no ops
        Value = false;
}
```

For V() operation, it release the lock by simply setting Value to true.

```
V(){
        Value = true;
}
```

The P() operation has to be atomic. And busy wait is used to implement this CS. So, the program use semaphore for its CS, and semaphore use busy wait for its CS. The program's CS is unknown length and is expected to be long, where we want to make is blocked to save computing resource.And Semaphore's CS is small, it is ok to do busy wait.
When a thread finished its CS, a method notify is used to weak up one which is blocked by the same semaphore.

5.1.3 Java Implementation

Here shows the Java code for BinarySemaphore

```
public class BinarySemaphore {
        boolean value;
        public BinarySemaphore(boolean initValue) {
        value = initValue;
        }
        public synchronized void P() {
        while (value == false)
        Util.myWait(this);// in the queue of blocked processes
        value = false;
        }
        public synchronized void V() {
        value = true;
        notify();
        }
}
```

The boolean value indicates whether the semaphore is ready or not. You can initial the semaphore to be ready or not. Unitl.myWait(this) in P() operation puts the thread into blocked queue. The V() operation calls notify() to weak up one thread in the blocked queue. The key work synchronized guarantees the operation is atomic.
The following code shows a CountingSemaphore:

```
public class CountingSemaphore {
        int value;
        public CountingSemaphore(int initValue) {
        value = initValue;
        }
        public synchronized void P() {
        while (value == 0) Util.myWait(this);
        value--;

        }
        public synchronized void V() {
        value++;
        notify();
        }
}
```

In this semaphore, the boolean value is replaced by a integer value. In this way, multiple threads are allowed

to be in CS at the same time.

5.1.4 Producer & Consumer

We have a producer thread and a consumer thread. The two threads share a circular buffer. This type of question has the following constraints:

· **mutual exclusion** shared resources

· **conditional synchronization** consumer must wait for the buffer to become not empty & producer must wait for the buffer to become not full

The following Java class shows how the bounded buffer is implemented.

```
class BoundedBuffer {
        final int size = 10;
        Object[] buffer = new Object[size];
        int inBuf = 0, outBuf = 0;
        BinarySemaphore mutex = new BinarySemaphore(true);
        CountingSemaphore isEmpty = new CountingSemaphore(0);
        CountingSemaphore isFull = new CountingSemaphore(size);

        public void deposit(Object value) {
                isFull.P();// wait if buffer is full
                mutex.P(); // ensures mutual exclusion
                buffer[inBuf] = value; // update the buffer
                inBuf = (inBuf + 1) % size;
                mutex.V();
                isEmpty.V();  // notify any waiting consumer
        }
        public Object fetch() {
                Object value;
                isEmpty.P(); // wait if buffer is empty
                mutex.P();  // ensures mutual exclusion
                value = buffer[outBuf]; //read from buffer
                outBuf = (outBuf + 1) % size;
                mutex.V();
                isFull.V(); // notify any waiting producer
                return value;
        }
}
```

A mutex semaphore is used to is used for mutual exclusion, it is a binary semaphore. Two counting semaphores are used for conditional synchronization. isEmpty is initialed to 0, and isFull is initaled to the size of the buffer.
The order of P() operations and V() operations for isFull, mutex and isEmpty is important. If mutex.P() is called first and mutex.V() is called the last, a producer thread can enter the CS section but wait on the buffer to become not full. At this time, the consumer is waited on mutex.P() to enter the CS so that it can notify the waiting producer to continue. This creates deadlock.

5.1.5 Reader & Writer

Problem is defined as Readers and Writers with a shared database. The constraints are:

· **no read write conflict**

· **no write write conflict**

· **multiple readers are okay**

The following Java class shows how the Reader and Writer is implemented:

```
class ReaderWriter {
        int numReaders = 0;
        BinarySemaphore mutex = new BinarySemaphore(true);
        BinarySemaphore wlock = new BinarySemaphore(true);
        public void startRead() {
                mutex.P();
                numReaders++;
                if (numReaders == 1) wlock.P();
                mutex.V();
        }
        public void endRead() {
                mutex.P();
                numReaders--;
                if (numReaders == 0) wlock.V();
                mutex.V();
        }
        public void startWrite() {
                wlock.P();
        }
        public void endWrite() {
                wlock.V();
        }
}
```

A binary semaphore, mutex, is used for mutual exclusion. Another binary semaphore, wlock, is used as lock for writer.
For the startWrite and endWrite method, the use of wlock ensures only one writer is writing at anytime. For Readers, we have to make sure no writer is writing. For the first reader, the wlock is acquired to make sure no writer can enter CS, and the wlock is released if there is no reader. The number of readers is increased & decreased in CS guaranteed by mutex.

5.1.6 Dining Philosophers Problem

The problem is defined as multiple philosophers (threads) are sitting in a round table. One chopstick is placed between every two philosophers. The philosopher needs both two chopsticks next to him to eat the food. The constraints are:

· **Two neighboring philosophers can not eat the same time**

The following Java class models the philosopher:

```
class Philosopher implements Runnable {
        int id = 0;
        Resource r = null;
        public Philosopher(int initId, Resource initr) {
                id = initId;
                r = initr;
                new Thread(this).start();
        }
        public void run() {
                while (true) {
                        try {
                                System.out.println("Phil " + id + " thinking");
                                Thread.sleep(30);
                                System.out.println("Phil " + id + " hungry");
                                r.acquire(id);
                                System.out.println("Phil " + id + " eating");
                                Thread.sleep(40);
                                r.release(id);
                        } catch (InterruptedException e) {
                                return;
                        }
                }
        }
}
```

Philosophers run in state of thinking, hungry to eat, eating, then back to thinking.
The following Java class implements the shared uses of the resource:

```
class DiningPhilosopher implements Resource {
        int n = 0;
        BinarySemaphore[] fork = null;
        public DiningPhilosopher(int initN) {
                n = initN;
                fork = new BinarySemaphore[n];
                for (int i = 0; i < n; i++) {
                        fork[i] = new BinarySemaphore(true);
                }
        }
        public void acquire(int i) {
                fork[i].P();
                fork[(i + 1) % n].P();
        }
        public void release(int i) {
                fork[i].V();
                fork[(i + 1) % n].V();
        }
        public static void main(String[] args) {
                DiningPhilosopher dp = new DiningPhilosopher(5);
                for (int i = 0; i < 5; i++)
                        new Philosopher(i, dp);
```

```
            }
    }
```

Use binary semaphore for each chopstick. For each philosopher tries to eat, it acquire chopstick from both side.When he finishes eating, release both chopsticks.
However, this implementation can lead to deadlock. For example, all the philosopher picks up the left chopstick at the same time, and has to wait for right chopstick.This is called the problem of symmetry.
There are some possible solutions to solve the problem:

- **Bring asymmetric** asking some philosophers to pick up right chopstick first, while others pick up left chopstick first.
- **Constrain the number of philosophers eligible to acquire is less than the max number**

References

[GITHUB] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore/