Producer/Consumer

CSCI 201
Principles of Software Development

Jeffrey Miller, Ph.D.
jeffrey.miller@usc.edu

Outline

- Producer/Consumer
- Program

Producer/Consumer Overview

- The producer/consumer problem is a famous problem for concurrent programming
- Assume you have a soda machine with a number of delivery people (producers) and a number of people wanting to buy sodas (consumers)
  - Each producer can insert as many sodas as he has, up to the capacity of the soda machine
  - Each consumer can purchase as many sodas as he wants, up to the current number in the soda machine
- If there are not enough sodas for a consumer to buy, he needs to wait until there are
- If there is not enough room for the producer to insert the number of sodas he has, he needs to wait until there is
- This problem can produce a situation called deadlock
- Download the `ProducerConsumerWithMonitors.java` from the course web site and execute it
  - What line(s) of code produce the potential deadlock?

Producer/Consumer Example

Put this one soda in the soda machine!

Is there a soda in

me to buy?

Producer/Consumer Example with Monitors

```java
import java.util.LinkedList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ProducerConsumerWithMonitors {
  private static Buffer buffer = new Buffer();

  public static void main(String [] args) {
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(new ProducerTask());
    executor.execute(new ConsumerTask());
    executor.shutdown();
  }

  private static class ProducerTask implements Runnable {
    public void run() {
      try {
        int i = 1;
        while (true) {
          System.out.println("Producer writes: " + i);
          buffer.write(i++);
          Thread.sleep((int)(Math.random() * 1000));
        }
      } catch (InterruptedException ie) {
        System.out.println("Producer IE: " + ie.getMessage());
      }
    }
  }

  private static class ConsumerTask implements Runnable {
    public void run() {
      try {
        while (true) {
          System.out.println("\t\tConsumer reads: " + buffer.read());
          Thread.sleep((int)(Math.random() * 1000));
        }
      } catch (InterruptedException ie) {
        System.out.println("Consumer IE: " + ie.getMessage());
      }
    }
  }
  private static class Buffer {
    private static final int CAPACITY = 1;
    private LinkedList<Integer> queue = new LinkedList<Integer>();
    private static Object notEmpty = new Object();
    private static Object notFull = new Object();
    public void write(int value) {
      synchronized(notFull) {
        synchronized(notEmpty) {
          try {
            while (queue.size() == CAPACITY) {
              System.out.println("Wait for notFull condition " + value);
              notFull.wait();
            }
            queue.offer(value);
            notEmpty.notify();
          } catch (InterruptedException ie) {
            System.out.println("Buffer.write IE: " + ie.getMessage());
          }
        }
      }
    }
    public int read() {
      int value = 0;
      synchronized(notFull) {
        synchronized(notEmpty) {
          try {
            while (queue.isEmpty()) {
              System.out.println("\t\tWait for notEmpty condition");
              notEmpty.wait();
            }
            value = queue.remove();
            notFull.notify();
          } catch (InterruptedException ie) {
            System.out.println("Buffer.read IE: " + ie.getMessage());
          }
        }
      }
      return value;
    }
  } // ends class Buffer
} // ends class ProducerConsumerWithMonitors
```

Producer/Consumer Example with Monitors

- Even if we reduce this to only having one monitor, there is a possible race condition.
- Assume the following execution
  - The Consumer checks to see if the queue is empty (line 64) and enters the loop.
  - The Consumer is switched out of the CPU in the middle of line 66, specifically after giving up the lock but before moving to the waiting state.
  - The Producer executes and adds a soda into the queue on line 53, then notifies any thread waiting on the `sodaMachine` (which is no one currently).
  - The Consumer then runs again and gets moved into the waiting state, waiting on the `sodaMachine` on line 66.
  - The Producer then runs again and finds that the queue is at capacity (line 49), so it waits (line 51).
  - We now have a deadlock state.

```
42   private static class Buffer {
43     private static final int CAPACITY = 1;
44     private LinkedList<Integer> queue = new LinkedList<Integer>();
45     private static Object sodaMachine = new Object();
46     public void write(int value) {
47       synchronized(sodaMachine) {
48         try {
49           while (queue.size() == CAPACITY) {
50             System.out.println("Wait for notFull condition " + value);
51             sodaMachine.wait();
52           }
53           queue.offer(value);
54           sodaMachine.notify();
55         } catch (InterruptedException ie) {
56           System.out.println("Buffer.write IE: " + ie.getMessage());
57         }
58       }
59     }
60     public int read() {
61       int value = 0;
62       synchronized(sodaMachine) {
63         try {
64           while (queue.isEmpty()) {
65             System.out.println("\t\tWait for notEmpty condition");
66             sodaMachine.wait();
67           }
68           value = queue.remove();
69           sodaMachine.notify();
70         } catch (InterruptedException ie) {
71           System.out.println("Buffer.read IE: " + ie.getMessage());
72         }
73       }
74       return value;
75     }
76   } // ends class Buffer
77 } // ends class ProducerConsumerWithMonitors
```

Producer/Consumer Example with Locks/Conditions

```java
import java.util.LinkedList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class ProducerConsumerWithLocks {
  private static Buffer buffer = new Buffer();
  public static void main(String [] args) {
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(new ProducerTask());
    executor.execute(new ConsumerTask());
    executor.shutdown();
  }
  private static class ProducerTask implements Runnable {
    public void run() {
      try {
        int i = 1;
        while (true) {
          System.out.println("Producer tries to write: " + i);
          buffer.write(i++);
          Thread.sleep((int)(Math.random() * 1000));
        }
      } catch (InterruptedException ie) {
        System.out.println("Producer IE: " + ie.getMessage());
      }
    }
  }
  private static class ConsumerTask implements Runnable {
    public void run() {
      try {
        while (true) {
          System.out.println("\t\tConsumer reads: " + buffer.read());
          Thread.sleep((int)(Math.random() * 1000));
        }
      } catch (InterruptedException ie) {
        System.out.println("Consumer IE: " + ie.getMessage());
      }
    }
  }
  private static class Buffer {
    private static final int CAPACITY = 1;
    private LinkedList<Integer> queue = new LinkedList<Integer>();
    private static Lock lock = new ReentrantLock();
    private static Condition notEmpty = lock.newCondition();
    private static Condition notFull = lock.newCondition();
    public void write(int value) {
      lock.lock();
      try {
        while (queue.size() == CAPACITY) {
          System.out.println("Wait for notFull condition " + value);
          notFull.await();
        }
        queue.offer(value);
        notEmpty.signal();
      } catch (InterruptedException ie) {
        System.out.println("Buffer.write IE: " + ie.getMessage());
      } finally {
        lock.unlock();
      }
    }
    public int read() {
      int value = 0;
      lock.lock();
      try {
        while (queue.isEmpty()) {
          System.out.println("\t\tWait for notEmpty condition");
          notEmpty.await();
        }
        value = queue.remove();
        notFull.signal();
      } catch (InterruptedException ie) {
        System.out.println("Buffer.read IE: " + ie.getMessage());
      } finally {
        lock.unlock();
        return value;
      }
    }
  } // ends class Buffer
} // ends class ProducerConsumerWithLocks
```

Producer/Consumer Output with Locks/Conditions

```
Producer trying to write: 1
                        Consumer reads: 1
Producer trying to write: 2
                        Consumer reads: 2
                        Wait for notEmpty condition
Producer trying to write: 3
                        Consumer reads: 3
Producer trying to write: 4
Producer trying to write: 5
Wait for notFull condition on 5
                        Consumer reads: 4
Producer trying to write: 6
Wait for notFull condition on 6
                        Consumer reads: 5
Producer trying to write: 7
Wait for notFull condition on 7
```

```
Producer trying to write: 1
                        Consumer reads: 1
                        Wait for notEmpty condition
Producer trying to write: 2
                        Consumer reads: 2
Producer trying to write: 3
                        Consumer reads: 3
Producer trying to write: 4
                        Consumer reads: 4
Producer trying to write: 5
Producer trying to write: 6
Wait for notFull condition on 6
                        Consumer reads: 5
Producer trying to write: 7
Wait for notFull condition on 7
                        Consumer reads: 6
Producer trying to write: 8
Wait for notFull condition on 8
                        Consumer reads: 7
Producer trying to write: 9
Wait for notFull condition on 9
                        Consumer reads: 8
Producer trying to write: 10
Wait for notFull condition on 10
                        Consumer reads: 9
Producer trying to write: 11
Wait for notFull condition on 11
```

Outline

- Producer/Consumer
- Program

Program

- Modify the Producer/Consumer program with locks so that a Producer will never write two lines in a row. Also provide an output line when a Producer or Consumer is notified or signaled.