Sep 20, 2006
Lecture 5: Process Synchronization II

October 2, 2006

1 NOTES ON READING

Today, we continue our discussion of syncronization from the last lecture.

After you have read Chapter 7, please start to Read Chapter 5 on Threads.

IMPORTANT NOTE ON READING: It is important to realize that in our lectures, we use a CONCEPTUAL description of the synchronization problems and their solutions, while the text tries to formulate the same issues using the Java language. BUT BE ASSURED that the underlying concepts are identical.

The advantage in the book's approach is that you have an unambiguous meaning in any proposed solution. And you can immediately use the solution in programs. The disadvantage is that Java may hide the conceptual simplicity of the solution witH extraneous java-specific features.

IN PARTICULAR, the final solution (Algorithm 3) to the MUTEX problem of Section 7.3 is the same as Peterson's solution in our lectures! You can judge for yourself which is conc.eptually easier to grasp (the one in our lecture notes or the Java program in the text).

2 Review

- 1. Q: Explain the following terms: mutual exclusion, critical section, semaphore, deadlock.

  A: A critical section is a contiguous portion of a process's code. It has the property that if two processors execute their critical section at the same time, error can occur.

  Mutual exclusion is the problem of ensuring that at most one process is executing its critical section at any moment in time.

  Semaphore is just a special SYSTEM WIDE global variable used for achieving synchronization (such as MUTEX).

  Deadlock is when the system has no useful work done because each process is waiting for another, but none can proceed.

  2. Q: Name the 4 properties required in a MUTEX solution.

  A: 1. MUTEX

  2. ARBITRARY SPEED ASSUMPTION

  3. NONBLOCKING (called "progress" in text)

  4. FAIRNESS (called "bounded waiting" in text)

  3. Q: "Deadlock-free synchronization may not be fair". Explain what this means.

  A: Deadlock-free means there is at least one process that is doing useful work. But if one process is constantly not allowed to make progress, it is unfair.

  So deadlock is a system-wide property, but fairness is a property for individual processes.

3 Intro

- Interprocess Communication is fairly general.
- We have seen the simplest problem, MUTEX.
- There are many other similar ”toy” problems. E.g. Dining Philosophers’ Problem.

  The point of each problem is to focus on a particular synchronization phenomena.
- Today, we generalize MUTEX to CONSUMER-PRODUCER problem. Our bank balance scenario fits this framework.

4 Producer-Consumer Problem

- The MUTEX SOLUTION CAN BE IMPLEMENTED as follows: IMAGINE that a variable S (”I want the C.S.”) can only be true or false. It also has an associated ”blocked queue”.

  Then P(S) sets S to true if it is false, otherwise blocks.

  Similarly, V(S) either wakes up a process blocking on its queue, or if there are no blocked processes, then it sets S to false.

  Modern OS, as we will see, provides such mechanisms.
- In MUTEX, the semaphore S is a binary variable. We now generalize S to be a counting variable, which holds any non-negative integer value.

  The consumer-producer problem can be viewed as a generalization of the MUTEX problem to counting variables.

  P(S) will block if S=0; otherwise, it decrements S.

  V(S) will block if S=maximum buffer size. Otherwise, it wake up some process blocked on S’s queue if any. If there are no blocked processes, it simply increments S.
- INSTANCES of such problems:

  Example: Producer=ttyin() keyboard interrupt service routine, Consumer=getchar().

  Example: Producer=ttyout() serial interface transmit interrupt, Consumer=putchar().

  Example: Our example of incrementing and decrementing a bank account balance is also an example.

  Example: Print spooler
- Think of integer variable S as the number of items in a buffer. Say the buffer have maximum size of N. So $0 \leq S \leq N$.

  Count variable, `0<=COUNT<=N`

```
Producer:
    loop forever:
       item= produce();
       if (COUNT == N) SLEEP();
       insert(item)
       COUNT++
       if (COUNT == 1) WAKEUP(Consumer)

Consumer:
    loop forever:
       if (COUNT == 0) SLEEP();
       item=remove(item)
       COUNT--
```

```
if (COUNT == N-1) WAKEUP(Producer)
consume(item)
```

WHAT ARE RACE CONDITIONS HERE?

None if there is only one producer and only one consummer. But suppose there are 2 consumers: Then we can have a race between the 2 consumer.

Similar if there are 2 producers, they could be racing (and overflow the buffer).

- Here is the JAVA View of the problem.

  First, we define the interface for buffers:

```
public interface Buffer {
    public abstract void insert(Object item); // insertion
    public abstract Object remove(); // removal
}
```

  Now we implement the interface:

```
import java.util.*;

public class BoundedBuffer implements Buffer {
    static final int BUFSIZE =5;
    int count; // number of items in buffer
    int in; // next free position
    int out; // next full position
    Object[] buffer;

    public BoundedBuffer() {
        count = in = out = 0;
        buffer = new Object[BUFSIZ];
    }
    public void insert(Object item) {
        while (count == BUFSIZE) ;
        ++count;
        buffer[in] = item; // produces item
        in = (in +1) % BUFSIZE;
    }
    public Object remove() {
        while (count == 0) ;
        –count;
        item = buffer[out]; // consumes item
        out = (out +1) % BUFSIZE;
        return item;
    }
}
```

- ANOTHER FORM OF SYNCHRONIZATION: the Buffer paradigm paradigm assumes that the producer and consumer shares a common memory (the Buffer). There is another scenario (e.g., distributed systems) where there is no common memory. Then we need the **message passing** paradigm.

  Instead of a Buffer, we now have a communication channel:

```
public interface Channel {
      public abstract void send(Object item); // sending
      public abstract Object receive(); // receiving
}
```

- MORE DISCUSSION ABOUT THE Consumer-Producer Problem (not discussed in class):

  * We now generalize the MUTEX problem to consider 2 classes of processes:

  o Producers, which produce items and insert them into a buffer. Producers are blocked when the buffer is full.

  o Consumers, which remove items and consume them. Consumer are blocked when the buffer is empty.

  * Instead of the binary semaphores of MUTEX problem, we now use counting semaphores S which hold integer values. Again, we support two operations P(S) and V(S).

  * V(S) is simply S++.

  * P(S) is the following "atomic operation":

  ```
       while (S>0) S--
  ```

  * Atomicity of P(S) should be clarified because the while-loop could encode any number of machine cycles! When we discover S¡=0, we are allowed to be interrupted. But when we discover S¿0, then the must be sure that there is no interruptions until S– is done.

  * The atomicity assumption implies that two processes doing P(S) cannot both see the same positive value of S (unless some intervening V(S) also occurs).

  ```
  initially S=k

  loop forever
      P(S) SCS   <== semi-critical-section V(S) NCS

  Initially e=k, f=0 (counting semaphore); b=open (binary semaphore)

  Producer                          Consumer

  loop forever                      loop forever
      produce-item                      P(f)
      P(e)                              P(b); take item from buf; V(b)
      P(b); add item to buf; V(b)       V(e)
      V(f)                              consume-item
  ```

  * k is the size of the buffer

  * e represents the number of empty buffer slots

  * f represents the number of full buffer slots

  * We assume the buffer itself is only serially accessible. That is, only one operation at a time.

  o This explains the P(b)–V(b) around buffer operations

  o I use ; and put three statements on one line to suggest that a buffer insertion or removal is viewed as one atomic operation.

  o Of course this writing style is only a convention, the enforcement of atomicity is done by the P/V.

  * The P(e), V(f) motif is used to force "bounded alternation". If k=1 it gives strict alternation.