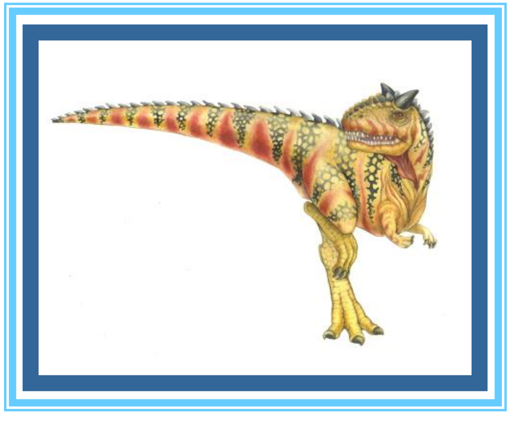
Chapter 7: Synchronization Examples



Outline

- Explain the bounded-buffer synchronization problem
- Explain the readers-writers synchronization problem
- Explain and dining-philosophers synchronization problems

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- The producer and consumer processes share the following data structures
  - Integer ***n*** to signifies buffer count
    - each can hold one item
  - Binary Semaphore **`mutex`** initialized to the value 1
    - provides mutual exclusion for accesses to the buffer pool
  - Counting Semaphore **`full`** initialized to the value 0
    - count the number of full buffers
  - Counting Semaphore **`empty`** initialized to the value n
    - count the number of empty buffers

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true) {
      ...
      /* produce an item in next_produced */
      ...
   wait(empty); // empty=0 => no space for new item
   wait(mutex);
       ...
      /* add next produced to the buffer */
       ...
   signal(mutex);
   signal(full);
}
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {
        wait(full); full=0 => nothing to consume
        wait(mutex);
           ...
         /* remove an item from buffer to next_consumed */
           ...
        signal(mutex);
        signal(empty);
           ...
          /* consume the item in next consumed */
           ...
        }
```

- Any symmetry between the producer and the consumer??

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - **Readers** – only read the data set; they do ***not*** perform any updates
  - **Writers** – can both read and write
- Readers-Writers Problem allows
  - multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- we require that the writers have exclusive access to the shared database while writing to the database
- Several variations of how readers and writers are considered – all involve some form of priorities
  - ***first*** readers–writers problem (writers may starve)
  - ***second*** readers–writers problem (readers may starve)
  - A solution to either problem may result in starvation

Readers-Writers Problem (Cont.)

- Shared Data
  - Data set
    - Shared among readers and writers
  - Binary Semaphore **`rw_mutex`** initialized to 1
    - common to both reader and writer processes.
      - mutual exclusion semaphore for the writers.
      - used by the first or last reader that enters or exits the critical section.
  - Binary Semaphore **`mutex`** initialized to 1
    - to ensure mutual exclusion when the variable read_count is updated
  - Integer **`read_count`** initialized to 0
    - variable keeps track of how many processes are currently reading the object

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
while (true) {
        wait(rw_mutex); // One writer or reader(s)
                           is writing or reading
            ...
        /* writing is performed */
            ...
    signal(rw_mutex);
}
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
while (true){
        wait(mutex);
        read_count++;
        if (read_count == 1) /* first reader */
             wait(rw_mutex);
        signal(mutex);
        ...
        /* reading is performed */
        ...
        wait(mutex);
        read_count--;
        if (read_count == 0) /* last reader */
               signal(rw_mutex);
        signal(mutex);
}
```

Readers-Writers Problem (Cont.)

- Let writer is in the critical section and *n* readers are waiting
  - one reader is queued on rw_mutex; rest *n*−1 readers are queued on mutex
- writer executes **signal(rw_mutex);** reader(s) or writer may be allowed.

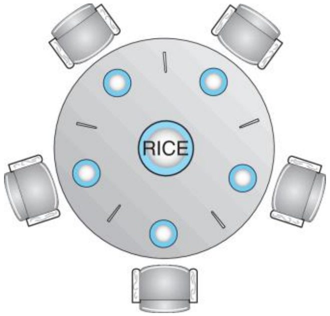
```
while (true){
        wait(mutex);
        read_count++;
        if (read_count == 1) /* first reader */
             wait(rw_mutex);
        signal(mutex);
        ...
        /* reading is performed */
        ...
        wait(mutex);
        read_count--;
        if (read_count == 0) /* last reader */
                signal(rw_mutex);
        signal(mutex);
}
```

Readers-Writers Problem Variations

- The solution in previous slide can result in a situation where a writer process never writes. It is referred to as the “First reader-writer” problem.
- The “Second reader-writer” problem is a variation the first reader-writer problem that state:
  - Once a writer is ready to write, no “newly arrived reader” is allowed to read.
- Both the first and second may result in starvation. leading to even more variations

Dining-Philosophers Problem

- N philosophers' sit at a round table with a bowel of rice in the middle.



- They spend their lives alternating thinking and eating.
- They do not interact with their neighbors.
- Occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done

Dining-Philosophers Problem

- The dining-philosophers problem is considered a classic synchronization problem neither because of its practical importance nor because computer scientists dislike philosophers but because it is an example of a large class of concurrency-control problems.
  - It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

- Semaphore Based Solution
- In the case of 5 philosophers, the shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- Philosophers (0,1,2,3,4)
- The structure of Philosopher i :

```
while (true){
    wait (chopstick[i] );  //left chopstick
    wait (chopStick[ (i + 1) % 5] ); //right chopstick

    /* eat for awhile */

    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );

    /* think for awhile */

}
```

- Algorithm guarantees no two neighbors are eating simultaneously.

Dining-Philosophers Problem Algorithm

- Philosophers (0,1,2,3,4)
- The structure of Philosopher i :

```
while (true){
     wait (chopstick[i] );  //left chopstick
     wait (chopStick[ (i + 1) % 5] ); //right chopstick

      /* eat for awhile */

     signal (chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );

      /* think for awhile */

}
```

- Algorithm guarantees no two neighbors are eating simultaneously.
- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm

- Philosophers (0,1,2,3,4)
- The structure of Philosopher i :

```
while (true){
     wait (chopstick[i] );  //left chopstick
     wait (chopStick[ (i + 1) % 5] ); //right chopstick

      /* eat for awhile */

     signal (chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );

      /* think for awhile */

}
```

- Algorithm guarantees no two neighbors are eating simultaneously.
- What is the problem with this algorithm?
  - Deadlock (Circular wait!!)

Deadlock Situation: Possible Remedies

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution
  - Odd-numbered philosopher picks up first her left chopstick and then her right chopstick
  - Even numbered philosopher picks up her right chopstick and then her left chopstick.
- A deadlock-free solution does not necessarily eliminate the possibility of starvation.

Monitor Solution to Dining Philosophers

- Monitor-based deadlock-free solution
  - This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.

Solution to Dining Philosophers (Cont.)

```
monitor DiningPhilosophers
{
   enum {THINKING, HUNGRY, EATING} state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait();
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
       (state[i] == HUNGRY) &&
       (state[(i + 1) % 5] != EATING)) {
         state[i] = EATING;
         self[i].signal();
      }
   }

   initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
    }
}
```

Solution to Dining Philosophers (Cont.)

- Each philosopher "i" invokes the operations **`pickup()`** and **`putdown()`** in the following sequence:

```
DiningPhilosophers.pickup(i);

    /** EAT **/

DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

End of Chapter 7