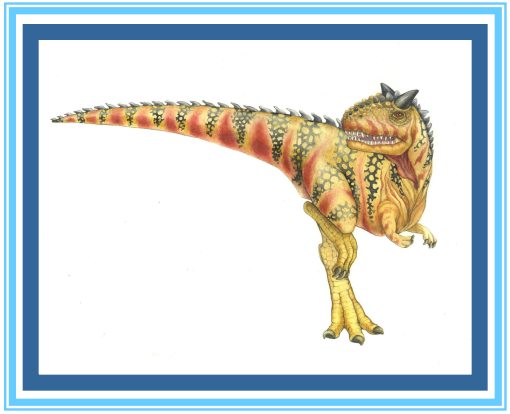
Chapter 7: Synchronization Examples



Outline

- Explain the **bounded-buffer** synchronization problem
- Explain the **readers-writers** synchronization problem
- Explain and **dining-philosophers** synchronization problems
- Describe the tools used by Linux and Windows to solve synchronization problems.
- Illustrate how POSIX and Java can be used to solve process 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

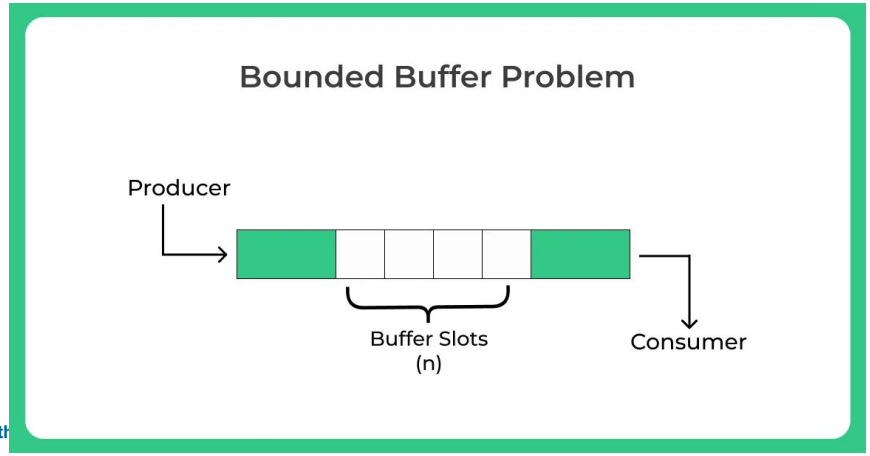
- The Bounded Buffer problem is also called the producer-consumer problem.

- The **Producer-Consumer** problem is a classical multi-process synchronization problem, that is we are trying to achieve synchronization between more than one process.
- There is one Producer in the producer-consumer problem,
  - Producer is producing some items, whereas
  - There is one Consumer that is consuming the items produced by the Producer.
- The same memory buffer is shared by both producers and consumers which is of fixed-size.

- The task of the Producer is to produce the item, put it into the memory buffer, and again start producing items. Whereas the task of the Consumer is to consume the item from the memory buffer

Bounded-Buffer Problem

- Below are a few points that considered as the problems occur in Producer-Consumer:
  - The producer should produce data only **when the buffer is not full**. In case it is found that the buffer is full, the producer is not allowed to store any data into the memory buffer.
  - Data can only be consumed by the **consumer if and only if the memory buffer is not empty**. In case it is found that the buffer is empty, the consumer is not allowed to use any data from the memory buffer.
  - **Accessing memory buffer** should not be allowed to producer and consumer **at the same time**.



Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
  - A binary semaphore used to acquire and release the lock
- Semaphore **`full`** initialized to the value 0
  - A counting semaphore which counts the full slots in the buffer
- Semaphore **`empty`** initialized to the value n
  - A counting semaphore which counts the empty slots in the buffer

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true) {
        ...
        /* produce an item in next_produced */
        ...
wait(empty); /Wait until empty>0 and then decrement empty
wait(mutex);/Acquire the lock so consumer cant change anything now
        ...
        /* add next produced to the buffer */
        ...
signal(mutex);/Release Lock
signal(full);/Increment full because one data is added to the slot
  so one less empty slot.
    }
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {
    wait(full); /Wait until full>0 and then decrement
full
    wait(mutex);//Acquire lock
        ...
     /* remove an item from buffer to next_consumed */
        ...
    signal(mutex); //Release lock
    signal(empty); //Increment empty
        ...
       /* consume the item in next consumed */
        ...
    }
```

Readers-Writers Problem

- The readers-writers problem relates to an object such as a file that is shared between multiple processes.
  - Some of these processes are readers i.e. they only want to read the data from the object and
  - Some of the processes are writers i.e. they want to write into the object.
- If two readers access the object at the same time there is no problem. However if two writers or a reader and writer access the object at the same time, there may be problems.
- To solve this situation, a writer should get exclusive access to an object
  - i.e. when a writer is accessing the object, no reader or writer may access it.
  - However, multiple readers can access the object at the same time.

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
- Problem – allow multiple readers to read at the same time
- Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities

Readers-Writers Problem (Cont.)

- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1: Common for both reader and writer
  - Semaphore **`mutex`** initialized to 1: Preserves mutual exclusion
  - Integer **`read_count`** initialized to 0: An integer variable that denotes how many readers are trying to read

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
while (true) {
        wait(rw_mutex);
            ...
        /* 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); /Lock is released for other
readers not for the writers
        ...
        /* 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
- **Problem is solved on some systems by kernel providing reader-writer locks**

Dining-Philosophers Problem

- The dining philosophers problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table and
- Their job is to **think and eat** alternatively.
- A bowl of noodles is placed at the center of the table along with five chopsticks for each of the philosophers.
- To eat a philosopher needs both their right and a left chopstick.
- A philosopher can only eat if both immediate left and right chopsticks of the philosopher is available.
- In case if both immediate left and right chopsticks of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again.
- The dining philosopher demonstrates a large class of concurrency control problems hence it's a classic synchronization problem.

Dining-Philosophers Problem

Dining-Philosophers Problem

- N philosophers' sit at a round table with a bowl 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
- In the case of 5 philosophers, the shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1(Which kind of semaphore and what do you mean by 1?)

Dining-Philosophers Problem Algorithm

- Semaphore Solution
- The structure of Philosopher i :

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

     /* eat for awhile */

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

     /* think for awhile */

}
```

- What is the problem with this algorithm?

Monitor Solution to Dining Philosophers

```
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 left and right neighbors
          test((i + 4) % 5);
          test((i + 1) % 5);
   }
```

Solution to Dining Philosophers (Cont.)

```
    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

Solution to Dining Philosophers (Cont.)

- The state array represents the state of each philosopher. There are three possible states: THINKING, HUNGRY, and EATING. It's an array of size 5 to represent 5 philosophers.

- The self array represents condition variables associated with each philosopher. These are used to allow a philosopher to wait until a certain condition is met (e.g., until they can start eating).

- The pickup function is called when a philosopher wants to pick up the forks to eat. It sets the philosopher's state to HUNGRY and then checks if the philosopher can start eating by calling the test function. If the philosopher is still not eating after testing, it waits on its associated condition variable.

Solution to Dining Philosophers (Cont.)

- The putdown function is called when a philosopher finishes eating and wants to put down the forks. It sets the philosopher's state to THINKING and then tests if its left and right neighbors can start eating by calling the test function for each neighbor.
- The test function is called to check if a philosopher can start eating. It checks if the philosopher's left and right neighbors are not eating (to ensure the forks are available), and if the philosopher itself is HUNGRY. If all conditions are met, the philosopher changes its state to EATING and signals its associated condition variable, allowing it to start eating.
- The initialization_code function initializes all philosophers' states to THINKING initially.

End of Chapter 7