CS 343 - Operating Systems

Module-3D
Classical Synchronization Problems

Dr. John Jose

Associate Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati

Session Outline

- ❖ Deadlock and Starvation Issues
- ❖ Bounded-Buffer Problem
- ❖ Readers and Writers Problem
- ❖ Dining-Philosophers Problem

Objectives of Process Synchronization

- To introduce the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- **To examine several classical process-synchronization problems**
- To explore several tools that are used to solve process synchronization problems

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let ***S*** and ***Q*** be two semaphores initialized to 1

P_0	P_1
wait(S);	**wait(Q);**
wait(Q);	**wait(S);**
...	**...**
signal(S);	**signal(Q);**
signal(Q);	**signal(S);**

```
wait(S)
{  while (S <= 0)
      ; // busy wait
      S--;
}
signal(S)
{  S++;
}
```

Deadlock and Starvation

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

P_0	P_1
wait(S);	**wait(Q);**
wait(Q);	**wait(S);**
...	**...**
signal(S);	**signal(Q);**
signal(Q);	**signal(S);**

Classical Problems of Synchronization

- ❖ Bounded-Buffer Problem
- ❖ Readers and Writers Problem
- ❖ Dining-Philosophers Problem

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **mutex** initialized to the value 1
- Semaphore **full** initialized to the value 0
- Semaphore **empty** initialized to the value n

Bounded-Buffer Problem

mutex (1), full (0), empty (n)

Producer process

```
do {
    /* produce an item in */
    wait(empty);
    wait(mutex);
    /* add item to the buffer */
    signal(mutex);
    signal(full);
} while (true);
```

Consumer process

```
do {
    wait(full);
    wait(mutex);
    /* remove an item from buffer */
    signal(mutex);
    signal(empty);
    /* consume the item */
} while (true);
```

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
- Allow multiple readers to read at the same time.
- Only one single writer can access the shared data at the same time
- Shared Data
  - Data set
  - Semaphore **rw_mutex** initialized to 1
  - Semaphore **mutex** initialized to 1
  - Integer **read_count** initialized to 0

Readers-Writers Problem

First Readers Writers Problem

It states that, once a **reader** is ready, then **readers** may read the file.

No **reader** should wait if a **reader** has access to the object, while the **writer** waits till the **reader** to complete it.

Writer starvation

Second Reader Writer Problem

It requires that, once a **writer** is ready, that **writer** performs its write as soon as possible.

If a **writer** is waiting to access the object, no new **readers** may start reading.

Reader starvation

First Readers-Writers Problem - Solution

Writer process

```
do {
    wait(rw_mutex);

    /* writing is performed */

    signal(rw_mutex);

} while (true);
```

Reader process

```
do {

    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
    /* reading is performed */
    wait(mutex);
    read count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);

} while (true);
```

Second Readers-Writers Problem - Solution

Writer process

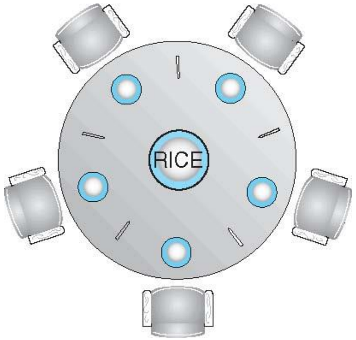
```
do {
    wait(mutex);lock=0;
    signal(mutex);

    wait(rw_mutex);
    /* writing is performed */
      signal(rw_mutex);

    wait(mutex);lock=1;
    signal(mutex);

} while (true);
```

Reader process

```
do { while(lock==0);
     wait(mutex);
     read_count++;
     if (read_count == 1)
          wait(rw_mutex);
     signal(mutex);
     /* reading is performed */
     wait(mutex);
     read count--;
     if (read_count == 0)
          signal(rw_mutex);
     signal(mutex);
 } while (true);
```

Dining-Philosophers Problem

- Philosophers spend their lives alternating thinking and eating
- Don't 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
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1



Dining-Philosophers Problem Algorithm

- The structure of Philosopher i:

```
do {
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );
            //  eat
    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );
            //  think
} while (TRUE);
```

What the limitations of this approach?

Dining-Philosophers Problem Algorithm contd..

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.)
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Monitor Solution to Dining Philosophers

```
monitor Dining Philosophers
{
    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.)

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

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

Sleeping Barbers Problem

- The analogy is based upon a hypothetical barber shop with one barber. There is a barber shop which has one barber, one barber chair, and n chairs for waiting for customers if there are any to sit on the chair.
- If there is no customer, then the barber sleeps in his own chair.
- When a customer arrives, he has to wake up the barber.
- If there are many customers and the barber is cutting a customer’s hair, then the remaining customers either wait if there are empty chairs in the waiting room or they leave if no chairs are empty.

Cigarette Smokers Problem

- There are four processes in this problem: three smoker processes and an agent process.
- Each of the smoker processes will make a cigarette and smoke it. To make a cigarette requires tobacco, paper, and matches. Each smoker process has one of the three items. ie, one process has tobacco, another has paper, and a third has matches.
- The agent has an infinite supply of all three. The agent places two of the three items on the table, and the smoker that has the third item makes the cigarette. Synchronize the processes.

johnjose@iitg.ac.in

http://www.iitg.ac.in/johnjose/