L20- CLASSICAL SYNCHRONIZATION PROBLEM

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

Second Reader Writer Problem

Writer process

```
do {
    wait(rw_mutex);

    /* writing is performed */

    signal(rw_mutex);

} while (true);
```

Reader process

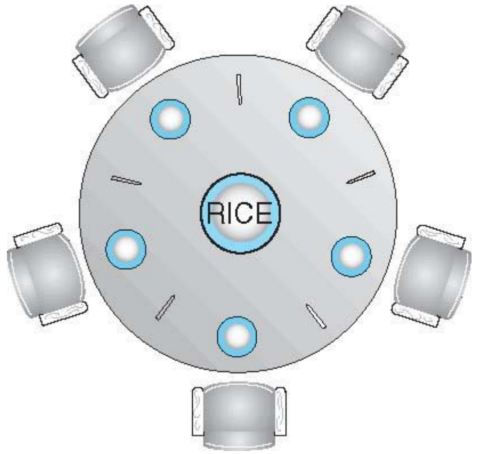
```
do {

    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);
```

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 chopsticks 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 () ;
        }
} }
```

Thank You