CISC 3320

C22a Process Synchronization: Classical Problems

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

Acknowledgement

- These slides are a revision of the slides provided by the authors of the textbook via the publisher of the textbook

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

- n buffers, each can hold one item
- A producer produces an item and inserts to the buffer
- A consumer removes an item from the buffer and consumes it.

Solution to Bounded-Buffer Problem

- Example solution using semaphores
  - A binary semaphore and two counting semaphores
  - Semaphore **`mutex`** initialized to the value 1
  - Semaphore **`full`** initialized to the value 0
  - Semaphore **`empty`** initialized to the value n

Producer Process

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

Consumer Process

```
while (true) {
      wait(full);
      wait(mutex);
           ...
       /* remove an item from buffer to next_consumed */
           ...
      signal(mutex);
      signal(empty);
           ...
       /* consume the item in next consumed */
           ...
  }
```

Questions?

- Producer-consumer problem
- Example solution using semaphores

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
  - Allow only one single writer to access the shared data at the same time, and
  - Variations

Variations of Readers-Writers Problem

- The first readers–writers problem
- The second readers–writers problem, and
- Other variations

The First Readers-Writers Problem

- Requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object.
- In other words, no reader should wait for other readers to finish simply because a writer is waiting.

The Second Readers-Writers Problem

- Requires that, once a writer is ready, that writer perform its write as soon as possible.
- In other words, if a writer is waiting to access the object, no new readers may start reading.

Solution to the First Readers-Writers Problem

- Shared Data
    - Data set
    - Semaphore **`rw_mutex`** initialized to 1
    - Semaphore **`mutex`** initialized to 1
    - Integer **`read_count`** initialized to 0

Writer P rocess

```
while (true) {
     wait(rw_mutex);
                    ...
     /* writing is performed */
                    ...
     signal(rw_mutex);
}
```

Reader Process

```
 while (true){
             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);

}
```

Readers-Writers Problem Variations

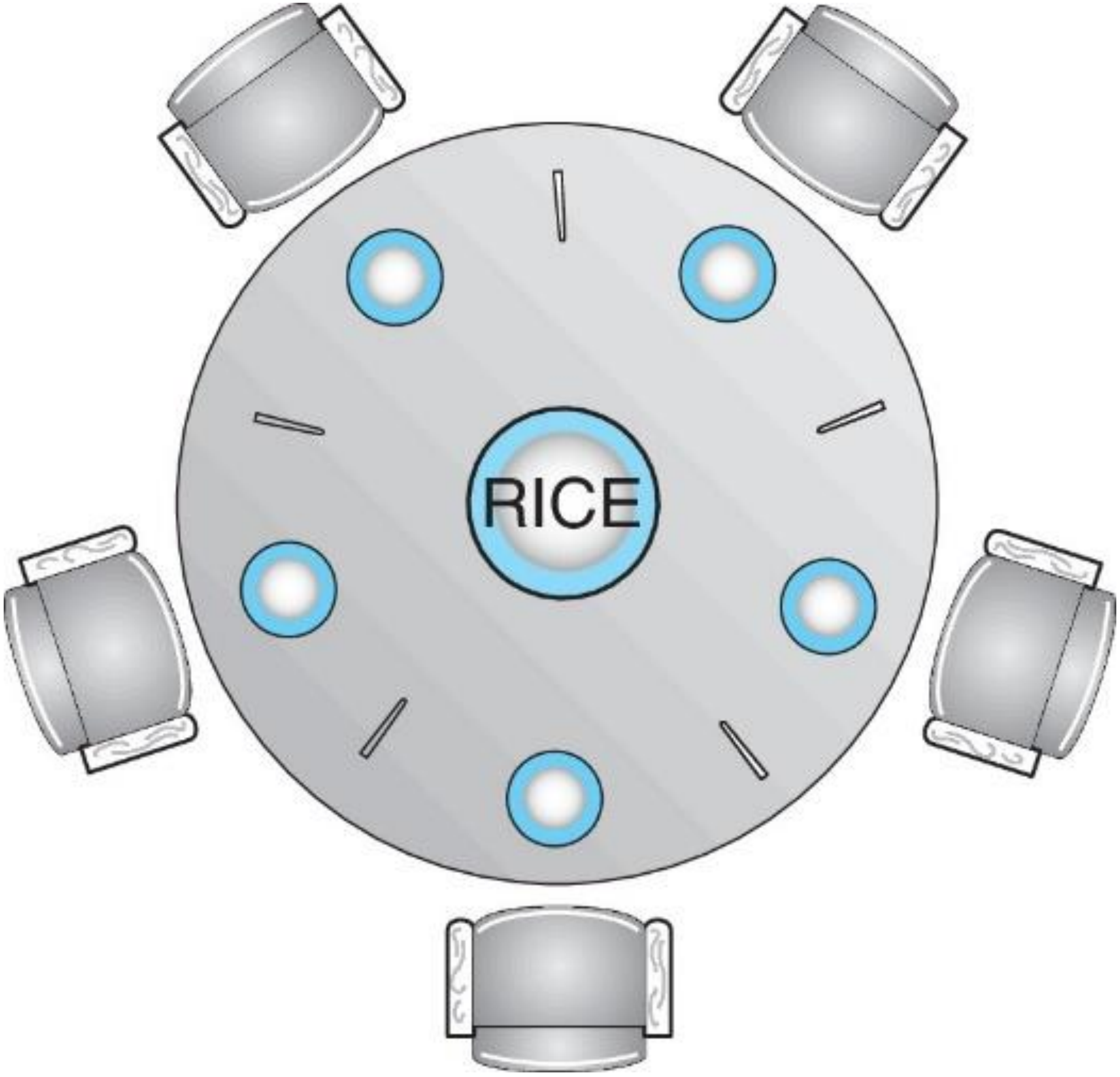
- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Questions?

- The Readers-Writers problem
- Variations of The Readers-Writers problem
- Solution to the First Readers-Writers problem.

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
- Starvation and deadlock


RICE

Solution to Dining-Philosophers Problem

- Assume there are 5 philosophers
- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

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?

Deadlock May Happen

- What if Process i and (i+1) both completes Line 2?

Monitor Solution to Dining Philosophers

- A deadlock free solution

Monitor Solution

```
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);
   }
```

Monitor Solution

```
   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;
    }
}
```

Starvation

- 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
- Dealing with starvation?

Questions?

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem
  - Semaphore solution
  - Monitor solution
  - Deadlock and starvation