CISC 7310X
C09b 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
- 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 */
           ...
       }
```

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
- 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

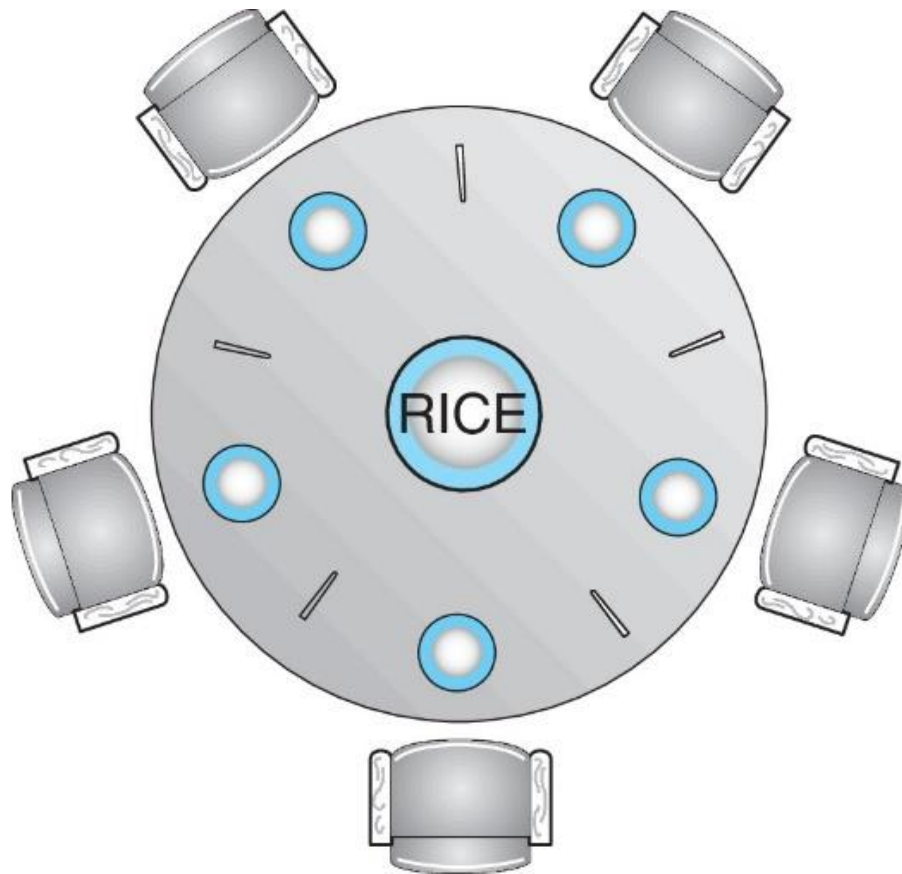
```
 while (true){
               wait(mutex);
               read_count++;
               if (read_count == 1) wait(rw_mutex);
               signal(mutex);

                 ...
               /* 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

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


RICE

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);
   }
```

```
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;
       }
}
```

- 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

Questions?

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