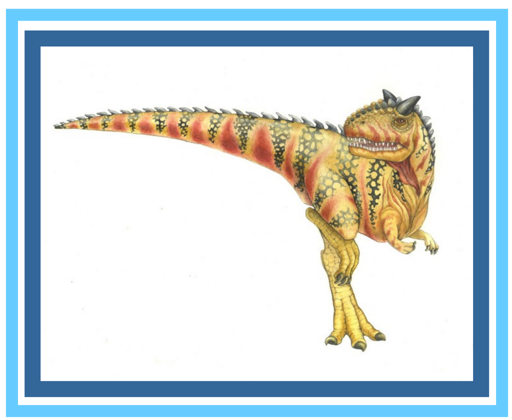
Chapter 7a: Synchronization Examples



Silberschatz, Galvin and Gagne ©2018

Outline

- Explain the bounded-buffer synchronization problem
- Explain the readers-writers synchronization problem
- Explain and dining-philosophers 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

- ***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 (Cont.)

- The structure of the producer process

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

Bounded Buffer Problem (Cont.)

- The structure of the 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

Readers-Writers Problem (Cont.)

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

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

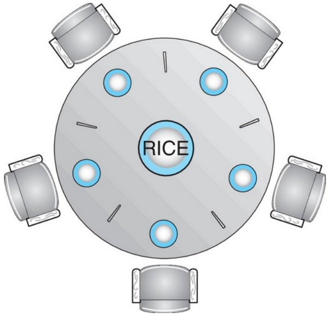
```
while (true){
        wait(mutex);
        read_count++;
        if (read_count == 1) /* first reader */
         wait(rw_mutex);
             signal(mutex);
         ...
        /* reading is performed */
         ...
        wait(mutex);
        read count--;
        if (read_count == 0) /* last 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

- N philosophers' sit at a round table with a bowel 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

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);
   }
```


Silberschatz, Galvin and Gagne ©2018

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:

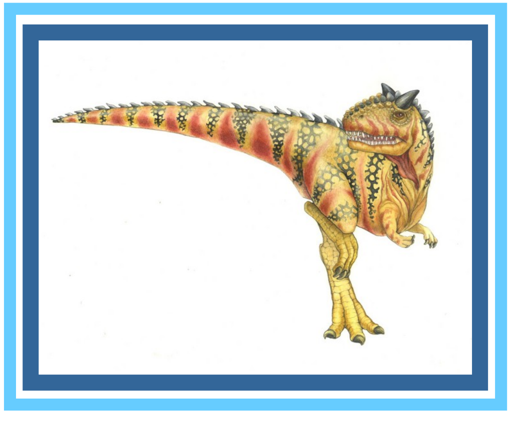
```
DiningPhilosophers.pickup(i);

    /** EAT **/

DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

End of Chapter 7a



Silberschatz, Galvin and Gagne ©2018