고려대학교 KOREA UNIVERSITY

KU-The Future

Operating Systems (10th Ed., by A. Silberschatz)

Chapter 7 Synchronization Examples

Heonchang Yu
Distributed and Cloud Computing Lab.

Objectives

- Explain the bounded-buffer, readers–writers, and dining–philosophers synchronization problems.

Semaphore

- Semaphore *S* – integer variable
- Two standard atomic operations : wait() and signal()
  - ✓ Originally called P() and V()

*locking*에 해당하는 것이 *wait()*
*unlocking*에 해당하는 것이 *signal()*

- Can only be accessed via two indivisible (atomic) operations

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

✓ signal (S) {
      S++;
   }
```

*semaphore*는 *db*에서 *concurrency control*
그리고 *two-phase locking protocol*과 유사

two-phase locking protocol : locking & unlocking

Classic Problems of Synchronization

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

Bounded-Buffer Problem

*queue*를 사용하는 프로세스들 간에 *synchronization* 작업 수행

- *n* buffers, each capable of holding one item
- Semaphore mutex *mutex :* 버퍼 접근 통제 *(*서로 다른 프로세스가 공유 자원을 한번에 접근하는 것 예방*)*
  - ✓ Provides mutual exclusion for accesses to the buffer pool
  - ✓ Initialized to the value 1
- Semaphore full
  - ✓ Count the number of full buffers
  - ✓ Initialized to the value 0
- Semaphore empty
  - ✓ Count the number of empty buffers
  - ✓ Initialized to the value n

Bounded Buffer Problem

- The structure of the producer process

```
while (true) {
        //   produce an item in next_produced
    wait (empty);
    wait (mutex);   mutex : 1 -> 0 값 변환
      //  add next_produced to the  buffer
    signal (mutex);
    signal (full);
}
```

Bounded Buffer Problem

- 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 database is to be shared among several concurrent processes.
  - ✓ Readers – only read the database; they do not perform any updates
  - ✓ Writers – can update the database
- Problem
  - *reader*가 많이 있는 것을 허용할 것인가?
  - ✓ Allow multiple readers to read at the same time.
  - ✓ Only one single writer can access the shared data at the same time.
  - ✓ If a writer and some other process (a reader or a writer) access the database simultaneously, chaos may ensue.

Readers-Writers Problem

- First solution
  - ✓ No reader is kept waiting unless a writer has already obtained permission to use the shared object.
  - ✓ Writers may starve
- Second solution
  - ✓ Once a writer is ready, that writer performs its write as soon as possible.
  - ✓ Readers may starve.

Readers-Writers Problem

- Shared Data
  - ✓ Semaphore mutex initialized to 1.
    - ❖ Used to ensure mutual exclusion when the variable read_count is updated

      writer가 실행될 때, 또 다른 writer가 실행되면 안됨 (writer의 mutual exclusion을 보장하는 것)
  - ✓ Semaphore rw_mutex initialized to 1.
    - ❖ A mutual-exclusion semaphore for writers
    - ❖ Used by the first or last reader that enters or exits the critical section
  - ✓ Integer read_count initialized to 0. reading process count를 하는 것
    - ❖ Keeps track of how many processes are currently reading the object

Readers-Writers Problem

- The structure of a writer process

```
while (true) {
        wait (rw_mutex) ;

          //    writing is performed

        signal (rw_mutex) ;
}
```

Readers-Writers Problem

- The structure of a reader process

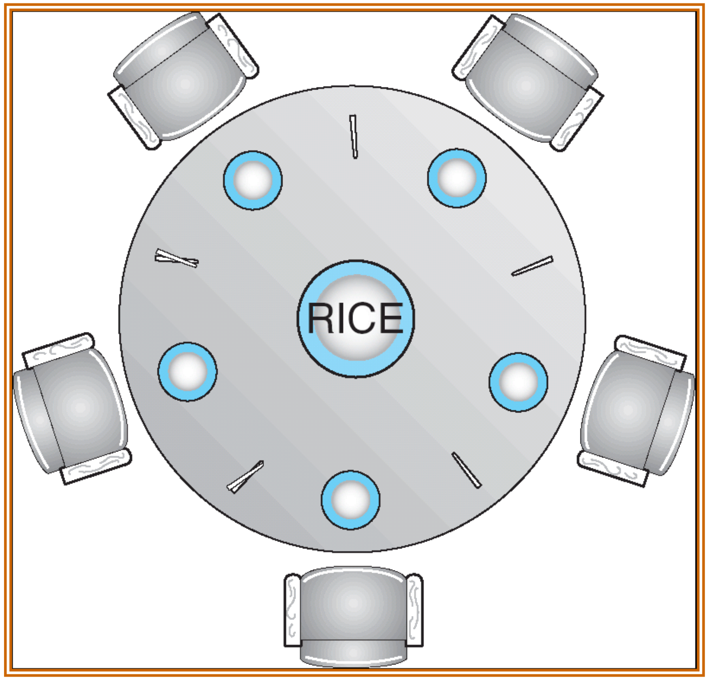
```
while (true)  {
        wait (mutex) ;
        read_count ++ ;   read_count : 0 -> 1로 변경하는 것
        if (read_count == 1)  wait (rw_mutex) ;
        signal (mutex)                  rw_mutex를 1-> 0으로 변경

                // reading is performed

        wait (mutex) ;
        read_count -- ;
        if (read_count  == 0)  signal (rw_mutex) ;
        signal (mutex) ;
}
```

Dining-Philosophers Problem

- Consider five philosophers who spend their lives thinking and eating.
- In the center of the table is a bowl of rice, and the table is laid with five single chopsticks. 젓가락이 한 짝 씩만 있는 경우임 (쌍이 안맞춰진 상태)
- A philosopher thinks, gets hungry, and tries to pick up the two chopsticks that are closest to her.
  - ✓ A philosopher may pick up only one chopstick at a time.



공유하는 자원을 적절하게 잘 생각해서 사용하는가?

Dining-Philosophers Problem

- One simple solution is to represent each chopstick with a semaphore.
  - ✓ A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore.
  - ✓ She releases her chopstick by executing the signal() operation on the appropriate semaphores.
- Shared data
  - ✓ Bowl of rice (data set)
  - ✓ Semaphore chopstick[5] initialized to 1 동시에 인접한 철학자들은 함께 식사를 할 수 없음
- Guarantees that no two neighbors are eating simultaneously

Dining-Philosophers Problem

- The structure of Philosopher *i* :

```
while (true) {
        wait( chopstick[i] );   1->0
        wait( chopStick[ (i + 1) % 5] );   1->0

             //  eat for a while   Eating (shared resource)

        signal( chopstick[i] );
        signal( chopstick[ (i + 1) % 5] );

            //  think for a while

}
```

Thinking

Hungry

Eating

Dining-Philosophers Problem

- Problem – deadlock 모든 프로세스가 *waiting*하는 상황이 *deadlock*
  - Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick.
- Several possible remedies to the deadlock problem
  - Allow at most four philosophers to be sitting simultaneously at the table
  - Allow a philosopher to pick up her chopsticks only if both chopsticks are available
  - Use an asymmetric solution
    - ❖ An odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick

1. 젓가락 수보다 *1*명 적은 철학자가 앉아 있는 경우

2. 양쪽 젓가락이 모두 가용 가능한 경우

3. 홀수 번호 철학자의 경우는 왼쪽 젓가락을 잡게 하고, 짝수 번호 철학자는 오른쪽 젓가락을 잡게하는 것

Dining-Philosophers Solution Using Monitors

- A deadlock-free solution to the dining-philosophers problem
- This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.
- This allows philosopher i to delay herself when she is hungry but is unable to obtain the chopsticks she needs.

```
condition self[5];
```

- Each philosopher, before starting to eat, must invoke the operation pickup().

- Ensures that no two neighbors are eating simultaneously.
- Ensures that no deadlocks will occur.
- It is possible for a philosopher to starve to death.

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((i + 4) % 5);
          test((i + 1) % 5);
    }
```

Solution to Dining Philosophers

```
    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;
    }
}
```