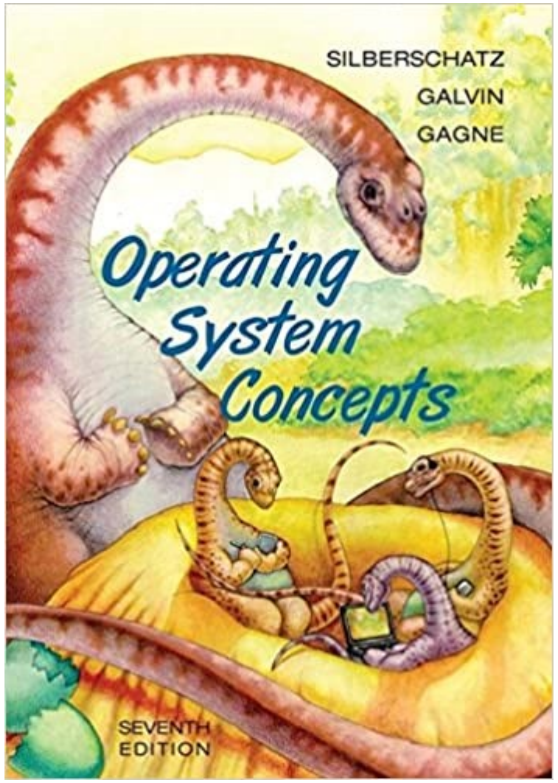
# Chapter 7: Synchronization Examples

School of Computing, Gachon Univ.
Joon Yoo




Gachon University
School of Computing

# Outline

- Classical Problems of Synchronization
  - Bounded-Buffer Problem
  - Reader-Writer problem
  - Dining philosopher problem

- Synchronization within the Kernel
  - Linux Synchronization
  - POSIX Synchronization

# The Bounded-Buffer Problem

- Producer thread
- Consumer thread

- N size buffer (Limited buffer)
  - Buffer can hold up to **N** items

PRODUCER

```
item   nextProduced;

while (TRUE) {
   while (counter == BUFFER_SIZE);
   buffer[in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;

   counter++;

}
```

CONSUMER

```
item  nextConsumed;

while (TRUE) {
   while (counter == 0);
   nextConsumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;

   counter--;
```

Critical section

| | full | empty |
|---|---|---|
| buffer full | N | 0 |
| buffer empty | 0 | N |

→ binary semaphore. → avoid mutual exclusion

- Solutions with semaphore – use 3 semaphores
  - Semaphore **mutex** = 1 // critical section
  - Semaphore **full** = 0 // number of items in buffer
  - Semaphore **empty** = N // number of empty slots in buffer
    - full + empty = N

↳ 비어있는 슬롯의 개수

Buffer (**N**)

| **full** | **empty** |
|---|---|

Buffer가 꽉 찼을 경우 full=N, empty=0 일거고, 반대의 경우 full=0, empty=N 일 것이다

# Bounded Buffer Problem (Cont.)

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

signal (S) {
     S++;
}
```

- The structure of the <u>producer thread</u>

```
do
{
        //   Produce an Item

        wait (empty);           // Check if buffer is full
        wait (mutex);           // Enter into Critical Section

     // Critical Section
     //  add an item to buffer

        signal (mutex);                        // Leave C.S.
        signal (full);                      // Produce an Item

} while (TRUE);
```

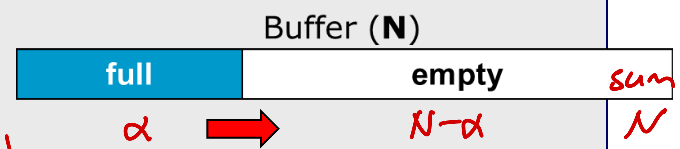
→ 0보다 작거나 같을 경우 무한루프, 0보다 크면 통과 →

empty ≤ 0, buffer full, wait!

empty > 0, → pass!

empty = empty − 1;

→ full = full + 1




| | full | empty | sum |
|---|---|---|---|
| | $\alpha$ | $N-\alpha$ | $N$ |
| wait(empty) | $\alpha$ | $N-\alpha-1$ | $N-1$ |
| signal(full) | $\alpha+1$ | $N-\alpha-1$ | $N$ |

# Bounded Buffer Problem (Cont.)

- The structure of the consumer thread

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

signal (S) {
    S++;
}
```

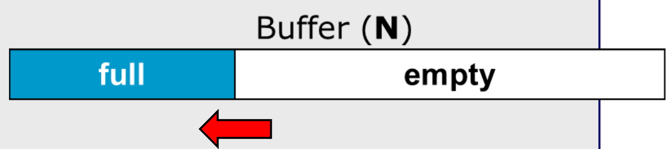
producer와 반대

```
do
{
        wait (full);          // Check if buffer is empty
        wait (mutex);         // Enter into Critical Section

      // Critical Section
      //  remove an item from  buffer

        signal (mutex);                 // Leave C.S.
        signal (empty);                 // Consume an 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.

  Read는 상관없음
  Write는 중요함!

- Problem
  - allow **multiple readers** to read at the same time.
  - **One single writer** can access the shared data at the same time.
  - How about writer and reader at the same time?

  writer이 들어오면 아무도 들어오지 못한다!

- Shared Data
  - Data set
  - Semaphore **wrt** initialized to 1. // mutual exclusion for writers and first/last reader
  - Semaphore **mutex** initialized to 1 // critical section to protect **readcount**
  - Integer **readcount** initialized to 0. // number of readers

  숫자상관 없음

# Readers-Writers Problem (Cont.)

- The structure of a <u>writer</u> process

```
do
{
        wait (wrt);

        //  writing is performed

        signal (wrt);

} while (TRUE);
```

writer만 있다고 생각

int wrt = 1;

wrt ≤ 0 → wait
> 0 → pass!

→ wrt--;    wrt=0

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

signal (S) {
     S++;
}
```

# Readers-Writers Problem (Cont.)

- The structure of a reader process

reader — 1) reader은 계속 들어와도 된다.
2) reader 실행중 writer이 들어올수 없다.
3) writer 실행중 reader이 들어올수 없다.

```
do {
        wait (mutex);
        readcount ++;
        if ( readcount == 1 )
                wait(wrt);
        signal(mutex)
```

- first reader (if ( readcount == 1 ))

- check if there a writer (wait(wrt);)

리더가 이미 있다면, writer이 없다는 소리이므로 wait(wrt);를 수행할 필요가 없다.

```
         // reading is performed

        wait(mutex);
        readcount --;
        if ( readcount == 0 )
                signal(wrt);
        signal (mutex);
} while (TRUE);
```

reader가 모두 나가야 writer을 실행할 수 있으므로, readcount == 0일 경우 signal(wrt);를 통해 새 wrt가 들어올 수 있도록 한다.

writer process

```
do
{
        wait (wrt);

        // writing is performed

        signal (wrt);

} while (TRUE);
```

# Final Exam'17

**1.** [4+3=7pts] Consider the readers/writers problem shown in right . Assume that the first request is a write **W1.** While **W1** is writing, the following requests arrive in the given order: **W2, R1, R2, W3, R3, W4.** (1) **In which order** will the above requests be processed? (2) **Which** readers will be reading *concurrently*? Assume that when a signal() is issued, the 1st process waiting in the semaphore is supposed to run.

```
do {      //reader process
          wait (mutex);
          readcount ++;
          if ( readcount == 1 )  wait(wrt);
          signal(mutex)
          // reading is performed
          wait(mutex);
          readcount --;
          if ( readcount == 0 )  signal(wrt);
          signal (mutex);
} while (TRUE);
```

```
do    //writer process
{
        wait (wrt);

        // writing is performed

        signal (wrt);

} while (TRUE);
```

④ R2 ⑥ R3 R1 ③

⑤ ⑦ ② W2 W3 W4

W1 ①

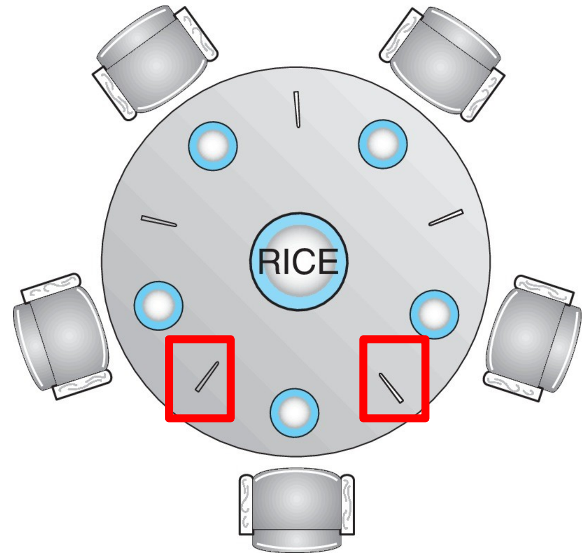
wait(wrt); 에 먼저 들어온 process가 실행

Answer:
(1) W2, R1, R2, R3, W3, W4
(2) R1, R2, R3

R1, R2가 signal(wrt)를 해주어야 W3 입장이 가능한데, R3은 wait(wrt)를 거치지 않으므로 ... W3이 뒤로 밀려난다.

# Dining Philosophers Problem

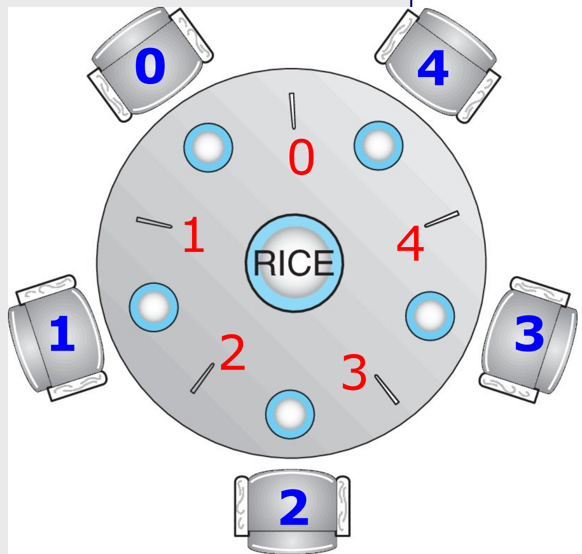
- *N* philosophers and *N* chopsticks
  - In right figure, 5 philosophers and 5 chopsticks
- Philosophers **eat**, **think**
  - Eating needs ***2*** chopsticks (from left and right)
- Pick up one chopstick at a time
  - First pick up left chopstick
  - Next pick up right chopstick
- Each chopstick used by one person at a time



# Dining-Philosophers Problem (Cont.)

- The structure of **Philosopher *i***:
  - Semaphore **chopstick [5]** all initialized to 1

```
do
{
        wait ( chopstick [i] );                  왼쪽 젓가락 + 오른쪽 젓가락
        wait ( chopstick [ (i+1)%5 ] );
        …
        //  eat
        …
        signal ( chopstick [i] );                젓가락 놓기
        signal ( chopstick [ (i+1)%5 ] );
        …
        // think
        …
} while (TRUE);
```

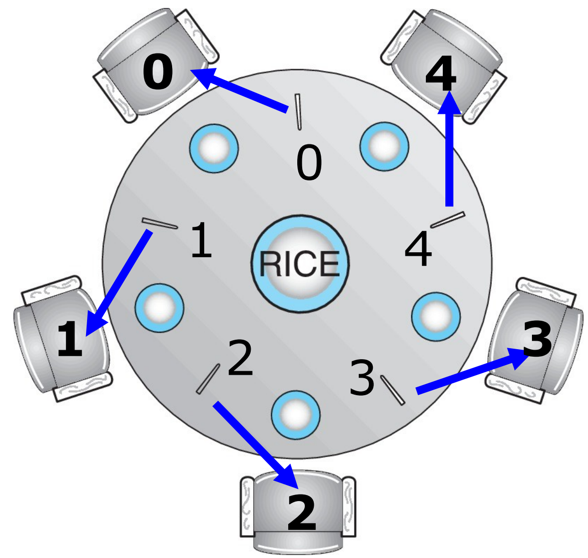


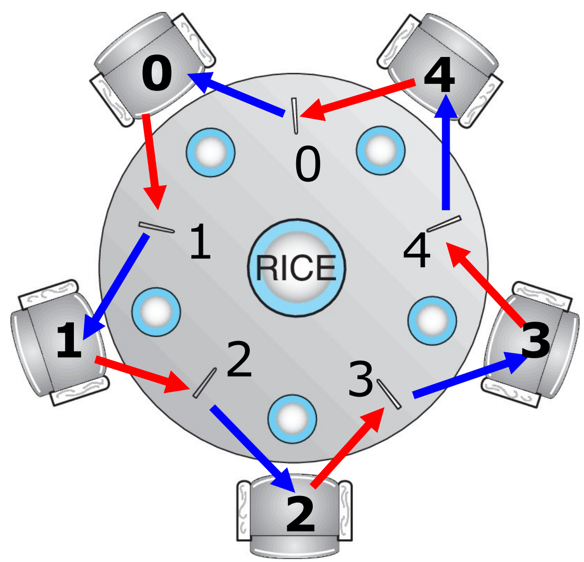
## Does this work ?

**NO? What is the problem?**

```
do
{
    wait ( chopstick [i] );
    wait ( chopstick [ (i+1)%5 ] );
    …
    // eat
    …
    signal ( chopstick [i] );
    signal ( chopstick [ (i+1)%5 ] );
    …
    // think
    …
} while (TRUE);
```

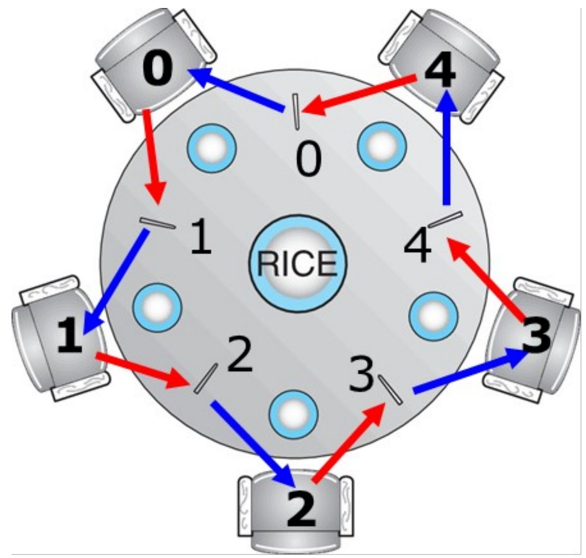
Progress X
deadlock!



# Dining-Philosophers Problem: Deadlock

- Deadlock
  - All five philosophers become hungry at the same
  - Each philosopher grabs the left chopstick – all chopstick semaphore will become 0
  - Each philosopher tries to grab he/her right chopstick, he/she will be delayed forever! – **deadlock!**

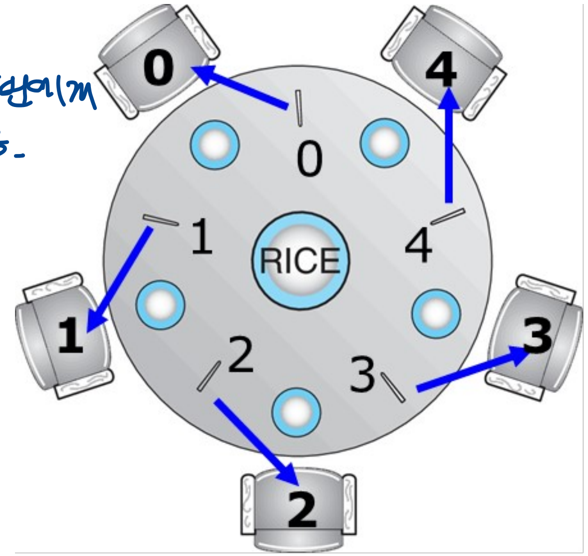


# Dining-Philosophers Problem: Solutions

- Allow at most four philosophers to be sitting simultaneously at the table → 4번을 없앨경우 4번 젓가락은 온전히 3번에게 달렸음.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available
  - To do this she must pick them up in a critical section
- Use an asymmetric solution
  - an odd-numbered philosopher picks up first her left chopstick – then her right chopstick
  - Whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick

짝수는 오른쪽, 홀수는 왼쪽을 먼저 가져가게 할 경우 deadlock 방지

- More in Ch. 8. Deadlock.



# Outline

- Classical Problems of Synchronization
  - Bounded-Buffer Problem
  - Reader-Writer problem
  - Dining philosopher problem

- Synchronization within the Kernel
  - Linux Synchronization
  - POSIX Synchronization

# Linux Synchronization

- Atomic variables

  **`atomic_t`** is the type for atomic integer
- Consider the variables

```
atomic_t counter;
int value;
```

All operations using atomic integers are performed without interruption

| *Atomic Operation* | *Effect* |
|---|---|
| `atomic_set(&counter,5);` | `counter = 5` |
| `atomic_add(10,&counter);` | `counter = counter + 10` |
| `atomic_sub(4,&counter);` | `counter = counter - 4` |
| `atomic_inc(&counter);` | `counter = counter + 1` |
| `value = atomic_read(&counter);` | `value = 12` |

→ 각 결과에서 preemptive 되지 않음!

# POSIX Mutex Locks

- Creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

Acquiring and releasing the lock

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

# POSIX Semaphores

- Creating an initializing the semaphore:

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

- Another process can access the semaphore by referring to its name **SEM**.
- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```


https://stock.adobe.com/kr/search?k=q%26a