Operating Systems

# 10. CPU: Synchronization (2)

Hyunchan, Park

http://oslab.chonbuk.ac.kr

Division of Computer Science and Engineering

Chonbuk National University

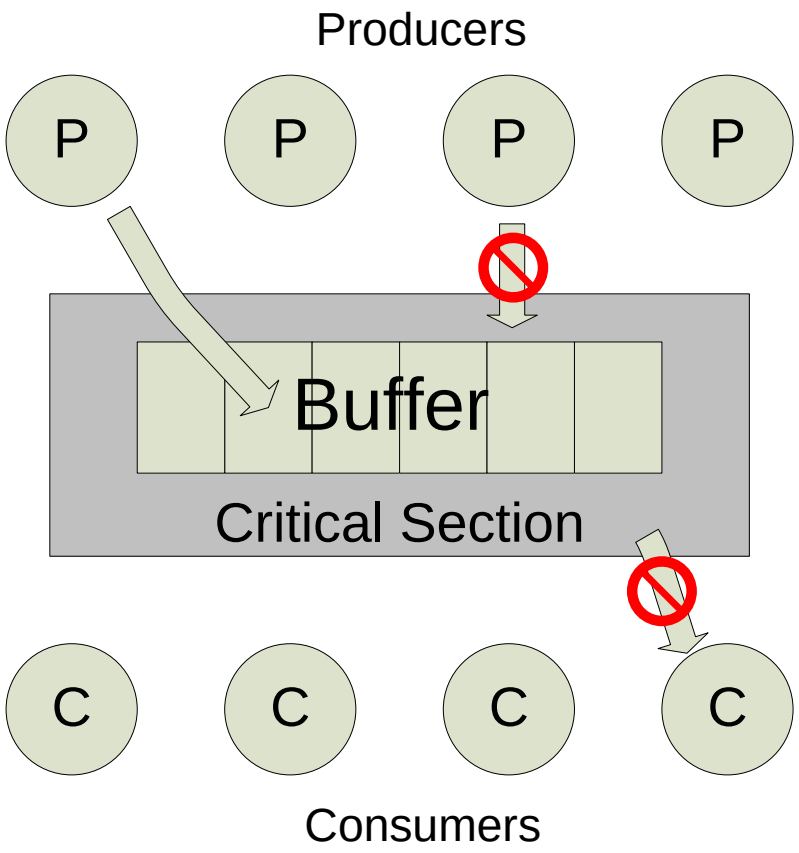
# Contents

- Classic Problems of Synchronization
  - Bounded-Buffer Problem
  - Readers-Writers Problem
  - Dining-Philosophers Problem
- Synchronization Examples
  - Windows
  - Linux
  - Pthreads

# Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
    - Solution 1,2,3
  - Dining-Philosophers Problem
    - Solution 1,2

# Bounded-Buffer Problem



# Bounded-Buffer Problem

- N개의 아이템을 삽입할 수 있는 버퍼에 여러 생산자와 여러 소비자가 접근
- 생산자 - 하나의 아이템을 생산해 버퍼에 저장
  - 동작 조건: 버퍼 내에 저장할 공간이 있는가?
    - Buffer의 상태가 full이면 대기
- 소비자 - 버퍼에서 하나의 아이템을 가져옴
  - 동작 조건: 버퍼 내에 소비할 아이템이 있는가?
    - Buffer의 상태가 empty이면 대기
- 여러 생산자·소비자 중 자신이 버퍼(critical section)에 접근하여 생산·소비할 수 있는가?

# Bounded-Buffer Problem

- Buffer의 구현
  - 1차원 배열로 구현
  - Boolean Buffer[n]; 으로 선언
  - 초기화
    - Buffer[0 … n-1] = not used;
- 생산자 operation
  - Buffer 배열 중, not used 인 index를 찾아 used 로 바꿈
  - Buffer[m] = used
- 소비자 operation
  - Buffer 배열 중, used 인 index를 찾아 not used 로 바꿈
  - Buffer[m] = not used

# Bounded-Buffer Problem: Solution

- 문제 해결을 위한 세마포어
  - Empty: 버퍼 내에 저장할 공간이 있음을 표시
    - 생산자의 진입을 관리
  - Full: 버퍼 내에 소비할 아이템이 있음을 표시
    - 소비자의 진입을 관리
  - Mutex: 버퍼에 대한 접근을 관리
    - 생산자, 소비자가 empty, full 세마포어를 진입한 경우, buffer의 상태 값을 변경하기 위한 세마포어
- 세마포어 value의 초기값
  - full = 0
  - empty = n  // buffer에 not used 인 entry가 n개
  - mutex = 1

# Bounded-Buffer Problem: Solution

- 생산자 프로세스

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

# Bounded-Buffer Problem: Solution

- 소비자 프로세스

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

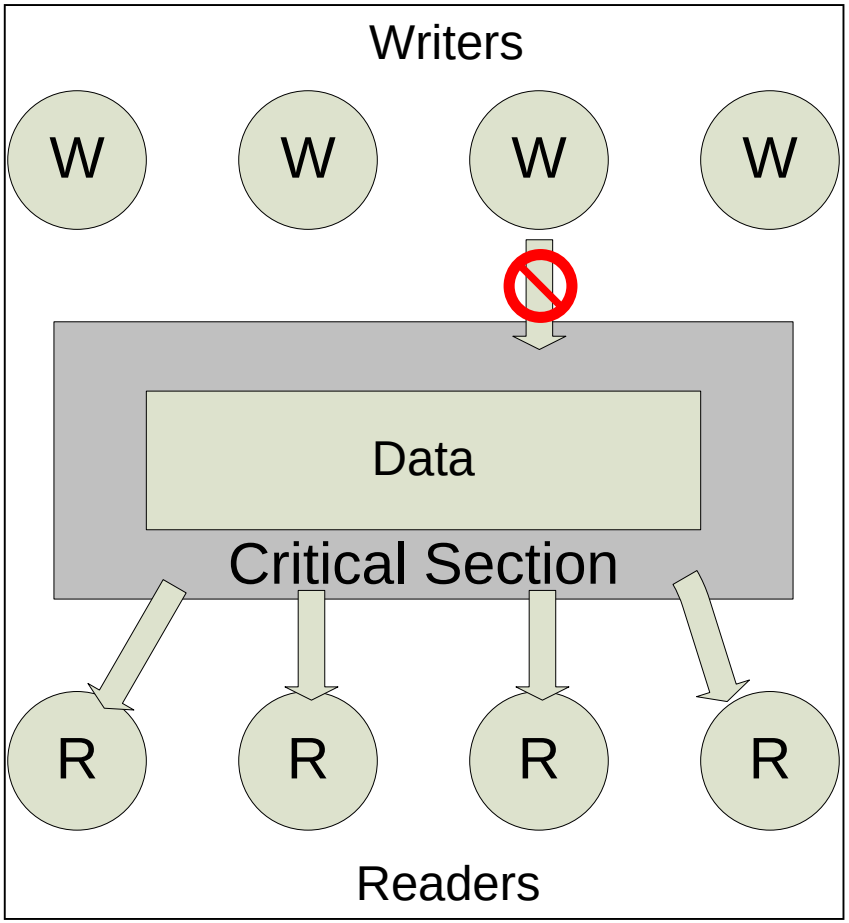
# Readers-Writers Problem

- 여러 Readers와 Writers가 하나의 공유 데이터를 읽거나 쓰기 위해 접근

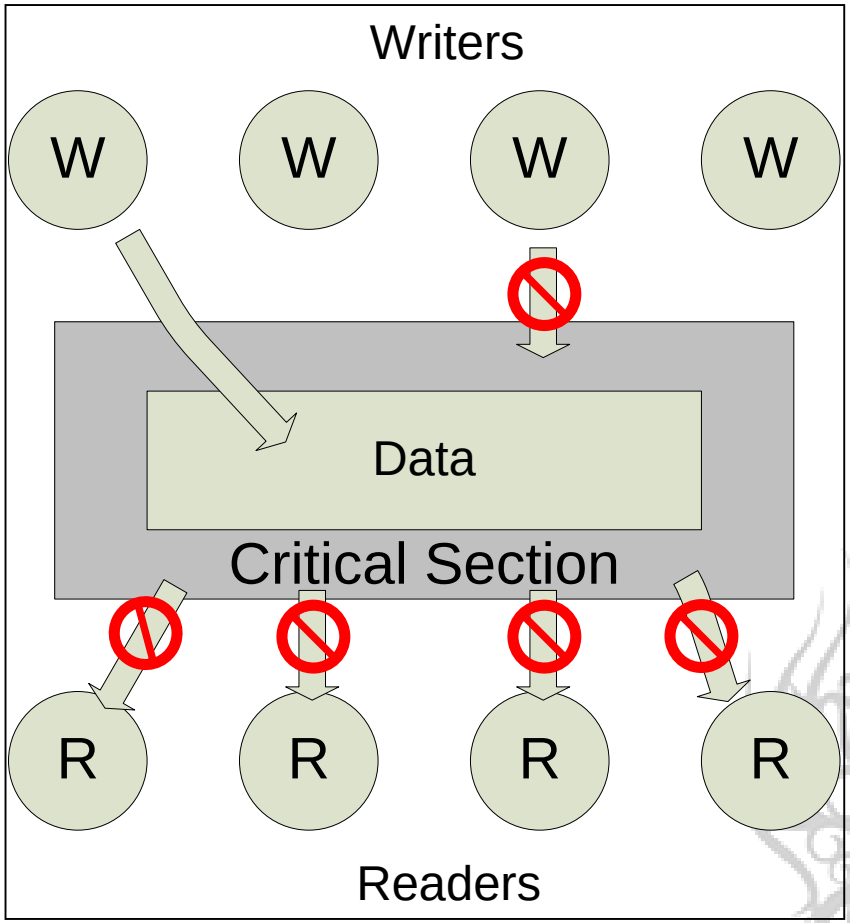
- Readers - 공유 데이터를 읽는다.
  - 여러 Reader는 동시에 데이터를 읽을 수 있다.

- Writers – 공유 데이터에 쓴다.
  - Writer가 데이터를 수정하기 위해서는, reader혹은 다른 writer가 작업을 하고 있지 말아야 한다.

# Readers-Writers Problem: Illustration




<여러 reader는 동시에 read가능>




<하나의 writer만 write가능>

# Readers-Writers Problem: Solution 1

- 문제 해결을 위한 자료구조와 세마포어
  - **Int Readcount :** 버퍼에 현재 몇 개의 Reader가 접근 중인지 표시
  - **Semaphore Wrt :** Writers 사이의 동기화 관리
  - **Semaphore Mutex :** Readcount와 wrt에의 접근이 원자적으로 수행됨을 보장하기 위한 세마포어

- 초기값
  - **mutex = 1, wrt = 1, readcount = 0**

# Readers-Writers Problem: Solution 1

- Writer 프로세스

```
do {
        wait(wrt);                  //entry section
                        ...
                writing is performed
                        ...
        signal(wrt);                //exit section
} while (true);
```

# Readers-Writers Problem: Solution 1

- Reader 프로세스 (do{…}while(true); 생략)

```
wait(mutex);
        readcount++;
        if (readcount == 1)
                wait(wrt);  //어떤 writer도 수행되고 있지 않음을 판별
signal(mutex);
        … reading is performed …
wait(mutex);
        readcount--;
        if (readcount == 0)
                signal(wrt);
signal(mutex);
```

# Readers-Writers Problem: Limitation of Solution 1

- Writer의 starvation
  - Readcount == 0 일 때만 signal(wrt)가 수행되어 wait(wrt)로 대기하고 있던 writer가 수행할 수 있음
  - 첫 번째 Reader(Readcount == 1)가 wait(wrt)만 통과하면, 다른 Reader들은 wait(mutex)에 대한 signal(mutex)만 기다리면 언제든지 수행할 수 있기 때문에 writer에 관계없이 계속해서 진입할 수 있다.
  - 여러 Reader들이 계속해서 진입할 경우, Readcount는 0까지 떨어지지 않을 수 있다.

- Writer의 starvation을 예방하는 solution이 존재함.

# Readers-Writers Problem: Solution 2

- Solution 1의 문제를 해결하는 방법
  - Writer가 수행하려고 때부터 Reader들의 진입을 막음
    - reader의 진입으로 인한 writer의 starvation 문제를 해결
    - 이미 수행중인 reader의 연산이 끝나면, writer의 수행 시작
- 문제 해결을 위한 자료구조와 세마포어
  - **Readcount :** 버퍼에 현재 몇 개의 Reader가 접근 중인지 표시
  - **Writecount :** 버퍼에 현재 몇 개의 Writer가 접근 중인지 표시
  - **Wrt :** Writers 사이의 동기화 관리
  - **Read :** Reader들의 진입을 제어하기 위한 세마포어
  - **Rmutex :** Readcount와 wrt에의 원자적 접근을 보장하기 위한 세마포어
  - **Wmutex :** Writecount와 read에의 원자적 접근을 보장하기 위한 세마포어
- 초기값
  - **rmutex, wmutex, wrt, read = 1**
  - **readcount, writecount = 0**

# Readers-Writers Problem: Solution 2

- **Writer process entry section**

```
wait(wmutex);
        writecount++;
        if (writecount == 1)
                wait(read);
signal(wmutex);
wait(wrt);
…writing is performed…
```

- **Writer process exit section**

```
…writing is performed…
signal(wrt);
wait(wmutex);
        writecount--;
        if (writecount == 0)
                signal(read);
signal(wmutex);
```

✓ wait(read) 를 수행함으로써 reader들이 계속해서 진입하는 것을 막고, 진입한 reader들이 읽기를 마치면 writer가 수행된다.

✓ signal(read) 를 수행함으로써 reader들이 다시 진입할 수 있도록 한다. 여기서 만약 writecount가 0으로 내려가지 않고 계속해서 writer가 쓰기 작업을 하려 한다면 reader들은 기아 상태에 빠진다.

# Readers-Writers Problem: Solution 2

- **Reader process entry section**

```
wait(read);
signal(read);
wait(rmutex);
        readcount++;
        if (readcount == 1)
                wait(wrt);
signal(rmutex);
…reading is performed…
```

- **Reader process exit section**

```
…reading is performed…
wait(rmutex);
        readcount--;
        if (readcount == 0)
                signal(wrt);
signal(rmutex);
```

✓ 첫 부분에서 wait(read), signal(read)가 번갈아가며 수행되므로, 여러 개의 reader가 함께 읽을 수 있다. 그런데 여기서 writer가 wait(read)를 수행하면 더 이상 signal(read)가 수행되지 않으므로 reader의 진입은 멈추게 된다.

✓ Reader가 더 이상 진입을 하지 않아 readcount가 0에 도달하면 공유데이터에 어떤 reader, writer도 없다는 것이 보장된다. 따라서 signal(wrt)를 통해 writer가 진입할 수 있도록 해준다.

# Readers-Writers Problem: Limitation of Solution 2

- Reader는 wait(read)에서 대기하는데, signal(read)가 보내지는 경우는 아래와 같다.
  - 처음 read가 1로 초기화되어 있을 경우
  - 수행하고자 대기중인 Writer가 하나도 없을 경우
  - 이미 진입한 Reader가 하나 있을 경우, 하나의 signal(read)가 수행되고, 그로 인해 진입한 Reader가 다시 signal(read)를 수행함으로써 연쇄적으로 Reader들이 진입할 수 있다.

# Readers-Writers Problem: Limitation of Solution 2

- 앞의 세 가지 경우를 살펴보면, 아래와 같이 Reader들이 Writer에게 수행을 넘기고, 기아상태에 빠질 수 있다.
  - Reader가 초기값을 이용해 먼저 진입했을 경우
    - 연쇄적으로 Reader들이 진입하다가 Writer가 수행을 위해 wait(read)에서 대기하면 signal(read) 하나가 Writer의 wait(read)를 풀게 되고, 이미 진입한 Reader가 모두 수행을 마치면 해당 Writer가 수행되면서 Writer로 수행이 넘어간다. 그리고 이 때는 Reader가 더 이상 진입할 수 없다.
  - Writer가 초기값을 이용해 먼저 진입했을 경우.
    - Writer가 수행을 시작하고, 계속해서 Writer들이 작업을 위해 대기한다면 결코 signal(read)는 수행되지 않고 Reader들이 기아 상태에 빠지게 된다. (Solution 1의 문제점과 반대의 상황)

# Readers-Writers Problem: Solution 3

- 모든 문제가 해결되어 동기화가 정확히 이루어지는 해결 방법
  - 수행을 기다리는 reader, writer를 분리해서 표시함으로써 어떤 작업이 수행 중일 때도 다른 reader, writer들이 대기할 수 있다. 또 대기 중인 작업이 있으면 그 작업을 수행하도록 함으로써 기아 상태를 방지한다.
- Writer의 동작
  - Writer는 어떤 작업이 수행되고 있거나, 대기중인 reader, writer가 있다면 쓰기 작업이 허가될 때까지 대기한다. 그렇지 않으면 쓰기를 수행한다.
  - Writer가 수행하고 나면 대기하고 있던 reader들을 모두 수행되게 한다. 대기중인 reader가 없고 대기중인 writer가 있다면 쓰기 작업을 수행하도록 한다.
- Reader의 동작
  - Reader는 writer가 기다리고 있거나 쓰기 작업이 수행 중이라면 대기한다.
  - 대기하고 있던 Reader가 모두 수행되면 대기하고 있던 writer를 하나 수행한다.

# Readers-Writers Problem: Solution 3

- 문제 해결을 위한 자료구조와 세마포어
  - **rc** : Critical section 내에 작업 중인 reader count
  - **wc** : Critical section 내에 작업 중인 writer count
  - **rwc** : Reader waiting count, 수행을 기다리고 있는 reader의 수
  - **wwc** : Writer waiting count, 수행을 기다리고 있는 writer의 수
  - **Wrt** : Writers 사이의 동기화 관리
  - **Read** : Readers 사이의 동기화 관리
  - **mutex** : 위 데이터와 세마포어에 대한 접근이 원자적으로 수행됨을 보장하기 위한 세마포어
- 초기값
  - **rc, wc, rwc, wwc = 0**
  - **wrt, read = 0**
  - **mutex = 1**

# Readers-Writers Problem: Solution 3

- **Writer process entry section**

```
wait(mutex);
    if(rc>0 || wc>0 ||
      rwc>0 || wwc>0) {
          wwc++;
          signal(mutex);
          wait(wrt);
          wait(mutex);
          wwc--;
    }
wc++;
signal(mutex);
…writing is performed…
```

- **Writer process exit section**

```
   …writing is performed…
wait(mutex);
        wc--;
        if (rwc>0) {

        for(i=0;i<rwc;i++)
                signal(read);
        } else
        if(wwc>0) signal(wrt);
signal(mutex);
```

# Readers-Writers Problem: Solution 3

- **Reader process entry section**

```
wait(mutex);
        if(wwc>0 || wc>0) {
                rwc++;
                signal(mutex);
                wait(read);
                wait(mutex);
                rwc--;
        }
rc++;
signal(mutex);
…reading is performed…
```

- **Reader process exit section**

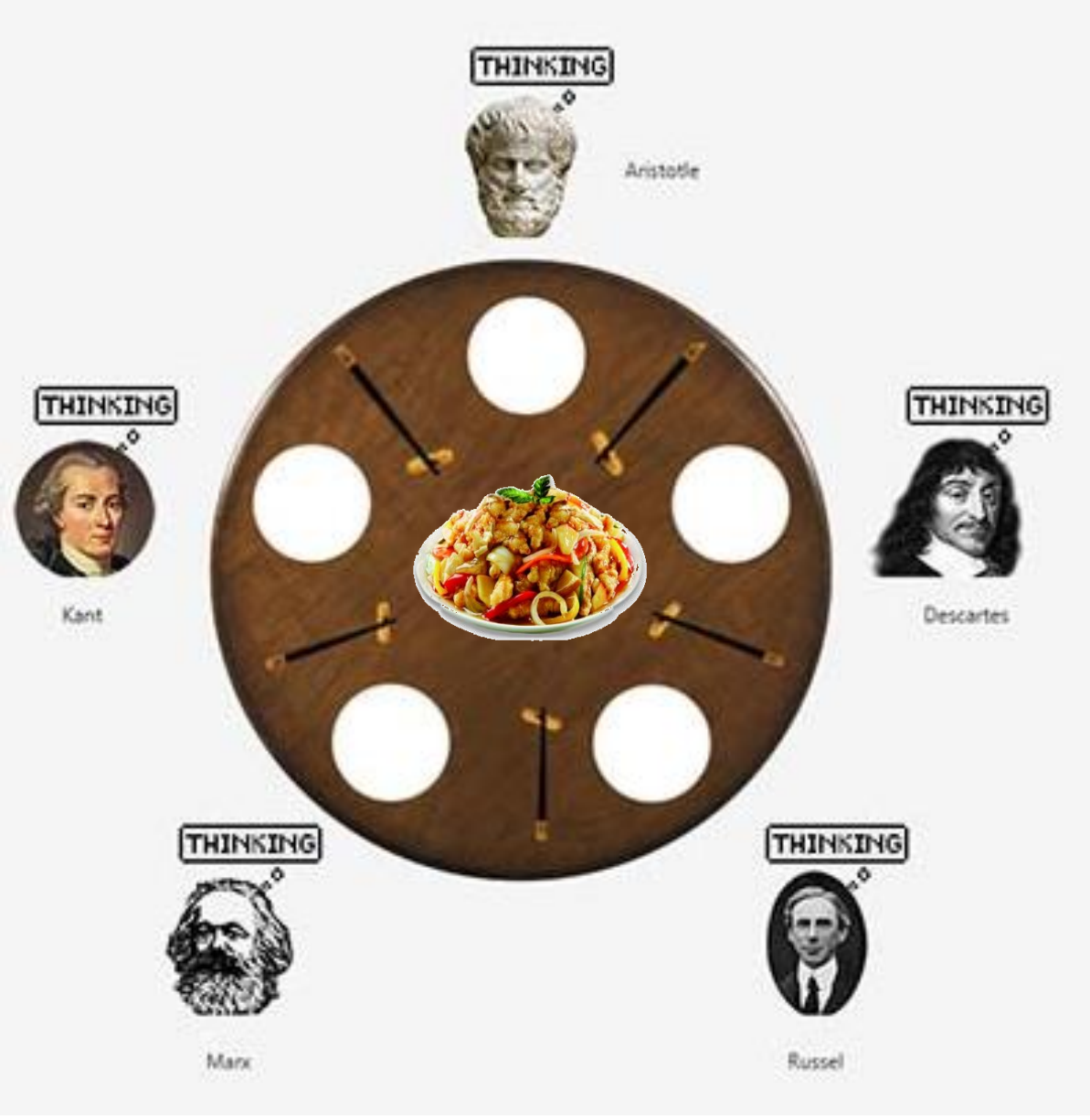
```
…reading is performed…
wait(mutex);
        rc--;
        if(rc==0 && wwc>0)
                signal(wrt);
signal(mutex);
```

# Dining-Philosophers Problem

- 고전적인 concurrency method
- 5명의 철학자가 한 식탁에서 함께 식사를 하는 상황
  - 젓가락이 5개뿐이다
  - 자신의 바로 옆 젓가락만 집을 수 있다.
  - 두 젓가락을 모두 집어야 식사를 할 수 있다.
  - 식사를 하고 난 다음에야 두 젓가락을 모두 내려놓는다.

- Deadlock과 Starvation이 발생하는 경우
  - 모두들 자신의 오른쪽 젓가락을 집었을 경우

# Dining-Philosophers Problem



# Dining-Philosophers Problem: Solution 1

- 단순히 젓가락을 집는 것에 대한 동기화를 부여하는 방법
  - 모든 철학자가 자신의 왼쪽 또는 오른쪽 젓가락부터 집도록 한다.

- 세마포어
  - **chopstick[5]** **: 각각의 젓가락에 대한 동기화 관리**

- 초기값은 모두 1

# Dining-Philosophers Problem: Solution 1

- **Philosopher process**

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

✓ 동시에 chopstick[i]를 잡으면 deadlock 발생

# Dining-Philosophers Problem: Solution 1

- Deadlock의 해결 방안
  - 한 번에 최대 4명의 철학자만 식탁에 앉도록 한다.
  - 젓가락의 상태를 미리 검사하여 양 쪽의 젓가락이 모두 사용 가능할 때만 젓가락을 집도록 한다.
  - 철학자에게 번호를 부여하여 홀수인 철학자는 왼쪽의 젓가락을, 짝수인 철학자는 오른쪽의 젓가락을 먼저 집도록 한다.

- 위의 해결방안들은 starvation까지 해결해주지는 못함
  - 각각의 방안에 대해 starvation에 대한 고려를 추가할 수 있다.
    - 예) 한 차례 굶은 철학자에게 우선권을 줌

# Dining-Philosophers Problem: Solution 2

- 양쪽의 젓가락을 한꺼번에 집는 방법
  - 젓가락의 상태를 미리 검사하여 양 쪽의 젓가락이 모두 사용 가능할 때만 젓가락을 집도록 하는 방법

- 사용되는 자료구조
  - **State[5] : 각 철학자의 상태를 기록 (THINKING, HUNGRY, EATING)**
- 문제 해결을 위한 세마포어
  - **mutex : 젓가락을 집거나 놓는 수행을 critical section으로 관리하기 위한 세마포어 (초기값: 1)**
  - **Self[5] : 철학자 각각이 젓가락 두 개를 잡기를 기다리는 세마포어**
    - **초기값: 모든 원소가 0**
    - **self[i] 의미는 철학자 i가 HUNGRY 상태이더라도, 다른 젓가락 하나를 사용 할 수 없을 경우 waiting을 하기 위한 세마포어**

# Dining-Philosophers Problem: Solution 2

- **Philosopher process**

```
do {
        …
        think
        …
    take_chopsticks(i);
        …
        eat
        …
    put_chopsticks(i)
        …
} while (1);
```

# Dining-Philosophers Problem: Solution 2

- **take_chopsticks(int i)**

```
{
    wait(mutex);
    state[i] = HUNGRY;
    test(i);
    signal(mutex);
    wait(self[i]);
}
```

✓ Mutex를 통해 진입하여, test(i)를 통해 양쪽의 철학자 상태를 검사한 후, 자신이 먹을 차례를 기다린다.

- **put_chopsticks(int i)**

```
{
    wait(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    signal(mutex);
}
```

✓ Mutex를 통해 진입하여, test(LEFT), test(RIGHT)를 통해 양쪽의 철학자 상태를 검사한 후, 먹을 차례를 기다리는 철학자에게 signal을 보내준다.
► test(i)에서 수행

# Dining-Philosophers Problem: Solution 2

- **test(int i)**

```
{
    if(state[i] == HUNGRY
    && state[LEFT] != EATING
    && state[RIGHT] != EATING) {
        state[i] = EATING;
        signal(self[i]);
    }
}
```

✓ Test를 수행한 철학자(i) 가 HUNGRY 상태이고,
양쪽의 철학자가 모두 젓가락을 집지 않은 상태(NOT EATING)이면,
take_chopsticks()에서 wait 했던 자신의 세마포어 self[i]에 signal 을 보내어 eat 로 진행하도록 한다.

# Dining-Philosophers Problem: Solution 2

- Solution2의 전략은 철학자 좌우 젓가락이 사용 가능할 때 critical section에 진입한다.
  - i 번째 철학자가 식사를 하기 위해서는 i-1(LEFT), 과 i+1(RIGHT) 철학자가 EATING 상태가 아니어야 한다.

- take_chopstick()과 put_chopstick()의 동작 상세 설명

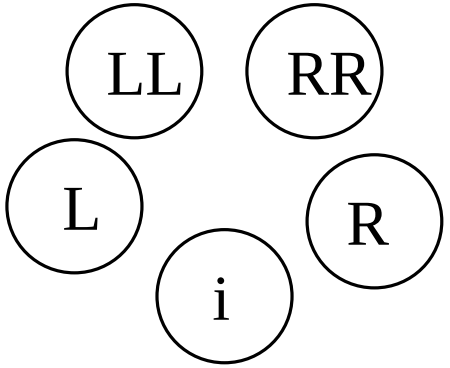
# Dining-Philosophers Problem: Solution 2

- 식사 할 조건
  - Test()함수 안에서 검사하는 LEFT와 RIGHT의 상태가 EATING이 아니어야 한다.
  - Test()를 만족하면, signal(self[i])에 의해 self[i]의 값은 1 이 되므로 wait(self[i])에서 block되질 않고 식사를 진행 한다.
    - self[i]의 초기 값은 0 임을 기억할 것

```
{
    wait(mutex);
    state[i] = HUNGRY;
    test(i);
    signal(mutex);
    wait(self[i]);
}
```

# Dining-Philosophers Problem: Solution 2

- 식사를 마치면, 좌, 우 철학자의 test()함수를 실행한다.
  - 이때, 철학자 L과 R은 i에 의해 take_chopsticks()에서 wait()함수에 의해 대기중




  - 철학자 i에 의해 실행한 test(LEFT)
    - i와 LL의 상태를 확인
  - 철학자 i가 실행한 test(RIGHT)
    - i와 RR의 상태를 확인

```
{
    wait(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    signal(mutex);
}
```

- 철학자 i가 식사 중인 동안 LL 혹은 RR 철학자의 상태가 EATING으로 바뀐다 하더라도, test(LEFT), test(RIGHT) 에서 LL과 RR의 상태를 확인 후 signal을 수행하므로 동기화에 문제가 없다

# Good solution for synchronization problem

- Check points
  - Data consistency
  - Concurrency
  - Deadlock
  - Starvation

# Synchronization Examples

- Windows
- Linux
- Pthreads

# Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides dispatcher objects user-land which may act mutexes, semaphores, events, and timers
  - Events
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either signaled-state (object available) or non-signaled state (thread will block)

# Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Semaphores
  - atomic integers
  - spinlocks
  - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

# Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variable
- Non-portable extensions include:
  - read-write locks
  - spinlocks