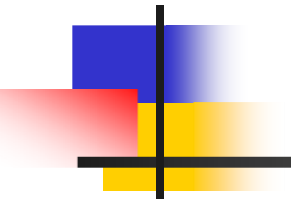
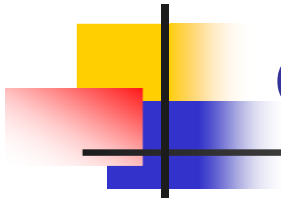


Operating Systems
(Synchronization & Deadlocks)

Chapter 6-8

These lecture materials are modified from the lecture notes written by A. Silberschatz, P. Galvin and G. Gagne.

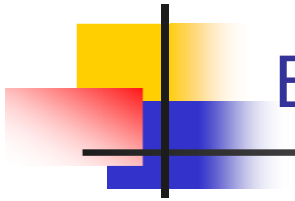
August, 2022



Outline

- Background
- The Critical-Section Problem
- Hardware solution
- OS supported SW solution
- Language-Level solution
- Deadlock

Background



Background

- cooperating process는 시스템에서 실행 중인 다른 프로세스에 영향을 미치거나 영향을 받을 수 있는 프로세스
- 그러나 공유 데이터에 대한 동시 액세스는 데이터 불일치를 초래할 수 있음.

Background

- Process synchronization
  - <Example> Too much milk problem

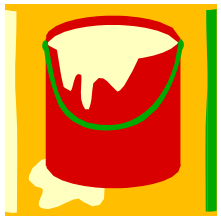
Person 1 **Process 1**

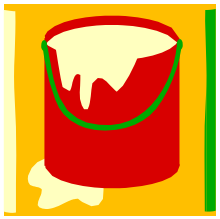
Look in fridge, out of milk
Leave for store
Arrive at store
Buy milk
Arrive home; put milk away

Shared resource

Person 2 **Process 2**

Look in fridge, out of milk
Leave for store
Arrive at store
Buy milk
Arrive home; put milk away





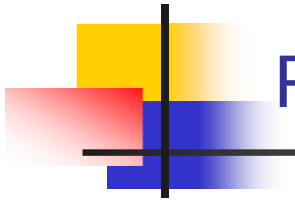
- To remedy this problem, locking mechanism is used

Person 1

Look in fridge, out of milk
Leave for store
Arrive at store
Buy milk
Arrive home; put milk away

Person 2

Look in fridge, out of milk
Leave for store
Arrive at store
Buy milk
Arrive home; put milk away



Race condition

- 만약 두 프로세스가 동시에 어떤 변수의 값을 바꾼다면 프로그래머의 의도와는 다른 결과가 나옴.

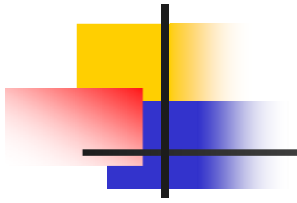
- **Race condition:** 프로세스가 어떤 순서로 데이터에 접근하느냐에 따라 결과 값이 달라질 수 있는 상황.

Producer

```
while (true) {
        /* produce an item and put in nextProduced */
                while (count == BUFFER_SIZE)
                        ; // do nothing
                buffer [in] = nextProduced;
                in = (in + 1) % BUFFER_SIZE;
                count++;
}
```

Consumer

```
while (true)  {
                while (count == 0)
                        ; // do nothing
                        nextConsumed =  buffer[out];
                         out = (out + 1) % BUFFER_SIZE;
                        count--;
}
```



count++ could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

count-- could be implemented as

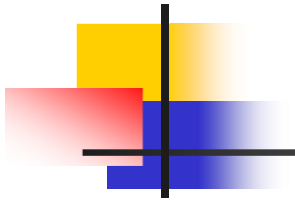
```
register2 = count  regis
ter2 = register2 - 1 cou
nt = register2
```

Originally, count=5

count++;	count--;
Producer	**Consumer**
register1 = count {register1 = 5}	register2 = count {register2 = 5}
register1 = register1+1 {register1 = 6}	register2 = register2-1 {register2 = 4}
count = register1 {count = 6}	count = register2 {count = 4}

Count is 4 !

Race condition



How to avoid race condition ?

count++;

Producer

```
register1 = count      {register1 = 5}
register1 = register1+1 {register1 = 6}
count = register1      {count = 6}
```

Atomic operation

count--;

Consumer

```
register2 = count      {register2 = 5}
register2 = register2-1  {register2 = 4}
count = register2      {count = 4}
```

Atomic operation

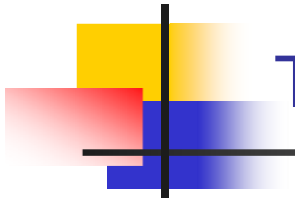
The Critical-Section Problem

The Critical-Section Problem

- **Critical-Section :** 프로세스가 Critical-Section내에 진입한 경우, 다른 프로세스가 Critical-Section에 진입할 수 없음.
  - 공유 리소스가 있는 코드를 Critical-Section로 지정함.

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

Figure 6.1 General structure of a typical process.



The Critical-Section Problem

- **The Critical-Section Problem :** Critical-Section에 진입할 때 Race condition이 발생할 수 있다는 문제가 있음.
  - OS가 비선점 스케쥴링을 채택한 경우 문제가 발생하지 않음. 그런 경우는 현대의 OS에서는 거의 없음.

The Critical-Section Problem

- 문제를 풀기 위한 3가지 요구사항
  - **1. Mutual exclusion(상호 배제):**
    어떤 프로세스가 Critical-Section에서 실행되고 있으면 다른 프로세스가 Critical-Section에서 실행될 수 없음
  - **2. Progress(진행):**
    Critical-Section에서 실행 중인 프로세스가 없다면 Critical-Section에 진입할 수 있음.
  - **3. Bounded waiting:**
    프로세스가 Critical-Section에 들어가도록 요청한 경우,
    요청한 프로세스가 Critical-Section에 들어가기 전까지는 기존에 임계섹션에 들어간 프로세스는 다시 들어갈 수 있는 횟수가 제한됨.
    즉 무한정으로 요청한 프로세스를 대기시켜서는 안됨.

Peterson's Solution

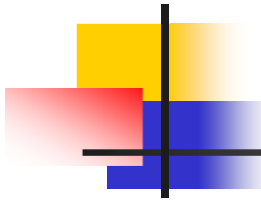
- Peterson's solution으로 Critical-Section problem를 해결할 수 있음
  - 변수 turn: Critical-Section에 들어갈 차례를 나타냄.
  - 변수 flag: Critical-Section에서 프로세스가 작업 중인지를 나타냄

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

-> 프로세스 j에게 turn을 양보

-> 프로세스 j의 flag가 true라면, 프로세스 j의 실행이 종료되는 것을 대기하겠다는 것을 의미
만약 false라면, 프로세스 i가 임계 구역에 진입함.

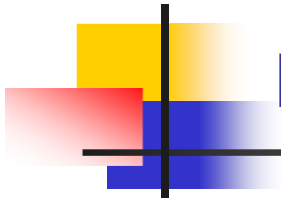
Figure 6.3 The structure of process P_i in Peterson's solution.



Peterson's Solution

- Critical-Section problem를 풀기 위한 3가지 요구사항
  - **1. Mutual exclusion(상호 배제) :**
    프로세스 i와 j가 함께 Critical-Section에 들어가려면, flag[i] = flag[j] = true여야 함.
    그러나 turn은 j 또는 i의 값을 가질 수밖에 없음.
    그러므로 하나의 프로세스는 반드시 대기를 하게 되므로 둘이 동시에 임계 영역에 진입하지 못함.
    따라서 mutual exclusion을 보장.

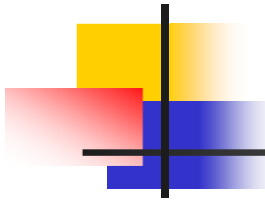
  - **2. Progress(진행):**
    프로세스 j가 Critical-Section에 들어가지 않았을 경우,
    flag[j]는 false이므로 프로세스 i는 Critical-Section에 진입가능함.
    따라서 Progress를 보장.



Peterson's Solution

- Critical-Section problem를 풀기 위한 3가지 요구사항
  - **3. Bounded waiting:**
    프로세스 j가 Critical-Section에서 나오기 직전,
    프로세스 j는 flag[j]를 false로 설정하고 turn을 i로 변경하므로,
    프로세스 i는 프로세스 j가 나온 후 Critical-Section에 진입 가능함.
    따라서 bound waiting을 보장.

- 3가지 요구사항을 모두 만족했으므로,
  Peterson's Solution은 Critical-Section problem를
  해결할 수 있음.



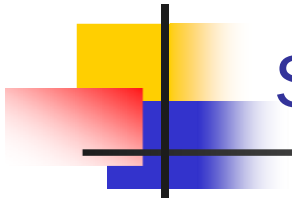
Peterson's Solution

- **문제점**

- 1. Busy waiting: 빈 반복문을 반복하기 때문에 **계속적으로 context switching이 발생하여** 오버헤드 발생

- 2. 현대 컴퓨터 아키텍쳐에서는 시스템 성능을 향상시키기 위해 프로세서 또는 컴파일러가 dependency이 없는 읽기 및 쓰기 작업의 순서를 바꿀 수 있음.

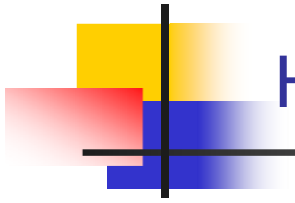
  만약 프로세스끼리 dependency가 없다고 자의적으로 판단해버릴 경우, **순서가 멋대로 바뀌어 Peterson's Solution로도 해결이 되지 않을 수 있음.**



Synchronization을 위한 다양한 방법

- Hardware solution
- OS supported SW solution
- Laqnguage-Level solution

Hardware solution



Hardware Support for Synchronization

- 1. Memory Barriers
- 2. Hardware Instructions
- 3. Atomic Variables

Memory Barriers

- memory barriers는 cpu나 컴파일러에게 특정 연산의 순서를 강제하도록 하는 기능.
- Peterson's Solution의 문제점을 해결함

```
while (!flag)
   memory_barrier();
print x;
```

It guarantee that the value of flag is loaded before the value of x.

Hardware Instructions

- 명령어가 동시에(각각 다른 코어에서) 실행되면 임의의 순서로 순차적으로 실행시키는 특수 하드웨어 명령을 사용하면 해결 가능함.
  - the test_and_set()
  - compare_and_swap()

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Figure 6.5 The definition of the atomic `test_and_set()` instruction.

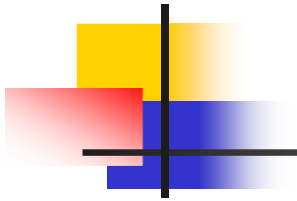
Atomic operation

1 return value = original value
2 set *target = true

Hardware Instructions

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

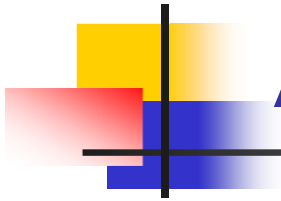
Figure 6.6 Mutual-exclusion implementation with `test_and_set()`.



```
boolean TestAndSet (boolean *target)
 {
        boolean rv = *target;
        *target = TRUE;
        return rv:
 }
```

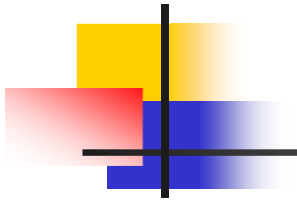
*target = FALSE	rv= FALSE	*target= TRUE	Next code is executed, but the next access will be blocked
*target =TRUE	rv= TRUE	*target= TRUE	Blocked

```
while (TestAndSet(&lock)) ;
Next code;
```



Atomic Variables

- 기본 데이터 유형에 대한 Atomic operation을 제공하는 변수.
- Atomic Variables은 카운터가 증가할 때와 같이 업데이트되는 동안 단일 변수에 대한 race condition이 있을 수 있는 상황에서 mutual exclusion를 보장하기 위해 사용할 수 있음
- Atomic Variables를 지원하는 대부분의 시스템은 Atomic Variables에 액세스하고 조작하기 위한 기능뿐만 아니라 special atomic data types을 제공

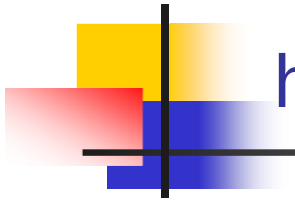


- Atomic variables in Linux
  - A special type “atomic_t” ensures that atomic operations are used only with these special types

```
atomic_t v;
atomic_t u=ATOMIC_INIT(0);
atomic_set(&v, 4);
atomic_add(2, &v);
atomic_inc(&v)
printk(“%d₩n”, atomic_read(&v));
atomic_dec_and_test(&v);
```

<asm/atomic.h>

OS supported SW solution



higher-level software tools for synchronization

- 1. Mutex Locks
- 2. Semaphores

좋은 설명 : https://worthpreading.tistory.com/90

Mutex Locks

- available 변수가 true인 단 하나의 프로세스만이 Critical-Section에 들어갈 수 있음.
- available는 단 하나의 프로세스만 true가 될 수 있음.

```
while (true) {

    acquire lock

        critical section

    release lock

        remainder section

}
```

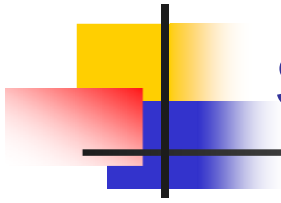

Figure 6.10 Solution to the critical-section problem using mutex locks.

Semaphore

- Semaphore는 wait()와 signal()이라는 2가지 atomic operation에 의해서만 엑세스 가능한 정수 변수 S라고 함.
- 정수는 1 혹은 그 이상일 수 있음. 이 변수는 오직 wait()와 signal()이라는 2가지 atomic operation에 의해서만 엑세스 가능.

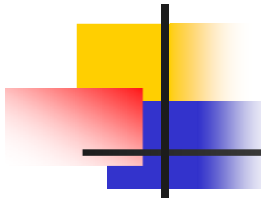
```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

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



Semaphore

- Semaphore의 값이 0이면 어떤 프로세스도 Critical-Section에 들어갈 수 없도록 함.
- 0보다 크면, 프로세스가 Critical-Section에 들어오면 Semaphore의 값을 1 감소시킴.
- 프로세스가 Critical-Section에서 나갈 때에는 Semaphore의 값을 1 증가시킴.



Semaphore

- 세마포어의 문제
  - 코딩이 어려움
  - timing errors가 일어날 수 있음
  - 각 프로세스는 Critical-Section 들어가기 전에 wait(mutex)를 실행하고 이후에 signal(mutex)을 실행해야 함
    - 이 순서가 관찰되지 않으면 허용한 것보다 많은 프로세스가 동시에 Critical-Section에 진입할 수도 있음.
    - 혹은 deadlock이 발생할 수도 있음.

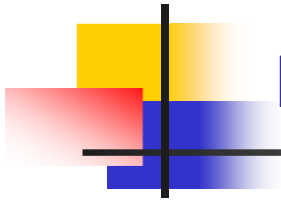
```
while (true) {
        signal (mutex);

        signal (mutex);
    }
```

Mutual exclusion property is violated

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

        wait (mutex);
    }
```

Deadlock occurs

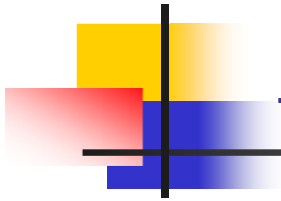
Language-Level solution



Language-Level software tools for synchronization

- 1. Moniter
- 2. Condition variable

좋은 설명 : https://worthpreading.tistory.com/90



모니터(Moniter)

- 언어 레벨에서 제공하는 동기화 기법

- 한 번에 하나의 프로세스만 모니터 내에서 활성화될 수 있음
- 공유 객체, 프로시저로 구성됨.
- 공유 객체는 프로시저로만 접근 가능함.

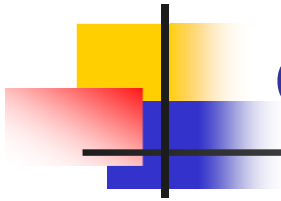
모니터(Moniter)

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;

   void acquire(int time) {
      if (busy)
         x.wait(time);
      busy = true;
   }

   void release() {
      busy = false;
      x.signal();
   }

   initialization_code() {
      busy = false;
   }
}
```

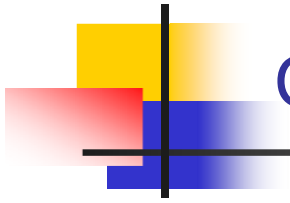
Figure 6.14 A monitor to allocate a single resource.

- 자바 좋은 예시: https://lordofkangs.tistory.com/28

Condition variable



- Type이 condition인 변수
  순서를 보장하기 위해 사용.
  - x.wait() : 이 함수를 호출한 프로세스는 대기
  - x.signal() : 대기중인 프로세스가 있다면 wait() 를 호출한 프로세스 중 하나가 실행됨. 대기중인 프로세스가 없다면 아무일도 일어나지 않음
- Signal 을 보낸다는 것은 해당 프로세스가 Lock 을 얻었음을 의미
  - Semaphore 의 signal() 이 항상 S++ 을 하는 것과는 다름.



Condition variable

- Signal and wait : signal 을 주고, 내가 대기함으로서 기다리고 있던 다른 프로세스가 먼저 수행되도록 양보
- Signal and continue : signal 을 주고, 내가 먼저 실행된 뒤에 sleep() 하여 다른 프로세스가 수행되도록 함

Condition variable

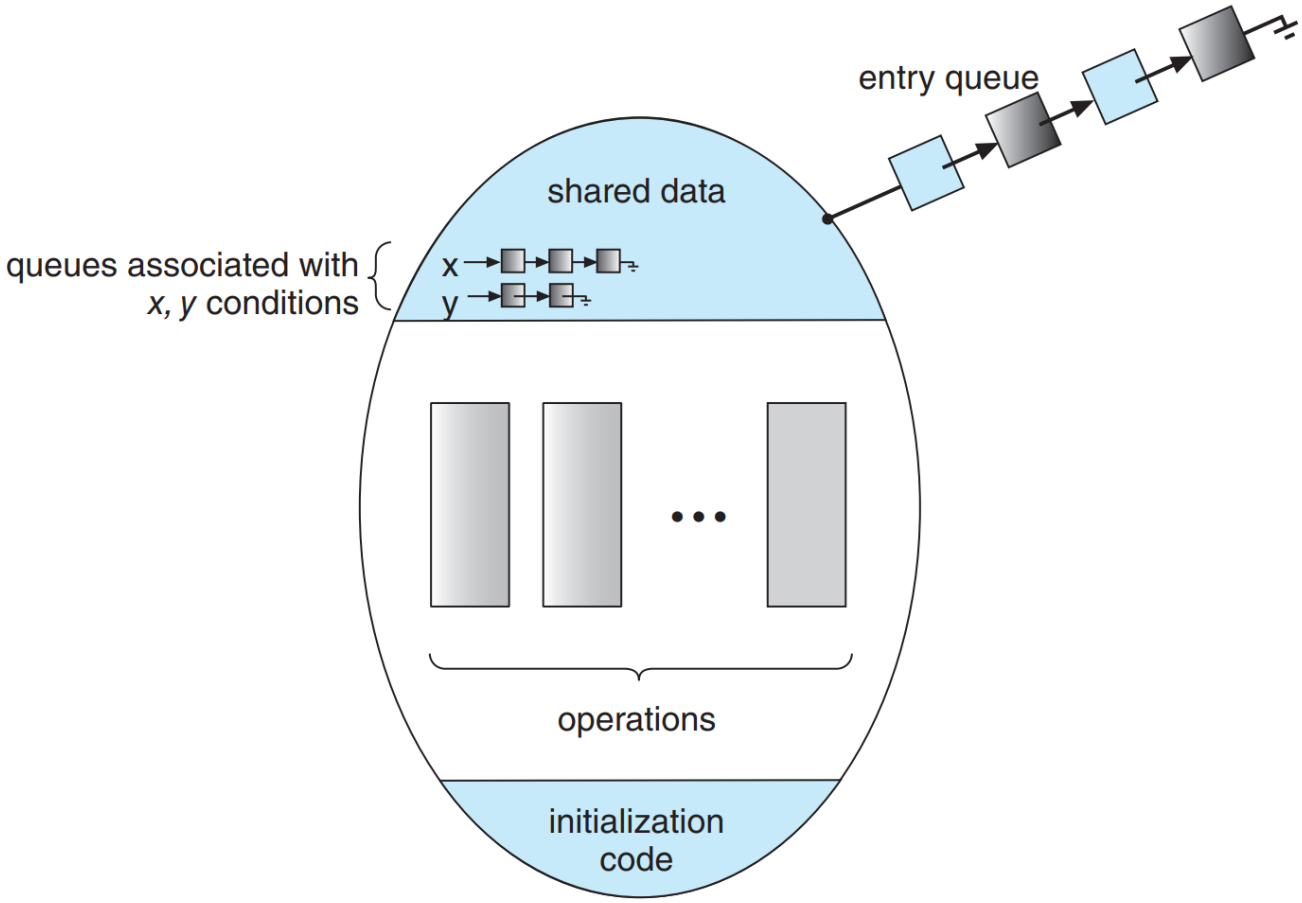
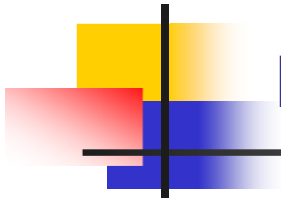


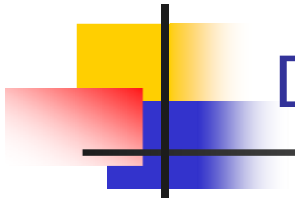
Figure 6.13 Monitor with condition variables.

Deadlock



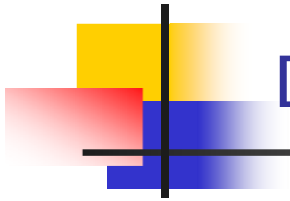
Deadlock

- 프로세스들이 서로가 가진 자원을 기다리며 block 되어 더 이상 진행이 될 수 없는 상태



Deadlock

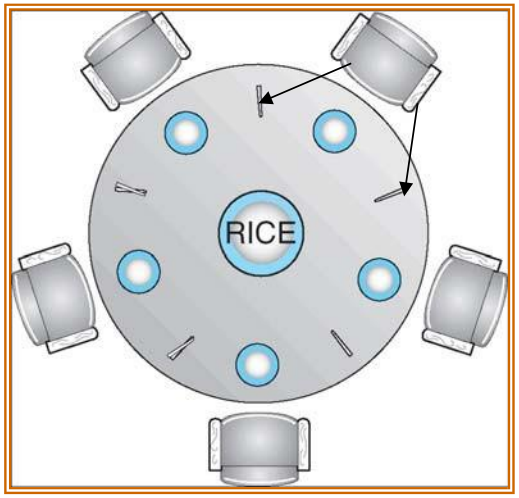
- 프로세스가 리소스를 사용하기 위해서는 요청해야함.
- Request -> Use -> Release



Deadlock

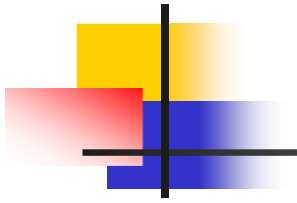
- 프로세스가 리소스를 사용하기 위해서는 요청해야함.
- Request -> Use -> Release

- Dining-Philosophers Problem



Shared data

- Bowl of rice (data set)
- Semaphore chopstick [5] initialized to 1

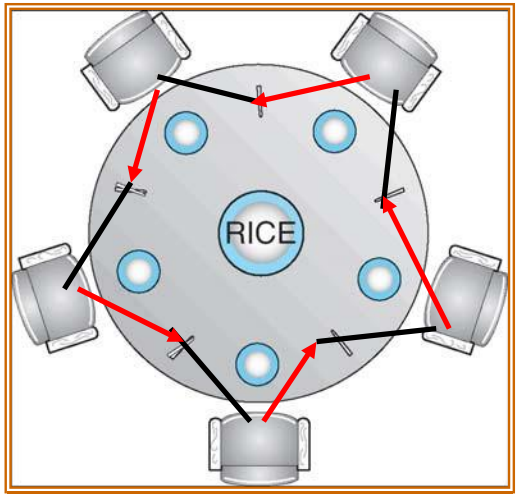


- Philosophers

2 chopsticks

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

        //  eat

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

        //  think

}
```

- Problem is a deadlock !



Deadlock Characterization

- 데드락 발생에는 네 가지 조건이 있고, 이 중 하나라도 충족하지 않으면 데드락은 발생하지 않음

- **Mutal Exclusion (상호 배제):**
- 어떤 리소스에 접근할 때 순간적으로 하나의 프로세스만 접근할 수 있음.

- **Hold and Wait (점유 및 대기):**
- 프로세스는 적어도 하나의 리소스를 점유하고 있고, 다른 리소스의 추가 획득을 대기하고 있음.
- 프로세스가 리소스를 점유하고 있지 않으면, 애초에 데드락이 일어나지 않음.

Deadlock Characterization

- **No preemption (선점 불가)**
- 리소스는 선점될 수 없음.
- 선점이 되어버리면 프로세스가 실행을 종료하니 데드락이 일어나지 않음

- **Circular Wait (순환 대기)**
- 프로세스가 순환적으로 서로를 기다리고 있음

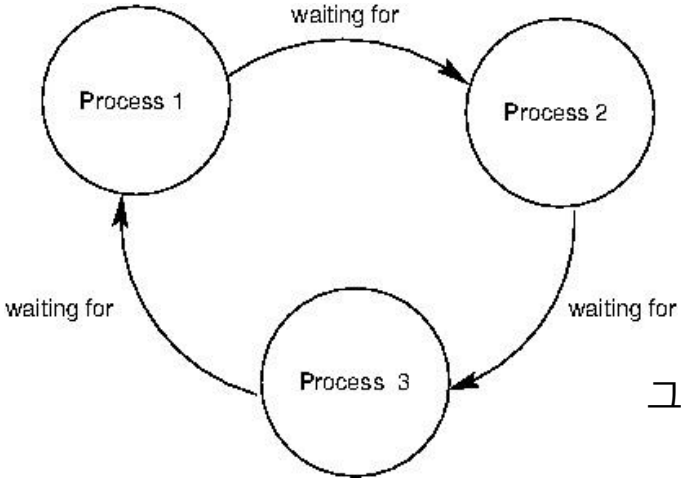
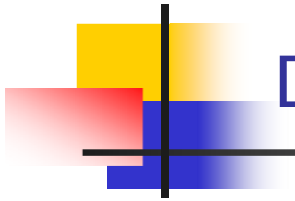
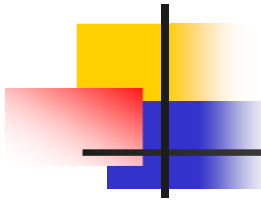


그림 출처: http://web.cs.ucla.edu/



Deadlock Prevention

- 데드락을 완전히 방지하려면, 데드락 발생의 네 가지 조건 중 하나를 해결하면 됨.



Deadlock Prevention

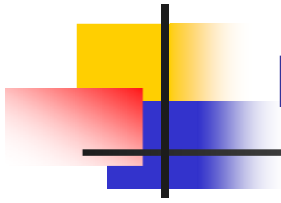
- **Mutual Exclusion**
- 리소스 접근 시 여러 프로세스가 접근하게 함.
- 제한된 리소스를 여러 프로세스가 공유하게 되면 다른 문제들이 발생하므로 본질적으로 불가능함.

- **Hold and Wait**
- 프로세스가 리소스를 점유하고 있지 못하게 하는 방법
- 프로세스가 리소스를 요청할 때, 가지고 있는 리소스를 모두 내려놓고 리소스를 요청하도록 함.
- 반드시 추가적으로 리소스가 요청해야할 경우들을 고려하지 못해 실용성이 없음.

Deadlock Prevention

- **No premmption**
- 다른 프로세스가 리소스를 점유하고 있을 때, 이를 선점하여 실행하는 방법이다.
- 선점당한 프로세스는 어떤 작업을 하든 일단 waiting 상태로 돌입한다.
- 다시 프로세스를 실행하려면 이전 리소스와 새 리소스를 되찾아야만 다시 시작할 수 있음.
- 이에 대한 오버헤드가 더 클 것으로 보이므로 실용성이 없음.

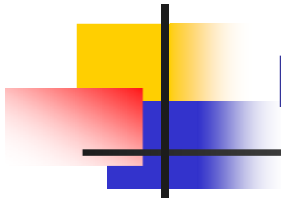
- **Circular Wait**
- **리소스에 순서를 부여하는 방법**
- **각 프로세스가 순서대로 리소스를 요청하게끔 요구**
- **제일 실용적인 방법이지만, starvation 상태가 발생할 가능성**



Deadlock Prevention

- **Circular Wait**
- 리소스에 순서를 부여하는 방법
- 각 프로세스가 순서대로 리소스를 요청하게끔 요구
- 제일 실용적인 방법이지만, starvation 상태가 발생할 가능성이 높음.

- Deadlock을 방지하는 것은 쉽지 않음.



Deadlock

- 윈도우, 리눅스 등의 대부분의 OS는 Deadlock에 대한 아무런 조치도 취하지 않음.
- 그 이유는, Deadlock이 매우 드물게 발생하기 때문에 Deadlock에 대한 조치 자체가 더 큰 오버헤드이기 때문.
- Deadlock이 발생한 경우, 모르는 척하고 그대로 놔둠. 시스템이 비정상적으로 작동하는 것을 사용자가 느낀 후 직접 프로세스를 죽이는 등의 방법으로 대처하도록 함.
- 그러므로 Deadlock이 발생하지 않도록 개발 때 잘 해주어야 함.

Chapter 6-8
Finish